37th International Symposium on Distributed Computing

DISC 2023, October 10-12, 2023, L'Aquila, Italy

Edited by

Rotem Oshman

LIPICS

Editors

Rotem Oshman
Blavatnik School of Computer Science, Tel Aviv University, Israel
roshman@tau.ac.il

ACM Classification 2012
Software and its engineering → Distributed systems organizing principles; Computing methodologies → Distributed computing methodologies; Computing methodologies → Concurrent computing methodologies; Hardware → Fault tolerance; Information systems → Data structures; Networks; Theory of computation; Theory of computation → Models of computation; Theory of computation → Design and analysis of algorithms

ISBN 978-3-95977-301-0

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-301-0.

Publication date
October, 2023

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at https://portal.dnb.de.

Digital Object Identifier: 10.4230/LIPIcs.DISC.2023.0

LIPIcs – Leibniz International Proceedings in Informatics

LIPIcs is a series of high-quality conference proceedings across all fields in informatics. LIPIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

ISSN 1868-8969

https://www.dagstuhl.de/lipics

Contents

Preface

Welcome to DISC 2023, the 37th International Symposium on Distributed Computing, held on October 10–12, 2023, in L'Aquila, Italy. DISC is an international forum on the theory, design, analysis, and implementation of distributed systems and networks, focusing on distributed computing in all its forms. DISC is organized in cooperation with the European Association for Theoretical Computer Science (EATCS).

DISC 2023 received 125 submissions in the "regular paper" category, and 14 submissions in the "brief announcement" category. The program was selected by a program committee consisting of 24 full members and 4 half-load members. The program committee was assisted by 127 external reviewers. As in previous years, the committee used a relaxed form of double-blind reviewing, where the submissions themselves were anonymous, but authors were permitted to disseminate their work by uploading it to online repositories or by giving talks about it. Each submission was evaluated by at least three reviewers, and final decisions were made during a 2-day virtual PC meeting. 34 regular papers were accepted (an acceptance rate of 27%), and 13 brief announcements. The keynote talks at DISC 2023 were given by Tal Rabin on behalf of the winners of the 2023 Dijkstra Award, by Amos Korman, and by Lorenzo Alvisi.

The following two awards are jointly sponsored by DISC and the ACM Symposium on Principles of Distributed Computing (PODC):

- The 2023 Edsger W. Dijkstra Prize in Distributed Computing was presented at DISC 2023. The award was given to Michael Ben-Or, Shafi Goldwasser and Avi Wigderson for their paper "Completeness Theorem for Non-Cryptographic Fault-Tolerant Distributed Computation", to David Chaum, Claude Crépeau and Ivan Damgård for their paper "Multiparty Unconditionally Secure Protocols", and to Tal Rabin and Michael Ben-Or for their paper "Verifiable Secret Sharing and Multiparty Protocols with Honest Majority".
- The 2023 Principles of Distributed Computing Doctoral Dissertation Award was presented at PODC 2023. The award was given to Dr. Siddhartha Jayanti for his dissertation "Simple, Fast, Scalable, and Reliable Multiprocessor Algorithms", and to Dr. Dean Leitersdorf for his dissertation "Fast Distributed Algorithms via Sparsity Awareness.".

This volume includes the citations for the best paper and best student paper awards at DISC 2023, as well as the citations for the 2023 Edsger W. Dijkstra Prize in Distributed Computing, which was presented at DISC 2023, and for the Best Dissertation Award, which was presented at PODC 2023.

I would like to warmly thank everyone who contributed to DISC 2023: the authors who submitted their work to PODC, the PC members and external reviewers, the keynote speakers, the organizing committee, the workshop chairs, members of the award committees, and participants of the conference. I am also grateful to the members of the steering committee and to former chairs of DISC, who shared their invaluable experience and advice; to EATCS for their support; and to the staff of Schloss Dagstuhl – Leibniz-Zentrum für Informatik for their help in preparing these proceedings.

October 2023

Rotem Oshman
DISC 2023 Program Chair

Organization

DISC, the International Symposium on Distributed Computing, is an annual forum for presentation of research on all aspects of distributed computing. It is organized in cooperation with the European Association for Theoretical Computer Science (EATCS). The symposium was established in 1985 as a biannual International Workshop on Distributed Algorithms on Graphs (WDAG). The scope was soon extended to cover all aspects of distributed algorithms and WDAG came to stand for International Workshop on Distributed AlGorithms, becoming an annual symposium in 1989. To reflect the expansion of its area of interest, the name was changed to DISC (International Symposium on DIStributed Computing) in 1998, opening the symposium to all aspects of distributed computing. The aim of DISC is to reflect the exciting and rapid developments in this field.

Program Chair

Rotem Oshman	Tel Aviv University (Israel)

Program Committee

Carole Delporte-Gallet	IRIF, Université Paris Cité (France)
Corentin Travers	LIS/Université d'Aix-Marseille (France)
Fabian Kuhn	University of Freiburg (Germany)
Gillat Kol	Princeton University (USA)
Gregory Chockler	University of Surrey (UK)
Guy Goren	Protocol Labs (Israel)
Jara Uitto	Aalto University (Finland)
Jennifer Welch	Texas A&M University (USA)
Juho Hirvonen	Helsinki Institute for Information Technology and Aalto University (Finland)
Kunal Agrawal	Washington University in St. Louise (USA)
Laurent Feuilloley	CNRS / Université de Lyon (France)
Manuela Fischer	ETH Zurich (Switzerland)
Mark Moir	Oracle Labs (USA)
Maurice Herlihy	Brown University (USA)
Nicola Santoro	Carleton University (Canada)
Oded Naor	Technion and StarkWare (Israel)
Orr Fischer	Weizmann Institute (Israel)
Paul G. Spirakis	University of Liverpool (UK)
Pedro Montealegre	Adolfo Ibáñez University (Chile)
Petr Kuznetsov	INFRES, Telecom Paris (France)
Petra Berenbrink	University of Hamburg (Germany)
Rafael Pass	Tel Aviv University, Israel and Cornell University (USA)
Rati Gelashvili	Aptos (USA)
Rob Johnson	VMWare (USA)
Siddhartha Visveswara Jayanti	Google Research and MIT (USA)
Tania Lorido Botran	Roblox (USA)
Wojciech Golab	University of Waterloo (Canada)
Zarko Milosevic	Informal Systems (Canada)

Organizing Committee

Alkida Balliu (co-Chair)	Gran Sasso Science Institute (Italy)
Dennis Olivetti (co-Chair)	Gran Sasso Science Institute (Italy)
Yannic Maus (Workshops Chair)	TU Graz (Austria)
Tijn de Vosa (Environmental co-Chair)	University of Salzburg (Austria)
Laurent Feuilloley (Environmental co-Chair)	Université Lyon 1 and CNRS (France)
William K. Moses Jr. (Publicity Chair)	Durham University (UK)
Gianlorenzo D'Angelo	Gran Sasso Science Institute (Italy)

Steering Committee

Jukka Suomela (Chair)	Aalto University (Finland)
Hagit Attiya (Vice Chair)	Technion (Israel)
Seth Gilbert (2021 PC Chair)	NUS (Singapore)
Christian Scheideler (2022 PC Chair)	University of Paderborn (Germany)
Rotem Oshman (2023 PC Chair)	Tel Aviv University (Israel)
Calvin Newport (Member-at-large)	Georgetown University (USA)
Moti Medina (Treasurer)	Bar-Ilan University (Israel)

External Reviewers

Davide Frey	Shantanu Das	Luciano Freitas de Souza
James Aspnes	Peter Robinson	George Skretas
Faith Ellen	Théo Pierron	Leonid Barenboim
Hossein Vahidi	Christopher Hahn	Grzegorz Stachowiak
Chien-Chih Chen	Ami Paz	Adam Gańczorz
Subhash Bhagat	Martín Ríos-Wilson	Alexander Spiegelman
Ivan Rapaport	Michael Elkin	Hagit Attiya
Nirupam Gupta	Gal Sela	Matej Pavlovic
Anup Joshi	Balaji Arun	Yuanhao Wei
Michiko Inoue	Michal Dory	Hagit Attiya
Felix Biermeier	Thomas Nowak	Ahmed Fahmy
Adam Morrison	Mikaël Rabie	Matej Pavlovic
Lewis Tseng	Ivan Rapaport	Giuseppe Prencipe
Euripides Markou	Arnaud Labourel	Raïssa Nataf
Gal Assa	Mélanie Cambus	Yannic Maus
Hafiz Imtiaz	Sergio Rajsbaum	Giuliano Losa
Lukas Hintze	Uri Meir	Chien-Chih Chen
Andrei Tonkikh	Dennis Olivetti	Francesco d'Amore
Achour Mostéfaoui	Ran Gelles	Josu Doncel
Gupta Nirupam	Tomer Koren	Yi-Jun Chang
Yadu Vasudev	Talya Eden	Sebastian Siebertz
Matthias Függer	Diana Ghinea	Peter Robinson
Sucharita Jayanti	Gal Assa	Daniel Collins
Armando Castaneda	Leqi Zhu	Chen-Da Liu Zhang
Yann Disser	Clément Legrand-Duchesne	Yitong Yin
Tijn de Vos	Michal Dory	Achour Mostéfaoui

Hsin-Hao Su
Shang-En Huang
Chetan Gupta
Zhuolun Xiang
Sergio Rajsbaum
Marios Mavronicolas
Peter Davies
Lioba Heimbach
Themistoklis Melissourgos
Chryssis Georgiou
Anissa Lamani
Stéphane Devismes
Laurent Viennot
Hendrik Molter
Hamed Hosseinpour
Mina Dalirrooyfard
Jeff Giliberti
Tom Friedetzky
Peter Kling
Andrea Richa
Saku Peltonen
Christoph Lenzen
Gokarna Sharma
Armando Castaneda

Shreyas Pai
William K. Moses Jr.
Sara Tucci-Piergiovanni
Saeed Akhoondian Amiri
Ivan Rapaport
Alan Kuhnle
Michel Raynal
Maximilian Hahn-Klimroth
Yi-Jun Chang
Keren Censor-Hillel
Gregory Schwartzman
Rustam Latypov
Seri Khoury
Marten Maack
Gal Assa
Malin Rau
Benjamin Jauregui
Yukiko Yamauchi
Mĺanie Cambus
Giorgi Nadiradze
Ami Paz
Xiaorui Sun
Hagit Attiya

Darya Melnyk
Leonid Barenboim
Hao Tan
Christoph Grunau
Valerie King
Elad Michael Schiller
George Giakkoupis
Thomas Nowak
Xavier Defago
Gabriele Di Stefano
Aaron Schild
Nina Klobas
Eric Ruppert
Giuseppe Antonio Di Luna
Igor Zablotchi
Slobodan Mitrović
João Paulo Bezerra
Shantanu Das
Benjamin Jauregui
Tigran Tonoyan
Ioan Todinca
Quentin Bramas
Alexander Spiegelman

Acknowledgements

DISC 2023 acknowledges the use of Easychair for handling submissions and managing the review process and LIPIcs for producing and publishing the proceedings.

DISC is organized in cooperation with the European Association for Theoretical Computer Science (EATCS).

Awards

Best Papers

The DISC Program Committee has selected the following two papers to receive the DISC 2023 best paper award:

Every Bit Counts in Consensus
by Pierre Civit, Seth Gilbert, Rachid Guerraoui, Jovan Komatovic, Matteo Monti and Manuel Vidigueira.

This paper improves the space complexity of multi-valued consensus by presenting an algorithm that requires only $O(n^{1.5}L+n^{2.5}k)$ bits for consensus on L-bit values (with security parameter k), an improvement of $\sqrt{n}$ upon prior work. Moreover, the paper devises a version of the protocol that uses stronger cryptographic assumptions – namely, the existence of STARK proofs – and achieves near-optimal bit complexity, $O(nL+n^2\text{poly}(k))$. Multi-valued consensus is an important problem in practice, where the value being agreed upon is often very large, and the paper uses interesting and novel techniques to achieve its strong results.

On the Node-Averaged Complexity of Locally Checkable Problems on Trees
by Alkida Balliu, Sebastian Brandt, Fabian Kuhn, Dennis Olivetti and Gustav Schmid.

This paper studies the node-averaged round complexity locally-checkable labeling (LCL) problems. The usual complexity measure in the LOCAL model is the *worst-case* round complexity across all nodes. The paper establishes relationships between the worst-case and the node-averaged complexity of LCL problems in trees, showing that every LCL problem whose worst-case complexity is $O(\log n)$ admits an algorithm with node-averaged complexity $O(\log^* n)$, and that every LCL problem with worst-case complexity $\Theta(n^{1/k})$ requires node-averaged complexity $\tilde{\Omega}(n^{1/(2^k-1)})$, which is in some cases tight. Node-averaged complexity is a new and interesting complexity measure, and the results of the paper show that node-averaged complexity can be significantly better than the worst-case complexity, making it a worthwhile measure to study.

Best Student Paper

The DISC Program Committee has selected the following paper to receive the DISC 2023 best student paper award:

The FIDS Theorems: Tensions between Multinode and Multicore Performance in Transactional Systems
by Naama Ben-David, Gal Sela and Adriana Szekeres

This paper studies the performance of transactional systems that are both parallel and distributed, meaning that they both use multiple nodes and employ multiple cores per node. The paper shows that there is an inherent tradeoff between the scalability of the system, the speed with which the system commits transaction in good executions, and its fault tolerance. On the positive side, the paper shows that if any one of the three requirements is dropped, then it is possible to construct a system satisfying the other two.

The tradeoff established and formalized in this paper is timely and relevant to large-scale transactional systems, and serves as an analog for the famous CAP theorem for this setting.

2023 Principles of Distributed Computing Doctoral Dissertation Awards

Many exceptionally high-quality doctoral dissertations were submitted for the 2023 Principles of Distributed ComputingDoctoral Dissertation Award. After careful deliberation, the award committee decided to share the award between:

- Dr. Siddhartha Jayanti for his dissertation “Simple, Fast, Scalable, and Reliable Multiprocessor Algorithms.”
- Dr. Dean Leitersdorf for his dissertation “Fast Distributed Algorithms via Sparsity Awareness.”

Dr. Siddhartha Jayanti completed his PhD on November 27th 2022, under the supervision of Prof. Julian Shun, at MIT. In his thesis, Dr. Jayanti identifies simplicity, speed, scalability, and reliability as four core design goals for multiprocessor algorithms, and designs and analyzes algorithms that meet these goals. The thesis comprises a vast number of novel results in the scope of distributed and concurrent synchronization. His algorithmic contributions include a scalable algorithm for concurrent union-find, a wait-free linearizable, fast array data structure that supports standard array operations in constant time and optimal space, and mutual exclusion (lock) algorithms with optimal complexity for real-time and persistent memory systems. Dr. Jayanti also defines a generalization of the fundamental wake-up problem, permitting him to prove fundamental new hardness results for many standard data structures, including queues, stacks, priority queues, counters, and union-find data structures. Moreover, he devises a novel simple-to-use technique for producing machine-verified proofs of the correctness (linearizability and strong linearizability) of concurrent algorithms, and successfully applied this method to verify fundamental data multicore data structures, such as queues, union-find, and snapshot objects. Dr. Jayanti also analyzes a parallel and asynchronous Markov Chain Monte Carlo (MCMC) algorithm, showing that it can speed-up the collection of low-bias statistics from probability distributions of interest in Machine Learning and Statistical Physics. Finally, Dr. Jayanti’s PhD dissertation introduces the Samskrtam Technical Lexicon Project, which incorporates ideas from Panini’s generative grammar to facilitate the coining of new technical vocabulary and increase the availability of scientific education and literature in Indian and other world languages. As part of the project, he uses Sanskrit roots to coin words for several concepts in algorithms and multiprocessors in Telugu, and contributes the first modern computer science research paper in the Telugu language, which has about 100 million speakers around the world.

Dr. Dean Leiterdorf completed his thesis on May 14th, 2022, under the supervision of Prof. Keren Censor-Hillel, at the Technion. In his thesis, Dr. Leitersdorf designs fast distributed algorithms for sparse matrix multiplication and demonstates their usefulness by applying them to shortest path and subgraph existence problems. Applications of matrix multiplication are found in many fields, including scientific computing, statistics, machine learning, and quantum computing, and therefore fast algorithms for matrix multiplication are critical for these. Dr. Leitersdorf does not just come up with solutions that can exploit the sparsity of the input matrices but also the sparsity of the output matrix, which allows him to come up with a large number of results for different communication models that partially significantly improve the state of the art. Among these are constant-round algorithms for computing graph spanners and approximate all-pairs-shortest-paths as well as constant-round algorithms for computing the girth of the input graph up to an additive 1 in the Congested Clique model. Through reductions between various models and a number of advanced techniques, Dr. Leitersdorf extends his results also to the CONGEST model, hybrid networks, and various other models. On top of this, he also designs a variety of algorithms that speed up clique detection in quantum computing settings and whose runtime breaks lower bounds known for classical distributed computing.

The award is sponsored jointly by the ACM Symposium on Principles of Distributed Computing (PODC) and the EATCS Symposium on Distributed Computing (DISC). It is presented annually, with the presentation taking place alternately at PODC and DISC. This year it was presented at PODC, to be held in Orlando, Florida USA, June 19-23, 2023.

The 2023 Principles of Distributed Computing Doctoral Dissertation Award Committee

Shlomi Dolev (Chair), BGU (Israel)
Rachid Guerraoui, EPFL (Switzerland)
Fabian Kuhn, University of Freiburg (Germany)
Woelfel Philipp, University of Calgary (Canada)
Christian Scheideler, Paderborn University (Germany)

2023 Edsger W. Dijkstra Prize in Distributed Computing

The 2023 Edsger W. Dijkstra Prize in Distributed Computing has been awarded to the papers

- Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation, by Michael Ben-Or, Shafi Goldwasser and Avi Wigderson (STOC 1988, 1–10).
- Multiparty Unconditionally Secure Protocols, by David Chaum, Claude Crèpeau and Ivan Damgård (STOC 1988, 11–19).
- Verifiable Secret Sharing and Multiparty Protocols with Honest Majority, by Tal Rabin and Michael Ben-Or (STOC 1989, 73–85).

for introducing Information-Theoretic Secure Multiparty Computations and showing how to achieve maximal resilience to malicious adversaries while providing unconditional security.

The area of Secure Multiparty Computation (MPC) answers the following fundamental question about distributed computations. How does a group of parties compute a function of their inputs while preserving not only correctness of the output but also, the secrecy of each party's input? Furthermore, this goal should be achieved in the case where some of the parties are malicious and try to foil the computation.

The awarded papers opened the vibrant area of MPC in the information theoretic setting, in which thousands of works have been published, and that is still going strong. Protocols in the information- theoretic model often are more efficient than their computational counterparts, in some cases by orders of magnitude, and thus have led to the most efficient state-of-the-art designs of MPC implementations. These protocols are an indispensable tool in the increasing demands for security and privacy in our modern digital society.

MPC and the techniques from the nominated papers have had tremendous impact on the broader area of cryptography with such results relating to zero-knowledge proofs and coding theory. They also have had far reaching impact on the broader area of theoretical computer science by providing a technical basis and inspiration for such results as locally random reductions, private information retrieval, and locally decodable codes.

The 2023 Dijkstra Award Committee

Magnús Halldórsson, Reykjavik University (chair)
Yehuda Afek, Tel-Aviv University
Idit Keidar, Technion
Rotem Oshman, Tel-Aviv University
Ulrich Schmid, TU Wien
Gadi Taubenfield, Reichman University

Colordag: An Incentive-Compatible Blockchain

Ittai Abraham
Intel Labs, Haifa, Israel

Danny Dolev
The Hebrew University of Jerusalem, Israel

Ittay Eyal
Technion, Haifa, Israel

Joseph Y. Halpern
Cornell University, Ithaca, NY, USA

Abstract

We present *Colordag*, a blockchain protocol where following the prescribed strategy is, with high probability, a best response as long as all miners have less than 1/2 of the mining power. We prove the correctness of Colordag even if there is an extremely powerful adversary who knows future actions of the scheduler: specifically, when agents will generate blocks and when messages will arrive. The state-of-the-art protocol, Fruitchain, is an ε-Nash equilibrium as long as all miners have less than 1/2 of the mining power. However, there is a simple deviation that guarantees that deviators are never worse off than they would be by following Fruitchain, and can sometimes do better. Thus, agents are motivated to deviate. Colordag implements a solution concept that we call *ε-sure Nash equilibrium* and does not suffer from this problem. Because it is an ε-sure Nash equilibrium, Colordag is an ε-Nash equilibrium **and** with probability $1 - \varepsilon$ is a best response.

2012 ACM Subject Classification Computing methodologies → Distributed computing methodologies; Theory of computation → Solution concepts in game theory; Security and privacy → Distributed systems security

Keywords and phrases Game theory, incentives, blockchain

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.1

Funding *Danny Dolev*: Supported by the Federmann Cyber-Security Center in conjunction with the Israel National Cyber Directorate.
Ittay Eyal: Supported by the Israel Science Foundation (grant No. 1641/18), Avalanche Foundation, and IC3.
Joseph Y. Halpern: Supported in part by AFOSR grant FA23862114029, MURI grant W911NF-19-1-0217, ARO grant W911NF-22-1-0061, and a grant from the Algorand Centers of Excellence program managed by the Algorand Foundation.

Acknowledgements We thank Roi Bar-Zur for comments on an early version of this manuscript, and the reviewers of the paper for their helpful comments.

1 Introduction

At the heart of Bitcoin [15] is the Nakamoto consensus protocol, which is based on proof-of-work [7, 12, 1]. The system participants, called *miners*, maintain a *ledger* that records all *transactions* – payments or so-called smart-contract operations. The transactions are batched into *blocks*; a miner can publish a block only by expending computational power, at a rate proportional to her computational power in the system. This rate is called *mining power*.

The Nakamoto consensus protocol achieves desirable ledger properties even against an adversary that controls $\alpha < 1/2$ of the mining power [10, 17, 13]. That is, as long as miners that control a majority of the cmoputing power follow the Nakamoto consensus protocol, security is guaranteed. But Nakamoto's protocol relies on *incentives*: The blocks form a tree, and each miner is rewarded for each block it generated that is included in the longest path (blockchain) in the tree. Unfortunately, following the Nakamoto consensus protocol is *not* a best response for miners that control a large fraction (but less than 1/2) of the total computational power [8, 16, 19]. For example, under some minimal modeling assumptions, even a coalition that controls 1/4 of the computational power can increase its reward by deviating from the Nakamoto Consensus protocol.[1] Stated differently, the Nakamoto consensus protocol is not a coalition-resistant equilibrium if there are coalitions that control more than 1/4 of the mining power.

Pass and Shi [18] make major progress with their Fruitchain protocol. In Fruitchain, the blocks form a dag (rather than a tree) with the longest chain determining rewards. However, miners are rewarded for a special type of block, called *fruit*. Each fruit block c is the child of a regular block b_1, and its miner is rewarded if a subsequent block b_2 points to the fruit, both blocks b_1 and b_2 are on the longest chain, and the path between them is shorter than some constant. If the longest chain is sufficiently long that the fruit c does not provide a reward, then c is called *stale*. Fruitchain is an *ε-Nash Equilibrium (NE)*, that is, a miner, even with mining power arbitrarily close to 1/2, can improve her revenue by only a negligible amount by deviating from the protocol. Like Bitcoin [17], Fruitchains is provably correct except with negligible probability in executions of length polynomial in the system's security parameter.

However, Fruitchain allows for a simple deviation by which any coalition can increase its utility without taking any risk: Specifically, a miner points only to its own fruit when generating blocks, ignoring fruit generated by others. This simple deviation dominates the prescribed protocol, as it creates a small probability that the ignored fruit will become stale, increasing the miner's relative revenue. While the probability increase is negligible in the staleness parameter, there is no risk to the miner. Moreover, if all agents are small and play this simple deviation, then the probability that any of them can point to its own fruit before it becomes stale is small; this results in a violation of the ledger properties, as progress becomes arbitrarily slow. Our conclusion is that ε-NE is an inappropriate solution concept in our setting; agents might still be incentivized to deviate from a ε-NE, although the benefit is small.

We present a more robust solution concept that we call *ε-sure NE*. A protocol is an ε-sure NE if, for any player, playing the prescribed protocol is a best response except for some set of runs (executions) that has probability at most ε. If utilities are bounded (as they are in our case), a ε-sure NE is an ε-NE, but the converse is not the case in general.

Our main contribution is the *Colordag* protocol, a PoW-based protocol that is an ε-sure NE, provided that each player controls less than half the total computational power. Like various solutions, starting from Lewenberg et al. [14, 22], Colordag constructs a directed acyclic graph rather than a tree. This graph is used for reward calculation; the ledger consists of a subset of blocks on the graph.

To achieve the required properties, Colordag makes use of three key ideas.

1. Due to the distributed nature of the system, two miners might generate a block before hearing of each others' blocks. The result is a *fork* where two blocks point to the same parent. This gives an advantage to the attacker, as the two blocks only extend the longest chain by one. To deal with forks that occur naturally, Colordag colors blocks randomly, and calculates the reward by looking at the graphs generated by the nodes of each color (technically, the *graph minors* of each color) separately. Adding more colors allows us to

[1] Under the most optimistic assumptions about the underlying network, this bound increases to only 1/3.

keep the original rate of block production, while mitigating the effects of forking: the fact that there are fewer blocks of a given color reduces the probability of forks in the minors. Previous work [10, 2, 24] randomly attributed properties to blocks for performance or resilience. In contrast, here coloring is used only for calculating the reward.

2. Colordag guarantees that, with high probability, malicious behavior (indeed, any deviation from the strategy) will not result in a higher reward for the deviating agent. The basic idea is that honest blocks of a given color will almost always be *acceptable*: they are on a chain that is almost the longest in its minor. Unacceptable blocks get no reward and do not affect the rewards of others. The approach is similar to Sliwinski and Wattenhofer's *block staling*; it is guaranteed to work as long as there is no agent has a majority of mining power, even if players know in advance the order in which they are scheduled.
3. To disincentivize deviation, Colordag penalizes forking: Considering the graph minors of each color separately, if there is more than one acceptable block of a given depth T in a minor, then all blocks of depth T get reward 0. Since each miner i aims to maximize its *relative* revenue (i.e., the ratio between i's revenue and the total reward received by miners while i is active, just as is the case in, e.g., [8, 19, 17, 11]), and (by assumption) deviators have less power than honest agents (i.e., agents that follow the prescribed protocol), a symmetric penalty to a deviator and an honest agent results in the deviator suffering more than the honest agents. Sliwinski and Wattenhofer [21] also use symmetric penalties in a blockdag for all blocks that are not connected by a directed path; each block in a set X of such blocks is penalized by $|X|c$ (for some constant c). However, with their approach, an adversary can harm honest agents. For example, if $c = 3$ and there is a benign honest fork, the attacker can add a third forked block, resulting in a total penalty of $6c$ for honest agents ($3c$ per block) while suffering only $3c$ itself, so deviation is worthwhile for a sufficiently large minority miner. In fact, their threshold is smaller than $1/2$, and their protocol is only an ε-NE, like Fruitchain.

The rest of the paper is organized as follows. In Section 2, we describe an abstract model of a PoW system, similar to models used in previous work, and discuss the bitcoin desiderata. In Section 3, we formalize mining as a game, so that we can make notions like incentive compatibility and best response precise. In Section 4, we formally describe the Colordag mechanism: the Colordag protocol and the revenue scheme that we use. We then prove in Section 5 that Colordag satisfies the ledger desiderata and is an ε-sure equilibrium in the face of coalitions with less than $1/2$ of the computational power, and even if the coalition knows what the scheduler does in advance. Specifically, we show that, for the appropriate choice of parameters, in all but a negligible fraction of histories, miners do not gain if they deviate from the Colordag protocol. Finally, in Section 6, we discuss the values of the Colordag parameters when dealing with a weaker adversary than we assume here and the path to a practical implementation.

2 Model and Desiderata

Blockchain protocols operate by propagating data structures called *blocks* over a reliable peer-to-peer network. We abstract this layer away and describe our model (see Section 2.1), which is similar to previous work. The goal of the protocol is to implement a distributed *ledger* (see Section 2.2), roughly speaking, a commonly-agreed upon record of transactions.

2.1 Model

The system proceeds in rounds in a synchronous fashion, as is common in many other analyses (e.g., [8, 10, 17, 18]). A *history* h is a complete description of what happens to the system over time. Formally, h is a function from rounds to a description of what has happened in the system up to round t (which blocks were generated, which were made public, which agents are in the system, and so on). We denote by $h(t)$ the prefix of h up to time t. There is a possibly unbounded number of agents, called *miners*, named $1, 2, \ldots$. We take the miners to represent coalitions of agents, so we do not talk about coalitions of miners (and will later assume that each miner controls less than $1/2$ of the computational power). For each history h and miner i, there exist rounds $T_1^{h,i}$ and $T_2^{h,i}$ such that i is *active* between $T_1^{h,i}$ and $T_2^{h,i}$.

Some previous analyses (e.g., [15, 8, 19, 5, 9]) focused on average rewards, and did not consider adversarial attacks that could lead to a violation of the ledger properties, although in an infinite execution such attacks may succeed with probability one. We aim to prove, with high probability, both that Colordag is incentive compatible (i.e., no agent can increase its utility by deviating from the protocol) and that, if all but at most one agent follow the protocol, then the ledger properties hold. So, like previous work (e.g., [17, 18, 13]), we assume that the system runs for a bounded time, up to some large $T_{\max}$. Without this assumption, even events with arbitrarily small frequency happen with probability one.

Let $Ag(h, t)$ be the set of active miners in the system at round t of history h, that is, all miners i such that $T_1^{h,i} \leq t \leq T_2^{h,i}$. For any given history and time, the set $Ag(h, t)$ is finite. Each miner i has so-called *mining power*, a positive value representing her computational power. The *power* of a miner i at time t, denoted $Pow_t^h(i)$, is her fraction of the mining power at time t in history h. Let $Pow^h(i) = \sup_t Pow_t^h(i)$, and let $Pow(i) = \sup_h Pow^h(i)$. We will be interested in the case that, for all miners i, there exists some $\alpha < 1/2$ such that $Pow(i) \leq \alpha$.

We assume that a scheduler determines which miners are active, which miners move in each round, and how long it takes a message to arrive. To simplify the discussion of the scheduler, we assume (as is the case for Colordag and all other blockchain algorithms) that each miner builds a local version of a directed acyclic graph called a *blockdag*. We refer to each node and its incoming edges in the graph as a *block*. Our hope is that miners have an "almost-common" view of the blockdag. Following the standard convention, we assume that the blockdag has a commonly-agreed-upon root that we refer to as the *genesis block*. The *depth* of a blockdag G, $d(G)$, is the length of a longest path in G. The *depth* of a block b in G, denoted $d(G, b)$, is the length of a longest path in G from the genesis to b.[2]

In every round, the scheduler chooses one miner at random among the miners that are active in that round (a miner i being chosen represents it having solved a computational puzzle), with probability proportional to its power (as in, e.g., [8, 19, 17]); that is, miner i is chosen in round t with probability proportional to $Pow_t(i)$. If the scheduler chooses a miner i in round t, then i either selects some set P of the nodes currently in its blockdag, with the constraint that no node in P can be the ancestor of another node in P, and adds a new vertex v to the blockdag with P as its parents or does nothing. If i adds (P, v), then i can either broadcast this fact or save it for possible later broadcast. Note that a miner cannot send (P, v) to a strict subset of miners; it is either broadcast to all miners or sent to none of

[2] We follow standard graph-theoretic terminology here. In the blockchain literature, what we are calling the depth of a node is sometimes called its height.

them (as in, e.g., [8, 10, 2] and deployed systems [15, 23]). Miners can also broadcast pairs that they saved earlier. If P violates the constraint that no node in P can be the ancestor of another node in P, the message (P, v) is ignored. We assume in the rest of the paper that this does not occur, as the outcome is indistinguishable from simply not generating a block.

Denote by $G^{h(t)}$ the blockdag including all blocks published at or before round t in execution h. Let $G_i^{h(t)}$ denote i's view of $G^{h(t)}$; this is the blockdag at round t of history h according to i. For example, i may not be aware at round t that j created block b, so block b will be in $G^{h(t)}$ but not in $G_i^{h(t)}$. Note that blocks that node i has generated but not published are not included in $G_i^{h(t)}$ (although, of course, i is aware of them); however, if a block $b \in G_i^{h(t)}$ refers to a block b' (i.e., b is a child of b', since we assume that the message broadcast by the miner that created block b has a hash of all the parents of b), then we take b' to have been published, and include it in $G_i^{h(t)}$. We omit the h if it is clear from context or if we are making a probabilistic statement; that is, if we say that a certain property of the graph holds at time t with probability p, then we mean that the set of histories h for which the property of $G^{h(t)}$ holds has probability p.

We assume that there is an upper bound $\Delta \geq 1$ on the number of rounds that it takes for a message to arrive. The arrival time of each message may be different for different miners; that is, if miner i broadcasts (P, v) at round t, miners j and j' might receive (P, v) in different rounds. Messages may also be reordered (subject to the bound on message delivery time).

Note that although there is a bound on message delivery time, miners do not know the publication time of a block. Thus, there is no way that a miner can tell if a block was withheld for a long period of time. Interestingly, in Colordag, agents can tell to some extent from the blockdag topology if a block was withheld for a long period of time; such blocks do not get any reward.

In summary, this is how the scheduler works: (1) it chooses, for each agent i, in which interval i is active and its power; (2) it chooses which agent generates a block in each round (randomly, in proportional to their power); and, finally, (3) it chooses a message-delivery function (i.e., a function that, given a history up to round m, decides how long it will take each round m message to be delivered, subject to the synchrony bound). We assume that the adversary knows the scheduler's choices.

The scheduler's protocol, including the choice of when agents are active and the random choice of which agents generate a block in each round, and the strategies used by the miners together determine a probability on the set of histories of the system. While we have specified that all messages must be delivered within Δ rounds, we have not specified a probability over message delivery times, block-generation times, or when agents are active. Our results hold whatever the probability is over message-delivery times (subject to it being at most Δ) and on when agents are active (subject to no agent having power greater than α). Thus, when we talk about a probability on histories, it is a probability determined by the strategies of the miners and a scheduler that satisfies the constraints above.

2.2 Desiderata

A ledger function $\mathcal{L}$ takes a blockdag G and returns a sequence $\mathcal{L}(G)$ of blocks in G; the kth element in the sequence is denoted $\mathcal{L}_k(G)$. The length of the ledger is denoted $|\mathcal{L}(G)|$. We want the ledgers that arise from the blockdags created by Colordag to satisfy certain properties [10, 17, 13].

The first property requires that once a block allocation is set, its position in the ledger remains the same in the view of all miners.

▶ **Definition 1** (Ledger Consistency). *There exists a constant K such that, for all miners i and j, if $k \leq |\mathcal{L}(G_i^{h(t)})| - K$ and $t \leq t'$, then $\mathcal{L}_k(G_i^{h(t)}) = \mathcal{L}_k(G_j^{h(t')})$.*

The next desideratum is that the length of the ledger should increase at a linear rate. Let $|\mathcal{L}(G)|$ denote the number of elements in the sequence $\mathcal{L}(G)$.

▶ **Definition 2** (Ledger Growth). *There exists a constant g such that, for all rounds $t < t'$ and all miners i, if $t' - t > g$, then $|\mathcal{L}(G_i^{h(t')})| \geq |\mathcal{L}(G_i^{h(t)})| + 1$.*

The final ledger desideratum says that the fraction of the total number of blocks in the ledger that are generated by honest miners should be larger than a positive constant.

▶ **Definition 3** (Ledger Quality). *There exist constants $D > 0$ and $\mu \in (0, 1)$ such that for all rounds t and t' such that $t' - t \geq D$, the fraction of blocks mined by honest miners placed on the ledger between round t and t' is at least μ.*

Note that this common requirement is fairly weak. As we will see, Colordag miners will be rewarded, on average, proportionally to their efforts. Indeed, to motivate miners to mine, the system rewards miners for essentially all the blocks they generate (not just the ones on the ledger). The revenue from each block is determined by the *revenue scheme*. Formally, a revenue scheme r is a function that associates with each block b and labeled blockdag G a nonnegative real number $r(G, b)$, which we think of as the revenue associated with block b in the blockdag G. Our final desideratum requires that revenue stabilizes.

▶ **Definition 4** (Revenue Consistency). *There exists a constant K such that, for all miners i and j and times t, t', and t'' such that $t', t'' > t + K$, if b is published at time t in history h, then $r(G_i^{h(t')}, b) = r(G_j^{h(t'')}, b)$.*

Most previous work (e.g., [15, 23, 18]) did not state this requirement explicitly. There, it follows from ledger consistency, since all and only blocks in the ledger get revenue.[3] In contrast, with Colordag, a miner might get revenue for a block even if it is not on the ledger, and may not get revenue for some blocks that are on the ledger. We thus need to separately require that the revenue that a miner gets from a block eventually stabilizes.

3 Revenue Scheme and ε-Sure NE

It is not hard to design protocols that satisfy the blockdag desiderata. However, there is no guarantee that the miners will actually use those protocols. We assume that miners are rational, so our goal is to have a protocol that is *incentive-compatible*: it is in the miners' best interests (appropriately understood) to follow the protocol. Before describing our protocol, we need to explain how the miners get utility in our setting.

3.1 Revenue Scheme

A miner's utility in a blockdag is determined by the miner's *revenue*. We denote by $B_i^{h(t)}$ the blocks generated by miner i in history $h(t)$. Given a revenue scheme r, for each miner i, history h, and round t, we can calculate the revenue $r(G_i^{h(t)}, b)$ for every block $b \in B_i^{h(t)}$.

[3] Ethereum's *uncle blocks* [23] are off-chain but rewarded; however, their rewards are explicitly placed in the ledger after a small number of blocks, therefore revenue consistency for Ethereum also follows almost trivially from ledger consistency.

Given a revenue scheme r, miner i's total revenue at round t according to r in history h of a protocol is the sum $\sum_{b \in B_i^{h(t)}} r(G_i^{h(t)}, b)$ of the revenue obtained for each block b generated by i while it is active in history h. For example, in Bitcoin [15], the revenue of a miner is the number of blocks it generated that are on the so-called main chain. Finally, i's *utility* according to revenue scheme r at round t in history h is i's normalized share of the total revenue while it is active. Taking $\mathit{time}(b)$ to be the time that block b was published, for $t \geq T_1^{h,i}$, we define:

$$u_i^r(h,t) = \frac{\sum_{b \in B_i^{h(t)}} r(G_i^{h(t)}, b)}{\sum_{\{b : T_1^{h,i} \leq \mathit{time}(b) \leq \min(t, T_2^{h,i})\}} r(G_i^{h(t)}, b)} . \tag{1}$$

This way of determining a miner's utility from a revenue function is common (see, e.g., [8, 19, 18, 11, 5, 4]). Intuitively, the utility is normalized because the value to a miner of holding a unit of currency depends on the total amount of currency that has been generated. A miner is interested in its utility during the time that it is active. Although miner i's utility may change over time, for a protocol that has the revenue consistency property (as Colordag does), in every history, i's utility eventually stabilizes (since the set of blocks that are published between $T_1^{h,i}$ and $T_2^{h,i}$ for which each miner gets revenue and the revenue that the miners get for these blocks eventually stabilize). When we talk about i's utility in history h, we mean the utility after all the revenue up to $T_2^{h,i}$ has stabilized.

3.2 ε-sure NE

As we said in the introduction, we are interested in strategy profiles that form a ε-sure Nash Equilibrium (NE), a strengthening of ε-NE as long as utility is bounded. We now define these notions carefully.

In the definition of ε-sure NE, we are interested in the probability that a history in a set H of histories occurs, denoted $\Pr(H)$. (Note that a history corresponds to a path in the game tree.) In general, the probability of a history depends on the strategies used by the miners. We are interested in sets of histories that have probability at least $(1-\varepsilon)$, independent of the strategies used by the miners. To ensure that this is the case, we take H to be a set of histories determined by the scheduler's behavior. The scheduler is a probabilistic algorithm. It chooses miners for block generation with probability $\mathit{Pow}_i(t)$, and chooses network propagation time arbitrarily, bounded by a constant Δ. The probabilities of the different histories are then defined by the probabilities of the scheduler's random coins. For example, suppose that there are 10 agents, all with the same computational power, and we consider histories where agent 1 is scheduled first, followed by agent 2. This set of histories has probability 1/100, independent of the agents' strategies.

We denote the strategy of each miner i by σ_i, a strategy profile by $\sigma = (\sigma_1, \ldots, \sigma_n)$, and the profile excluding the strategy of i by σ_{-i}. The profile with miner i's strategy replaced by σ'_i is (σ'_i, σ_{-i}).

▶ **Definition 5** (ε-sure NE). *A strategy profile $\sigma = (\sigma_1, \ldots, \sigma_n)$ is an ε-sure NE if, for each agent i, there exists a set H_i of histories with probability at least $1-\varepsilon$ such that, conditional on H_i, σ_i is a best response to σ_{-i}; that is, for all strategies $\sigma'_i \neq \sigma_i$ of agent i:*

$$u_i(\sigma \mid H_i) \geq u_i((\sigma'_i, \sigma_{-i}) \mid H_i).$$

Of course, if, for each agent i, we take H_i to consist of all histories; then we just get back NE, so all Nash equilibria are ε-sure NE for all ε. As the next result shows, if all utilities are in the interval $[m, M]$ then every ε-sure NE strategy profile is an $(M-m)\varepsilon$-NE. Since in our setting, the utility of a miner i is the fraction of total revenue that i obtains while i is active, the utility is in $[0, 1]$, so is clearly bounded.

▶ **Lemma 6.** *If a strategy profile σ is an ε-sure NE and all players' utilities are bounded in the range $[m, M]$, then σ is an $(M-m)\varepsilon$-Nash Equilibrium.*

Proof. For a player i, there is a set of histories H_i with probability $\Pr(H_i) > 1-\varepsilon$ where σ_i is a best response. In histories not in H_i, denoted $\overline{H}_i$, player i might improve her utility by up to $(M-m)$. The probability of $\overline{H_i}$ is bounded by ε. Therefore, the utility increase of a player by switching her strategy is at most $0(1-\varepsilon) + (M-m)\varepsilon = (M-m)\varepsilon$. Thus, σ is an $(M-m)\varepsilon$-NE. ◀

However, there are ε-NE that are not ε'-sure NE for any $\varepsilon' < 1$. For example, consider a game where a player chooses 0 or 1. She gets utility 0 for choosing 0 and utility ε for choosing 1. Choosing 0 is ε-NE but is not ε'-sure NE for any ε' as choosing 1 strictly increases her utility in all histories. Thus, ε-sure NE is a solution that lies strictly between ε-Nash and Nash equilibrium when utility is bounded, as it is in our case.

We will show that, for all ε, we can choose parameter settings to make Colordag an ε-sure NE. In addition, it satisfies the ledger desiderata.

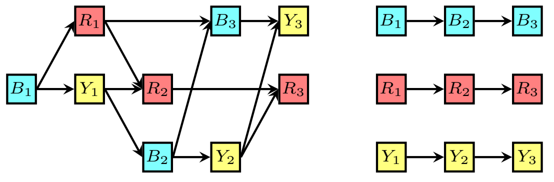
4 Colordag

The Colordag mechanism consists of a recommended strategy that we want participants to follow and a revenue scheme. The strategy, denoted σ^{cd} (*cd* stands for Colordag) is extremely simple: If chosen at round t in history h, miner i takes P to consist of the leaves of $G_i^{h(t)}$. It thus generates a block labeled b with parents P and broadcasts (P, b), adding it to its local view $G_i^{h(t)}$.

The reward function is more involved. Before describing it formally, we give some intuition for it. Suppose that we give all blocks reward 1. It is easy to see that σ^{cd} is a Nash equilibrium. But, with this reward function, so is every strategy profile where miners always publish the blocks they generate at some point. For example, miners can hang blocks off the genesis; this is also a best response. But if all miners choose to do this, it would be impossible to define a ledger that preserves consistency.

There is a simple fix to the second problem: if there is more than one block of the same depth, all blocks of that depth get reward 0. This stops hanging blocks off the genesis from being a best response. But now we have a new problem – we lose reward consistency. At any point, an adversary can penalize an arbitrary block b by adding a new block with the same depth as b. To obtain reward consistency, we would want to call the adversary's block in such cases *unacceptable*, and completely ignore it. Intuitively, we want blocks that hang off a block of depth T to be viewed as unacceptable if they are added after the blockdag has height sufficiently greater than T. This motivates our notion of unacceptability.

Roughly speaking, our reward function gives a reward of 1 to all blocks except those that are unacceptable or those that are forked; these get reward 0. The mechanism thus relies on a rational miner not being able to form a longer chain privately than the honest miners can form. (If a dishonest miner could form a longer chain privately than the honest miners can form, it could then publish that chain and make all the blocks that the honest miners formed during that time unacceptable.) However, forks can happen naturally, due to network



(a) A colored dag. **(b)** Graph minors.

Figure 1 Coloring a dag.

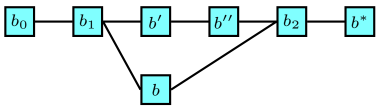


Figure 2 An unacceptable block.

latency, meaning honest miners' chain-extension rate is less than their block-generation rate, whereas the rational miner's rate is unimpaired. To mitigate the effect of forking, we color the nodes, effectively partitioning the blockdag into disjoint *graph minors* [6] (one minor for each color); we determine forking (and acceptability) in these graph minors. We can make the amount of forking as small as we want by using enough colors. We now present the key components needed for the reward function, and then give the actual function.

Coloring nodes

Because messages may take up to Δ rounds to arrive, two honest miners can both extend a given block b, because neither has heard of the other's extension at the point when it is doing its own extension. To make our results as strong as possible, following the literature [10, 13, 21], we assume that a deviating miner is able to avoid forking with its own blocks. Thus, a deviator can extend paths in the blockdag faster than would be indicated by her relative power. In particular, a deviator with power less than (but close to) 1/2 may be able to (with high probability) build paths longer than the honest miners can build, due to forking.

To deal with this problem, Colordag assigns each block a color chosen at random from a sufficiently large set of N_C colors; that is, it assigns each block a number in $\{1, \ldots, N_C\}$ (which we view as a color). In practice, this would be done by taking the color to be the hash of the contents of the block mod N_C. This ensures that, except with negligible probability (1) all colors are equally likely, (2) the color of a block b is learned by the miner that generates b only after b is generated, and (3) colors are commonly known (every miner can compute the color of every block, just knowing its content). In our model, this is like having the scheduler allocate a random color when it chooses a miner in a round. Figure 1a shows a blockdag where the nodes are colored either blue (B), red (R), or yellow (Y).

After coloring each node in the graph G, we consider the graph minor G_c corresponding to color c: The nodes in this graph minor are just the nodes of color c in G; node b' is a child of b in G_c iff b' is a descendant of b in G and there is no path in G from b to b' with an intermediate node (i.e., one strictly between b and b') of color c. Figure 1b shows the minors resulting from our example.

The key point is that, by taking N_C sufficiently large, we make the probability of a fork among the blocks generated by honest miners in G_c arbitrarily small. The reasoning is simple: Suppose that b and b' are generated by honest miners at times t_b and $t_{b'}$, respectively, where $t_{b'} > t_b$. If b and b' have the same color and there are enough colors, then with high probability, $t_{b'} > t_b + \Delta$, so b' is a descendant of b in G, and hence also in G_c. In other words, if two honest blocks are neither an ancestor nor a descendant of one another in G, they are unlikely to have the same color.

Acceptable blocks

We now define what it means for a block to be acceptable. We want it to be the case that a block is unacceptable if it has depth T but was added after the depth of the blockdag is considerably greater than T. The way we capture this is by requiring acceptable blocks to be on paths that are almost the same as a particular longest path in the graph.

Given a dag G_c, we "close off" G_c so that it has a unique initial node and a unique final node (whether or not it already had them), by adding special vertices b^0 and b^*, where b^0 is the parent of all the roots of G_c (essentially we consider b^0 to be the genesis, belonging to all minors) and b^* is the child of all leaves in G_c. We refer to this graph as G_c^+. We denote by $|Q|$ the length of a path Q, which is the number of edges in Q, and hence one less than the number of vertices in Q.

Given a graph G, for each color c, we choose one particular longest path in G_c^+ from b^0 to b^*. If there is more than one longest path, we use a canonical tie-breaking rule, which we now define, as it will be useful later. Intuitively, if there are several paths of maximal length, we order the paths by considering the point where they first differ, and choose using some fixed tie-breaking rule that depends only on the contents of the blocks where they first differ.

▶ **Definition 7** (Canonical path). *Given a blockdag, the canonical path starts at the genesis and continues as all longest paths do up to the first point where some longest paths diverge (this could already happen at the genesis). At this point, we choose some tie-breaking rule to decide which longest paths to follow.*[4] *The canonical path continues as all these longest paths until the next point of divergence. Again, at this point we use the tie-breaking rule to decide which longest paths to follow. We apply this procedure each time longest paths diverge.*

The key point is that all these tie-breaking rules are local. The decisions made are the same (if all the prefixes of these paths exist) in all the graphs we consider.

▶ **Definition 8** (Acceptable Block). *A path P in G_c^+ from block b^0 to block b^* is N_ℓ-*almost-optimal *if the symmetric difference between P and the canonical longest path P^* (i.e., the set of blocks in exactly one of the paths P and P^*) has fewer than N_ℓ blocks. A block b of color c is N_ℓ-*acceptable *iff it is on an N_ℓ-almost-optimal path P of color c. The path P is said to be a* witness *to the acceptability of b.*

We need one more definition before we can define the revenue scheme.

▶ **Definition 9** (Forked Block). *An N_ℓ-acceptable block b in blockdag G is N_ℓ-*forked *if there is another N_ℓ-acceptable block b' with the same color as b, say c, such that $d(G_c, b) = d(G_c, b')$.*

We can now make Colordag's revenue scheme precise. As we said, a block of color c gets reward 1 unless it is unacceptable or it is forked in G_c. The revenue scheme takes N_ℓ as a parameter, so we denote it $r^{cd}_{N_\ell}$.

▶ **Definition 10** (Colordag Revenue Scheme). *A node b is N_ℓ-*compensated *if b is N_ℓ-acceptable in G_c and is not N_ℓ-forked; $r^{cd}_{N_\ell}(G, b) = 1$ if b is N_ℓ-compensated; otherwise, $r^{cd}_{N_\ell}(G, b) = 0$.*

[4] For example, in practice this could be the smallest hash of the block contents.

Colordag Ledger Function

We present here a ledger function that makes the analysis easier, and satisfies all the ledger properties. This function is somewhat inefficient, since not all blocks are a part of the ledger. In Section 6, we show how a small modification of this approach lets us include in the ledger the transactions that appear in all acceptable blocks in the blockdag.

The ledger function of Colordag chooses a fixed color $\hat{c}$, and given graph G, chooses the canonical path in the subgraph of G of color $\hat{c}$. The ledger is defined by the blocks on this path. For example, given the blockdag in Figure 1a, and assuming $\hat{c}$ is yellow, the ledger is the sequence of blocks (Y_1, Y_2, Y_3).

▶ **Definition 11** (Colordag Ledger Function)**.** *Given a blockdag G and a fixed color $\hat{c}$, Colordag's ledger function $\mathcal{L}^{cd}$ returns a sequence consisting of the blocks on the canonical path in $G_{\hat{c}}$.*

Reward Calculation

Since following the protocol is the miners' best response, in practice they will generate a single chain of each color and get rewarded per block. As we now show, the reward calculation can be done in polynomial time, even if miners deviate. Given N_ℓ, a graph G, and a block b of color c, we want to calculate $r^{cd}_{N_\ell}(G, b)$. The first task is to construct the graph minor G_c of color c; this clearly can be done in time polynomial in $|G|$. The next step is to determine the canonical longest path P^* in G_c. We can do this quickly, since it is well known that longest paths in dags can be calculated in linear time [20]. (Indeed, it is straightforward to keep a table of lengths of longest paths and update it as G_c grows over time.) Finally, using depth-first search, we can quickly compute the block b_2 of least depth on P^* that is a descendant of b (which is b itself if b is on P^*) and the block of greatest depth b_1 on P^* that is an ancestor of b. By construction there is a path from b_1 to b_2 that includes b. It is easy to see that b is acceptable iff the number of nodes on the path fromn b_1 to b_2 that includes b (not including b_1 and b_2) and the number of nodes on the canonical path from b_1 to b_2 (again, not including b_1 and b_2) is less than N_ℓ. If b is forked, then similar arguments allow us to check whether a block forking b is acceptable. If b is acceptable and no block forking b is acceptable, then $r^{cd}_{N_\ell}(G, b) = 1$; otherwise, $r^{cd}_{N_\ell}(G, b) = 0$.

5 Analysis

In this section, we show that Colordag satisfies all the blockdag desiderata and is an ε-sure NE (and thus also an ε-NE). Note that it follows directly from the utility definition (Equation 1) that if all agents follow the Colordag protocol, the expected utility of each miner is its relative power. We do the analysis under the assumption that we have a very strong adversary, one who knows the scheduler's protocol. This means that the adversary knows when agents will join and leave the system, when agents will generate blocks, and when messages will arrive. To get this strong guarantee, we may need the parameters N_C and N_ℓ to be large (in general, the choice of N_C and N_ℓ depend on $T_{\max}$). We believe that in practice much smaller parameters will suffice. We return briefly to this issue in the conclusion.

The first step in doing this is to identify a set of "reasonable" histories that has probability at least $1 - \varepsilon$. One of the things that makes a history reasonable is that there is little forking. The whole point of coloring is that we can make the probability of forking arbitrarily small in the graphs of color c, by choosing enough colors.

▶ **Definition 12.** *A pair (b_1, b_2) of blocks is a* natural c-fork *in a history h if b_1 and b_2 both have color c, they are both generated within a window of Δ rounds, and neither is an ancestor of the other in G^h. An interval $[t_1, t_2]$* suffers at most δ-c-forking loss *if, the set of blocks b_1 generated in $[t_1, t_2|$ for which there exists a block b_2 such that (b_1, b_2) is a natural c-fork is a fraction less than δ of the total number of blocks of color c generated in $[t_1, t_2]$.*

We now consider histories that satisfy three properties that will turn out to be key to our arguments.

▶ **Definition 13** (Safe history). *A history is $(N_C, N_\ell, \delta, \delta_C, T_{\max})$-*safe *if, for all miners i, and all colors c,*

SH1. *for every subinterval $[t'_1, t'_2]$ of $[0, T_{max}]$, such that at least N_ℓ blocks of color c are generated in the interval $[t'_1, t'_2]$, miner i generates less than $1/2 - \delta$ of them;*

SH2. *every subinterval $[t'_1, t'_2]$ of $[0, T_{max}]$ such that $t'_2 - t'_1 \geq N_\ell$ suffers at most δ-c-forking loss; and*

SH3. *for every subinterval $[t'_1, t'_2]$ of $[0, T_{max}]$ such that $t'_2 - t'_1 \geq N_\ell$, there are at least $\delta_C(t'_2 - t'_1)$ blocks of color c generated in $[t'_1, t'_2]$.*

Let $H^{N_C, N_\ell, \delta, \delta_C, T_{max}}$ denote the set of histories that are $(N_C, N_\ell, \delta, \delta_C, T_{max})$-safe.

▶ **Proposition 14.** *Suppose that for all miners i, $Pow(i) \leq \alpha < 1/2$. Then for all $\varepsilon > 0$, there exists a positive integer T^*_{max} such that for all $T_{max} \geq T^*_{max}$, there exist N_C, $N_\ell < T_{max}$, $\delta \in (0, 1/2)$, and $\delta_C \in (0, 1)$ such that* $\Pr(H^{N_C, N_\ell, \delta, \delta_C, T_{max}}) \geq 1 - \varepsilon$.

To prove the proposition, we use Hoeffding's inequality to find conditions on the parameters on N_C, N_ℓ, δ, and δ_C for the conditions SH1-SH3 to hold given α and $T_{\max}$ with probability $1 - \varepsilon/3$. If all conditions are satisfied, then SH1-SH3 hold with probability at least $1 - \varepsilon$. Finally, we show that such conditions can be found for all sufficiently large $T_{\max}$ values. The proof is deferred to Appendix A.

We say that $(N_C, N_\ell, \delta, \delta_C, T_{\max})$ is *suitable* for ε and α if $\Pr(H^{N_C, N_\ell, \delta, \delta_C, T_{\max}}) \geq 1 - \varepsilon$. We show that $(N_C, N_\ell, \delta, \delta_C, T_{\max})$-safe histories are "good" (in systems where $(N_C, N_\ell, \delta, \delta_C, T_{\max})$ is suitable for the desired ε, and $\alpha < 1/2$). The following propositions show that good things happen in $H^{N_C, N_\ell, \delta, \delta_C, T_{\max}}$. The first one shows that all of blocks generated by honest miners are acceptable.

▶ **Proposition 15.** *For all histories $h \in H^{N_C, N_\ell, \delta, \delta_C, T_{max}}$ and all colors c, there exists a path P from b^0 to b^* in $G_c^{h(t)}$ that contains all blocks of honest miners of color c that are not naturally c-forked. Moreover, every block on P is acceptable.*

Proof. Fix a color c. If b and b' are blocks of honest miners in $G_c^{h(t)}$ that are not naturally forked, then either b is an ancestor of b' or b' is an ancestor of b in $G_c^{h(t)}$. Thus, there is a path P from b^0 to b^* that contains all the blocks of honest miners that are not naturally c-forked (see Figure 3).

Now consider any block b on P. If b is on the canonical longest path P^*, then it is acceptable by definition. Suppose that b is not on P^*. Let b_1 be the last node on P preceding b that is on P^*, and let b_2 be the first node on P following b that is on P^*. Let Q (resp., Q^*) be the subpath of P (resp., P^*) from b_1 to b_2. If the total number of nodes on Q and Q^*, not counting b_1 and b_2, is less that N_ℓ, then the path P' that is identical to P^* up to b_1, continues from b_1 to b_2 along P, and then continues along P^* again, is an N_ℓ-almost optimal path that contains b, showing that b is acceptable.

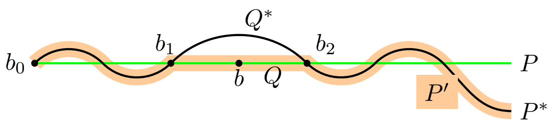


Figure 3 Honest (and hence acceptable) blocks on the path containing all non-forked honest.

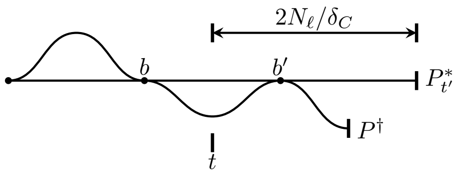


Figure 4 The situation if b' is the only honest block generated after b.

It thus suffices to show that there cannot be more than N_ℓ nodes on Q and Q^*, not counting b_1 and b_2. Suppose, by way of contradiction, that there are. Further suppose that b_1 is generated at time t_1 and b_2 in generated at time t_2. That means that all the blocks on Q and Q^* other than b_1 and b_2 are generated in the interval $[t_1+1, t_2-1]$. Thus, at least N_ℓ blocks are generated in this interval. Since P^* is a longest path, Q^* must be at least as long as Q (otherwise going from b_1 to b_2 along Q would give a longer path). But by Proposition 14, at least a fraction $1/2+\delta$ in the interval $[t_1+1, t_2-1]$ are generated by honest miners. Since there is at most δ-c forking loss, it follows that the majority of the c-colored blocks in this interval are generated by honest miners and are not naturally forked. These blocks must all be on Q. Thus, Q must have a majority of the blocks in this interval, giving us the desired contradiction. ◀

We are now ready to prove that $\mathcal{L}^{\text{cd}}$ satisfies the ledger desiderata (in safe histories) with the Colordag protocol. Note that since we view a miner as representing a coalition of agents, the fact that all but at most one miner is honest means that we allow a coalition with power up to $\alpha < 1/2$ to deviate. The proofs are deferred to Appendix B.

▶ **Proposition 16** (Colordag ledger consistency). *If $(N_C, N_\ell, \delta, \delta_C, T_{max})$ is suitable for ε and $\alpha < 1/2$ then for all miners i, j and all histories $h \in H^{N_C,N_\ell,\delta,\delta_C,T_{max}}$, if all but at most one miner is honest in h, $t \le t'$, and $k \le |\mathcal{L}(G_i^{h(t)})| - N_\ell$, then $\mathcal{L}_k(G_i^{h(t)}) = \mathcal{L}_k(G_j^{h(t')})$.*

▶ **Proposition 17** (Colordag ledger growth). *If $(N_C, N_\ell, \delta, \delta_C, T_{max})$ is suitable for ε and $\alpha < 1/2$, then for all rounds t and t' such that $t' - t \ge N_\ell/\delta_C$, if all but at most one miner is honest in $h \in H_i^{N_C,N_\ell,\delta,\delta_C,T_{max}}$, then $|\mathcal{L}^{cd}(G_i^{h(t')})| \ge |\mathcal{L}^{cd}(G_i^{h(t)})| + 1$.*

▶ **Proposition 18** (Colordag ledger quality). *If $(N_C, N_\ell, \delta, \delta_C, T_{max})$ is suitable for ε and $\alpha < 1/2$ then for all rounds t and t' such that $t' - t \ge 2N_\ell/\delta_C$, and all $h \in H_i^{N_C,N_\ell,\delta,\delta_C,T_{max}}$, at least two of the blocks of color $\hat{c}$ added to $\mathcal{L}(G_i^{h(t')})$ in the interval $[t, t']$ are generated by honest miners.*

▶ Note 19. In Propositions 17 and 18, we explicitly assume that we are given an acceptable tuple. Of course, if N_ℓ and δ_C in the tuple are such that $N_\ell/\delta_C > T_{\max}$, then the propositions are essentially vacuous, since there are no times $t, t' < T_{\max}$ such that $t' - t > N_\ell/\delta_C$. Put another way, although it is true that if the system runs for at least N_ℓ/δ_C steps then the ledger is guaranteed to increase in length by 1, given that the system runs for only $T_{\max}$ steps, this is not terribly interesting if $N_\ell/\delta_C > T_{\max}$. Similar comments apply to Proposition 18. The good news is that even for stringent choices of ε and α, there exist suitable tuples that make Propositions 17 and 18 non-vacuous. For example, if $\alpha = .49$ and $\varepsilon = 10^{-7}$, and we assume that $\Delta = 5$, then we can take $T_{\max} = 10^{11}$, $N_\ell = 10^4$, $N_C = 10$, $\delta = .005$, and $\delta_C = 0.04$, to get a suitable tuple, even with the crude analysis in the proof of Proposition 14. In this case, $N_\ell/\delta = 2 \times 10^6$, which is much less than $T_{\max} = 10^{11}$. A more careful analysis should give better numbers, but these suffice to make the point. (As we hinted earlier, with a more realistic adversary, who does not have perfect knowledge of the future, we would also expect far better numbers.) We also note that although Fruitchain does not seem to have an explicit bound $T_{\max}$ on how long the system runs, that bound does arise from the polynomial bound of the p.p.t. environment Z ([18] Section 2.1, *Constraints on (A, Z)*).

The next proposition essentially shows that Colordag is an ε-sure NE.

▶ **Proposition 20.** *If $(N_C, N_\ell, \delta, \delta_C, T_{max})$ is suitable for ε, $\alpha < 1/2$, $h \in H^{N_C,N_\ell,\delta,\delta_C,T_{max}}$, and $t_2^i - t_1^i > N_\ell$, then i does not benefit by deviating if all other miners are honest, given revenue scheme $r_{cd}^{N_\ell}$.*

Proof. By Proposition 15, all honest blocks are acceptable in h, no matter what i does. Obviously i can make her own blocks unacceptable, but this would only affect her own revenue and decrease her utility.

It remains to show that i decreases her utility by creating forks. Suppose that M blocks generated in h in the interval $[t_1^i, t_2^i]$ by miners other than i and M' blocks are generated by i. We must have $M > M'$ (SH1). If i does not deviate, then all these blocks are compensated, so i's utility is $\frac{M'}{M+M'}$. If i deviates, i can decrease the utility of the other miners only by forking blocks (since there is nothing that i can do to make a block unacceptable, as we mentioned above). It is easy to see that every block of the other miners that is forked by i comes at a cost of i forking one of his own blocks. Thus, if i deviates so as to fork M'' blocks, then i's utility is $\frac{M'-M''}{M+M'-2M''}$. Since $M'' \le M' < M$, simple algebra shows that $\frac{M'}{M+M'} > \frac{M'-M''}{M+M'-2M''}$, so this deviation results in the deviator losing utility.

Note that since we assume the deviator knows the history, it can deterministically deviate without affecting the blockdag structure. Hence the equilibrium is not strict. ◀

▶ **Corollary 21.** *If $(N_C, N_\ell, \delta, \delta_C, T_{max})$ is suitable for ε and $\alpha < 1/2$, then Colordag with this choice of parameters is an ε-sure NE.*

Proof. This is immediate from Proposition 20, since if $(N_C, N_\ell, \delta, \delta_C, T_{\max})$ is suitable for ε and $\alpha < 1/2$, then $\Pr(H^{N_C,N_\ell,\delta,\delta_C,T_{\max}}) \ge 1 - \varepsilon$. ◀

Finally, we prove that the Colordag revenue scheme satisfies revenue consistency. We begin by showing that once a block is deep enough, its revenue is set and does not change.

▶ **Lemma 22.** *If $(N_C, N_\ell, \delta, \delta_C, T_{max})$ is suitable for ε and $\alpha < 1/2$, then for all miners i, j, all histories $h \in H_i^{N_C,N_\ell,\delta,\delta_C,T_{max}}$, all blocks b, and all colors c, if $d(G_{i,c}^{h(t)}, b) \le d(G_{i,c}^{h(t)}) - 2N_\ell$ and $t \le t'$, then $r_{N_\ell}^{cd}(G_i^{h(t)}, b) = r_{N_\ell}^{cd}(G_j^{h(t')}, b)$.*

Proof. As in the proof of Proposition 16, let $P_{t'}^*$ be the canonical longest path in $G_{j,c}^{h(t')}$, let P_t be its prefix in $G_{i,c}^{h(t)}$, let P_t^* be the canonical longest path in $G_{i,c}^{h(t)}$, and let b' be the last common block on P_t^* and P_t. As in the proof of Proposition 16, P_t^* and P_t are identical up to b', and we can derive a contradiction if $d(G_{i,c}^{h(t)}, b') \le d(G_{i,c}^{h(t)}) - N_\ell$, so

$$d(G_{i,c}^{h(t)}, b') > d(G_{i,c}^{h(t)}) - N_\ell. \tag{2}$$

Suppose that b is acceptable in $G_i^{h(t)}$. That means that it is on some N_ℓ-almost optimal path P in $G_{i,c}^{h(t)}$. Let b_1 be the first block on P_t^* that is an ancestor of b, and let b_2 be the first block on P_t^* that is a descendant of b. Perhaps $b_1 = b'$ and perhaps $b_2 = b^*$ (the final block added at the end of the graph). Let Q be the subpath of P from b_1 to b_2, and let Q' be the subpath of P_t^* from b_1 to b_2. Since P is N_ℓ-almost optimal in $G_i^{h(t)}$, it must be the case that $|Q| + |Q'| - 2 < N_\ell$. Since the depth of b is at least N_ℓ less than that of b' (from the proposition statement and from Equation 2), it follows that b_2 must precede b'. Since P_t^* and P_t agree up to b', this argument also shows that $P_{t'}^*$ with Q instead of Q' between b_1 and b_2 is N_ℓ-almost optimal in $G_{j,k}^{h(t')}$, hence that b is acceptable in $G_{j,k}^{h(t')}$. Just changing the roles of $G_i^{h(t)}$ and $G_j^{h(t')}$, this argument shows that if b is acceptable in $G_j^{h(t')}$, then it is also acceptable in $G_i^{h(t)}$.

It is now almost immediate that b is not forked by an acceptable block in $G_i^{h(t)}$ iff it is not forked by an acceptable block in $G_j^{h(t')}$.

In conclusion, block b is acceptable and not forked by an acceptable block in $G_i^{h(t)}$ iff it is acceptable and not forked by an acceptable block in $G_j^{h(t')}$. That is, by the definition of $r_{N_\ell}^{cd}$, it is compensated in $G_i^{h(t)}$ iff it is compensated in $G_j^{h(t')}$. ◀

The next proposition shows that Colordag satisfies revenue consistency.

▶ **Proposition 23** (Colordag Revenue Consistency). *If $(N_C, N_\ell, \delta, \delta_C, T_{max})$ is suitable for ε and $\alpha < 1/2$, then for all miners i and j and times t, t', and t'' such that $t', t'' > t + 4N_\ell N_C/(\delta_C(1-\delta))$, if b is published at time t in history $h \in H_i^{N_C,N_\ell,\delta,\delta_C,T_{max}}$, then $r(G_i^{h(t')}, b) = r(G_j^{h(t'')}, b)$.*

Proof. Suppose that block b is published at time t and has color c. By SH3, within $2N_\ell N_C/(\delta_C(1-\delta))$ rounds, at least $2N_\ell N_C/(1-\delta)$ blocks of color c are generated. By SH1, at least $N_\ell N_C/(1-\delta)$ are honest. By SH2, a fraction $(1-\delta)$ of these are not forked. This means at least $N_\ell N_C$ blocks are not forked, so the depth of G_c has increased by at least $N_\ell N_C$ after $2N_\ell N_C/(\delta_C(1-\delta))$ rounds. Now, for any pair of times $t', t'' > t + 4N_\ell N_C/(\delta_C \delta)$, the depth of the graph is larger by at least $2N_\ell$ than b's depth, therefore, by Lemma 22, the reward for b is the same in both $G_i^{h(t')}$ and $G_j^{h(t'')}$. ◀

6 Conclusion

We present Colordag, a protocol that incentivizes correct behavior of PoW blockchain miners up to 50%, and is an ε-sure equilibrium. That is, unlike previous solutions, the desired behavior is a best response in all but a set of histories of negligible probability. As long as a majority of the participants follow the behavior prescribed by Colordag, the ledger desiderata, as well as reward consistency, all hold.

We prove the properties of Colordag when playing against an extremely strong adversary, one that knows before deviating when agents will generate blocks and when messages will arrive. Intuitively, to benefit from a deviation, a deviator must produce an acceptable path longer than N_ℓ and longer than the honest path. Knowing in advance what order messages can arrive in and whether there is forking means that a deviator knows in advance whether the deviation can succeed. Our analysis shows that, even with this knowledge, a deviation can succeed with only low probability. Unfortunately, to get such a strong guarantee, we may need the parameters N_C and N_ℓ to be quite large Moreover, our ledger is quite inefficient, in that it does not include transactions in blocks that are not on the canonical path in $G_{\hat{c}}$. In practice, we believe that both problems can be dealt with.

We start with the second problem. To improve throughput, we can use ideas that have also appeared in previous work (e.g., [14, 2]): Suppose that b and b' are consecutive blocks on the ledger (which thus both have color $\hat{c}$). When we add b' to the ledger, we also add to the ledger not just the transactions in b', but all the transactions of the acceptable predecessors of b' (of all colors) that were not already included in the ledger. These additional transactions are ordered by the depth of the block they appear in, using color as a tiebreaker, and hash as a second tiebreaker. For example, given the blockdag of Figure 1b, if $\hat{c}$ is blue, then the ledger function includes the transactions in the blocks B_1, Y_1, B_2, R_1, B_3 (in that order); if $\hat{c}$ is red, the ledger includes the transactions in the blocks $B_1, R_1, Y_1, R_2, B_2, Y_2, R_3$ (in that order). It is not hard to check that, with this approach, all transactions in honest blocks of honest agents will be included in the ledger, so our throughput is quite high.

We next consider the fact that we require N_C and N_ℓ to be quite large. This is due to our assumption that a deviator knows what order messages can arrive in and whether there is forking. In practice, a potential deviator will not have this information. For such a weaker adversary, the parameters can be significantly smaller than those required to obtain the bounds presented here. Without this a priori knowledge, the probability that a deviation succeeds drops quickly with N_ℓ. Therefore, the cost of failed attempts grows with N_ℓ, while their overall benefit drops. An analysis of this kind can be done using deep reinforcement learning, which is helpful when the state and action spaces are too rich for an exact solution [11, 3, 4]. This is beyond the scope of this paper, but preliminary experiments suggest that under practical assumptions, with this more limited adversary, Colordag can perform well with reasonable parameter choices. We hope to report on this work in the future.

References

1 Adam Back. Hashcash – a denial of service counter-measure. `http://www.cypherspace.org/hashcash/hashcash.pdf`, 2002.

2 Vivek Bagaria, Sreeram Kannan, David Tse, Giulia Fanti, and Pramod Viswanath. Prism: Deconstructing the blockchain to approach physical limits. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 585–602, 2019.

3 Roi Bar-Zur, Ameer Abu-Hanna, Ittay Eyal, and Aviv Tamar. Werlman: To tackle whale (transactions), go deep (RL). In *IEEE Symposium on Security and Privacy (SP)*, 2022.

4 Roi Bar-Zur, Danielle Dori, Sharon Vardi, Ittay Eyal, and Aviv Tamar. Deep bribe: Predicting the rise of bribery in blockchain mining with deep RL. In *6th workshop on Deep Learning Security and Privacy (DLSP)*, 2023.

5 Roi Bar Zur, Ittay Eyal, and Aviv Tamar. Efficient MDP analysis for selfish-mining in blockchains. In *2nd ACM Conference on Advances in Financial Technologies (AFT)*, 2020.

6 Reinhard Diestel. *Graph Theory.* Springer Graduate Texts in Mathematics. Springer-Verlag, 5th edition, 2017.

7 Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *Proceedings CRYPTO '92: 12th International Cryptology Conference*, pages 139–147. Springer, 1992.

8 Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *Financial Cryptography and Data Security*, 2014.

9 Matheus V. X. Ferreira and S. Matthew Weinberg. Proof-of-stake mining games with perfect randomness. In *Proceedings of the 22nd ACM Conference on Economics and Computation*, pages 433–453, 2021.

10 Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The Bitcoin backbone protocol: Analysis and applications. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 281–310, 2015. `doi:10.1007/978-3-662-46803-6_10`.

11 Charlie Hou, Mingxun Zhou, Yan Ji, Phil Daian, Florian Tramer, Giulia Fanti, and Ari Juels. Squirrl: Automating attack discovery on blockchain incentive mechanisms with deep reinforcement learning. *arXiv:1912.01798*, 2019.
12 Markus Jakobsson and Ari Juels. Proofs of work and bread pudding protocols. In *Secure Information Networks*, pages 258–272. Springer, 1999.
13 Lucianna Kiffer, Rajmohan Rajaraman, and Abhi Shelat. A better method to analyze blockchain consistency. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 729–744, 2018.
14 Yoad Lewenberg, Yonatan Sompolinsky, and Aviv Zohar. Inclusive block chain protocols. In *Financial Cryptography*, Puerto Rico, 2015.
15 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `http://www.bitcoin.org/bitcoin.pdf`, 2008.
16 Kartik Nayak, Srijan Kumar, Andrew Miller, and Elaine Shi. Stubborn mining: Generalizing selfish mining and combining with an eclipse attack. *IACR Cryptology ePrint Archive*, 2015:796, 2015. URL: `http://eprint.iacr.org/2015/796`.
17 Rafael Pass, Lior Seeman, and Abhi Shelat. Analysis of the blockchain protocol in asynchronous networks. Technical report, Cryptology ePrint Archive, Report 2016/454, 2016.
18 Rafael Pass and Elaine Shi. Fruitchains: A fair blockchain. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 315–324, 2017.
19 Ayelet Sapirshtein, Yonatan Sompolinsky, and Aviv Zohar. Optimal selfish mining strategies in Bitcoin. In *Financial Cryptography and Data Security*, 2016.
20 R. Sedgewick and K. Wayne. *Algorithms.* Addison-Wesley, fourth edition, 2011.
21 Jakub Sliwinski and Roger Wattenhofer. Blockchains cannot rely on honesty. `https://disco.ethz.ch/courses/distsys/lnotes/rationalblockchainpaper.pdf`, 2019.
22 Yonatan Sompolinsky, Shai Wyborski, and Aviv Zohar. Phantom ghostdag: a scalable generalization of nakamoto consensus. In *Proceedings of the 3rd ACM Conference on Advances in Financial Technologies*, pages 57–70, 2021.
23 Gavin Wood. Ethereum yellow paper. `https://web.archive.org/web/20160820211734/http://gavwood.com/Paper.pdf`, 2015.
24 H. Yu, Nikolić I., R. Hou, and P. Saxena. Ohie: Blockchain scaling made simple. In *2020 IEEE Symposium on Security and Privacy (SOSP)*, 2020.

A The Probability of a Safe History

We prove that a safe history has overwhelming probability.

▶ **Proposition 14.** *Suppose that for all miners i, $Pow(i) \leq \alpha < 1/2$. Then for all $\varepsilon > 0$, there exists a positive integer T^*_{max} such that for all $T_{max} \geq T^*_{max}$, there exist N_C, $N_\ell < T_{max}$, $\delta \in (0, 1/2)$, and $\delta_C \in (0, 1)$ such that $\Pr(H^{N_C,N_\ell,\delta,\delta_C,T_{max}}) \geq 1 - \varepsilon$.*

Proof. We show that there exist constraints on $T_{\max}$, N_C, N_ℓ, and δ_C such that, if the constraints are satisfied, then the probability for the set of histories that have property SH1 (resp., SH2; SH3) is at least $1 - \varepsilon/3$. We then show that these constraints are satisfiable. The result then follows from the union bound.

We start with SH2. Fix a color c, and suppose that there are N_C colors. The probability that a block b has color c is $1/N_C$. To simplify notation in the rest of this proof, we take $\gamma = 1/N_C$. For b to be the earlier of two blocks that are naturally c-forked, there must be another block of color c that is generated within an interval of less than Δ after b is generated. Suppose that b is generated in round r. The probability that a block b generated in round r has color c is γ. The probability that none of the blocks generated in rounds $r + 1, \ldots, r + \Delta - 1$ has color c is $(1 - \gamma)^{\Delta - 1}$, so the probability b is not naturally c-forked is at least $(1 - \gamma)^{\Delta - 1}$.

Fix an interval $[t_1', t_2']$. The probability that that $[t_1', t_2']$ suffers greater than δ-c-forking loss is exactly the probability that there are fewer than $(1-\delta)(t_2'-t_1')$ blocks of some color c that are naturally forked by a later block. For a fixed color c, by Hoeffding's inequality, this probability is at most $e^{-2(t_2'-t_1')[(t_2'-t_1')((1-\gamma)^{\Delta-1}-\delta)]^2}$. Since we are interested only in the case that $t_2'-t_1' \geq N_\ell$, there are N_C colors, $\gamma = 1/N_C$ and there are at most $\binom{T_{\max}}{2} \leq T_{\max}^2$ possible choices of t_1' and t_2', SH2 holds with probaiblity at least $1-\varepsilon/3$ if

$$N_C T_{\max}^2 e^{-2N_\ell^3((\frac{N_C-1}{N_C})^{\Delta-1}-\delta)^2} < \varepsilon/3. \tag{3}$$

Equation (3) is thus the constraint that needs to be satisfied for SH2.

For SH3, again, fix a color c, and suppose that there are N_C colors. Then the expected number of blocks of color c in an interval $[t_1', t_2']$ is $\gamma(t_2'-t_1')$, so by Hoeffding's inequality, the probability of there being fewer than $\delta_C(t_2'-t_1')$ blocks of color c in the interval $[t_1', t_2']$ is at most $e^{-2(t_2'-t_1')[(t_2'-t_1')(\gamma-\delta_C)]^2}$. Much as in the argument for SH2, it follows that SH3 holds with probability at least $1-\varepsilon/3$ if

$$N_C T_{\max}^2 e^{-2N_\ell^3(\frac{1}{N_C}-\delta_C)^2} < \varepsilon/3. \tag{4}$$

Equation (4) is thus the constraint that needs needs to be satisfied for SH3.

Finally, for SH1, fix $M \geq N_\ell$, K such that $N_\ell \leq K \leq M$, a round t, an miner i, and a color c, and let N_C be the number of colors and $\overline{\alpha}_{i,t,M}$ be i's average power in the interval $[t, t+M]$. Take

$$\delta = (1/2-\alpha)/2. \tag{5}$$

Let $\mathcal{H}_{t,M,K,i}$ consist of all histories where, in the subinterval $[t, t+M]$ of $[0, T_{\max}]$, there are exactly $K \geq N_\ell$ blocks of color c, at least a fraction $1/2-\delta$ of them are generated by miner i. The probability of there being exactly K blocks of color c in the interval is $\binom{M}{K}\gamma^K(1-\gamma)^{M-K}$. Applying Hoeffding's inequality, the probability of being at least $\delta+\alpha$ away from the mean $\overline{\alpha}_{i,t,M}$ is $e^{-2(\delta+\alpha-\overline{\alpha}_{i,t,M})^2K}$. It follows that $\Pr(\mathcal{H}_{t,M,K,i}) \leq \binom{M}{K}\gamma^K(1-\gamma)^{t_2'-K}e^{-2(\delta+\alpha-\overline{\alpha}_{i,t,M})^2K}$.

Let $\mathcal{H}_{t,M,K}$ consist of all histories where, in the interval $[t, t+M]$, there are exactly $K \geq N_\ell$ blocks of color c, and of these, greater than $1/2-\delta$ were generated by some miner i. Thus, $\mathcal{H}_{t,M,K} = \cup_i \mathcal{H}_{t,M,K,i}$, so

$$\Pr(\mathcal{H}_{t,M,K}) \leq \sum_i \Pr(\mathcal{H}_{t,M,K,i}) \leq \sum_i \binom{M}{K}\gamma^K(1-\gamma)^{M-K}e^{-2(\delta+\alpha-\overline{\alpha}_{i,t,M})^2K}.$$

Suppose that

$$N_\ell \geq 4/\delta^2. \tag{6}$$

Then we show that

$$\sum_i e^{-2(\delta+\alpha-\overline{\alpha}_{i,t,M}))^2K} \leq \lceil 1/\alpha \rceil e^{-2\delta^2K}. \tag{7}$$

To see this, recall that, by assumption, $\overline{\alpha}_{i,t,M} \leq \alpha$, and $\sum_i \overline{\alpha}_{i,t,M} = 1$. Straightforward calculus (details given below) shows that if $\alpha \geq x+z$, $z \leq y \leq x$, and $N > 1/4\delta^2$, then

$$e^{-2(\delta+\alpha-x-z)^2K} + e^{-2(\delta+\alpha-y+z)^2K} \geq e^{-2(\delta+\alpha-x)^2K} + e^{-2(\delta+\alpha-y)^2K}. \tag{8}$$

That is, if $x \geq y$, shifting a little of the weight from y to x increases the sum. It easily follows from this that the sum is maximized if we have as many miners as possible with weight α, and one miner with whatever weight remains. Given that the sum of the weights is 1, we will have roughly $1/\alpha$ miners with weight α. The desired inequality (7) easily follows. Thus,

$$\Pr(\mathcal{H}_{t,M,k}) \leq \binom{M}{K}\gamma^K(1-\gamma)^{M-K}\lceil 1/\alpha\rceil e^{-2\delta^2 K}.$$

Here are the details of the calculation for (8): It's clear that the two sides of the inequality are equal if $z=0$, So we want to show that the left-hand side increases as z increases. Taking the derivative, it suffices to show that $4(\delta+\alpha-x-z)Ke^{-2(\delta+\alpha-x-z)^2K} - 4(\delta+\alpha-y+z)Ke^{-2(\delta+\alpha-y+z)^2K} \geq 0$ if $z \geq 0$, or equivalently, that $f(z) = (\delta+\alpha-x-z)e^{-2(\delta+(\alpha-x-z)^2K} - (\delta+\alpha-y+z)e^{-2(\delta+\alpha-y+z)^2K} \geq 0$ if $z \geq 0$. We first consider what happens if $z=0$. We must show that $(\delta+\alpha-x)e^{-2(\delta+\alpha-x)^2K} \geq (\delta+\alpha-y)Ke^{-2(\delta+\alpha-y)^2K}$ if $x \geq y$. The two sides are equal if $x=y$. Taking the derivative with respect to x, it suffices to show that $-e^{-2(\delta+\alpha-x)^2K}+4(\delta+\alpha-x)^2Ke^{-2(\delta+\alpha-x)^2K} \geq 0$, or equivalently, that $4(\delta+\alpha-x)^2K-1 \geq 0$. Since $K \geq N_\ell > 1/4\delta^2$ by (5) and $\delta < 1/4$, we have that $f(0) > 0$. Next note that $f'(z) = -e^{-2(\delta+\alpha-x-z)^2K} + 4(\delta+\alpha-x-z)^2Ke^{-2(\delta+\alpha-x-z)^2K} + e^{-2(\delta+\alpha-y+z)^2K} - 4(\delta+\alpha-y+z)^2Ke^{-2(\delta+\alpha-y+z)^2K}$. If $K > 1/4\delta^2$, then $f'(z) = \eta_1 e^{-2(\delta+\alpha-x-z)^2K} - \eta_2 e^{-2((\delta+\alpha-y+z)^2K}$, where $\eta_1 > 0$ and $\eta_2 < 0$. Thus, $f'(z) > 0$, as desired.

Note that $\cup_{\{t,M,K:\ N_\ell \leq K \leq M \leq T_{\max},\ t \leq T_{\max}-M\}}\mathcal{H}_{t,M,K}$ consists of all histories where there are at least N_ℓ blocks of color c and, of these, at least $1/2-\delta$ are generated by some miner i.

$$\begin{array}{ll} & \Pr(\cup_{\{t,M,K:\ N_\ell \leq K \leq M \leq T_{\max},\ t \leq T_{\max}-M\}}\mathcal{H}_{t,M,K}) \\ \leq & \sum_{\{M:\ N_\ell \leq M \leq T_{\max}\}}(T_{\max}-M)\lceil 1/\alpha\rceil \sum_{\{K:\ N_\ell \leq K \leq M\}}\binom{M}{K}\gamma^K(1-\gamma)^{M-K}e^{-2(\delta/2)^2K} \\ \leq & \sum_{\{M:\ N_\ell \leq M \leq T_{\max}\}}T_{\max}\lceil 1/\alpha\rceil e^{-2(\delta/2)^2N_\ell}\sum_K\binom{M}{K}\gamma^K(1-\gamma)^{M-K} \\ \leq & T_{\max}^2\lceil 1/\alpha\rceil e^{-2(\delta/2)^2N_\ell}. \end{array}$$

Since SH1 must holds for all colors c, SH1 holds with probability greater than $1-\varepsilon/3$ if

$$N_C T_{\max}^2\lceil 1/\alpha\rceil e^{-2(\delta/2)^2N_\ell} < \varepsilon/3. \tag{9}$$

To get all of SH1, SH2, and SH3 to hold with probability at least $1-\varepsilon$, we must choose N_ℓ, N_C, $T_{\max}$, δ, and δ_C so that constraints (3), (4), (5), (6), and (9) all hold. Given α, (5) determines δ. We take it to have this value. Recall that $\delta < 1/4$. Given Δ, we next choose N_C sufficiently large such that $(\frac{N_C-1}{N_C})^{\Delta-1} > \frac{1}{2}$. We then choose $\delta_C < \frac{1}{2N_C}$. Finally, for reasons that will become clear shortly, we replace $T_{\max}$ in the equations by N_ℓ^2. (We could equally well have used N_ℓ^k for $k > 2$.) With this replacement and the choices above, we can simplify (3), (4), and (9) to

$$\begin{array}{r} N_C N_\ell^4 e^{-2N_\ell^3/16} < \varepsilon/3 \\ N_C N_\ell^4 e^{-2N_\ell^3(\delta_C/2)^2} < \varepsilon/\text{ and} \\ N_C N_\ell^4\lceil 1/\alpha\rceil e^{-2(\delta/2)^2N_\ell} < \varepsilon/3. \end{array} \tag{10}$$

Given N_C, δ, δ_C as determined above, we can clearly choose N_ℓ^* sufficiently large to ensure that these inequalities, together with (6), hold for all $N_\ell > N_\ell^*$. Take $T_{\max}^* = (N_\ell^*)^2$. It follows that for all $T_{\max} \geq T_{\max}^*$, for $\sqrt{T_{\max}} < N_\ell < T_{\max}$, all the constraints hold. This completes the proof. ◀

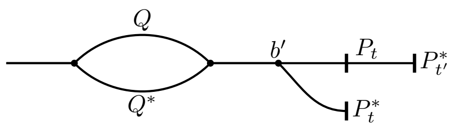


Figure 5 Paths P_t (and $P^*_{t'}$) that are identical to P^*_t up to b'.

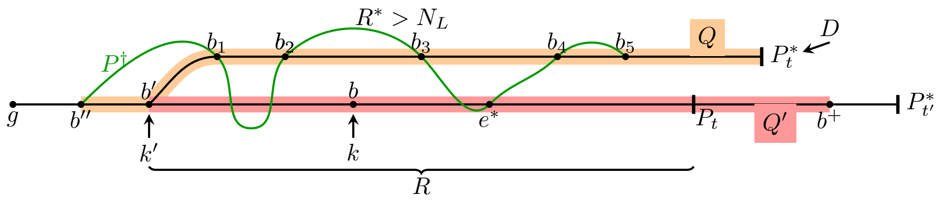


Figure 6 Ledgers in $G^{h(t)}_{i,\hat{c}}$ and $G^{h(t')}_{j,\hat{c}}$ that are identical except for their suffixes.

B Verifying the Colordag Ledger Properties

We prove the three ledger properties.

▶ **Proposition 16** (Colordag ledger consistency). *If $(N_C, N_\ell, \delta, \delta_C, T_{max})$ is suitable for ε and $\alpha < 1/2$ then for all miners i, j and all histories $h \in H^{N_C,N_\ell,\delta,\delta_C,T_{max}}$, if all but at most one miner is honest in h, $t \le t'$, and $k \le |\mathcal{L}(G^{h(t)}_i)| - N_\ell$, then $\mathcal{L}_k(G^{h(t)}_i) = \mathcal{L}_k(G^{h(t')}_j)$.*

Proof. Suppose that $\mathcal{L}_k(G^{h(t')}_j) = b$ and $k \le |\mathcal{L}(G^{h(t)}_i)| - N_\ell$. Let $P^*_{t'}$ be the canonical longest path in $G^{h(t')}_{j,\hat{c}}$. Let P_t be its prefix in $G^{h(t)}_{i,\hat{c}}$ and let P^*_t be the canonical longest path in $G^{h(t)}_{i,\hat{c}}$ (see Figure 5).

Let b' be the last common block on P^*_t and P_t. We claim that P^*_t and P_t must be identical up to b'. For if they diverge before b', there must be subpaths Q^* and Q of P^*_t and P_t, respectively, that are disjoint except for their first and last nodes. Since P^*_t and $P^*_{t'}$ are longest paths, we must have $|Q^*| = |Q|$ (if, for example, $|Q^*| > |Q|$, then we can find a path longer that $P^*_{t'}$ by replacing the Q segment by Q^*). The canonical choice will be the same for P^*_t and $P^*_{t'}$, providing the desired contradiction, so the prefixes are the same up to b'.

Let $D = |\mathcal{L}(G^{h(t)}_i)|$ (see Figure 6). Since P^*_t is a longest path in $G^{h(t)}_{i,\hat{c}}$, its length is D. Suppose, by way of contradiction, that b is not on P^*_t. Both blocks b and b' are on $P^*_{t'}$, and block b cannot precede b' on its prefix P_t, otherwise it would be on P^*_t. Thus, b' precedes b, and we must have $b' = \mathcal{L}_{k'}(G^{h(t)}_i)$, where $k' < D - N_\ell$. Since $|\mathcal{L}(G^{h(t)}_i)| = d(G^{h(t)}_{i,\hat{c}})$, it follows that $d(G^{h(t)}_{i,\hat{c}}, b') < D - N_\ell$. (We note for future reference, since it is used in the proof of Proposition 23, that the contradiction comes from this fact.) It follows that the segment R^* of P^*_t from b' to the end must have length greater than N_ℓ. Moreover, if R is the segment of P_t from b' to the end, then R and R^* must be disjoint except for their initial block b'.

We now get a contradiction by considering a path $P^\dagger$ that includes all the honest blocks in $G^{h(t')}_{i,\hat{c}}$ that are not naturally forked. Let b'' be the last block at or preceding b' that is honest and not naturally forked. (If b' is honest and not naturally forked, then $b'' = b'$.) Consider the subpath going from b'' to b' followed by R^*. Call this path Q (highlighted in Figure 6). $P^\dagger$ must intersect Q. For if not, there must be at least as many blocks on Q as there are on $P^\dagger$ generated at or before time t (since P^*_t is the canonical longest path), but none of the blocks on Q other than b'' is an honest block that is not naturally forked.

Suppose that b'' is generated at time t''. It follows that in the interval $[t''+1, t]$, fewer honest blocks that are not naturally forked are generated than dishonest blocks, contradicting the assumption that $h \in H^{N_C, N_\ell, \delta, \delta_C, T_{\max}}$.

Without loss of generality, suppose that, starting at b'', $P^\dagger$ intersects with R^* after it intersects with R. (If $P^\dagger$ does not intersect with R at all, we take R to be the path it intersects with later. The argument is the same if $P^\dagger$ intersects with R after it intersects with R^*.) Let $b_1, b_2, \ldots, b_k$ be the blocks on $P^\dagger$ that are also on R^*, in the order that they appear. For convenience, we take $b_k = b^*$ (the virtual final block). For each pair e, e' of consecutive blocks in $b_1, \ldots, b_k$, the path from e to e' on Q must be at least as long as the path from e to e' on $P^\dagger$ (if $e' = b^*$, we take the path from e to e' on $P^\dagger$ to be the subpath of $P^\dagger$ starting from e and including all the blocks generated at or before time t). It follows that there are at least as many blocks on Q that are not on $P^\dagger$ as there are blocks on $P^\dagger$ that are generated after b'' and at or before time t and are not on Q. We can repeat this process with R to show, roughly speaking, that there are at least as many blocks on R that are not on $P^\dagger$ as there are on $P^\dagger$ that are generated after b'' and at or before time t that are not on R. Suppose that b'' is generated at time t''. It follows that there are at least as many blocks that are either not honest or naturally forked generated between time t'' and t as there are honest blocks that are not naturally forked. This contradicts the assumption that $h \in H^{N_C, N_\ell, \delta, \delta_C, T_{\max}}$.

The reason that we said "roughly speaking" above is that this argument does not work in one special case. Suppose that the final block on R that is also on $P^\dagger$ is e^*. Further suppose that there are blocks on $P^\dagger$ that are generated at or before time t but after e^*. We cannot conclude that the path from e^* to b^* on R is at least as long as the subpath of $P^\dagger$ consisting of blocks generated after e^* and at or before time t, since R is not necessarily a longest path up to time t.

We deal with this as follows. Let Q' (highlighted in Figure 6) be the segment of $P^*_{t'}$ starting at b' and ending with the first honest block that is not naturally forked that is generated after time t. Call this block b^+. Note that R is a prefix of Q'. Moreover, the subpath of Q' from c to b^+ is indeed at least as long as the subpath of $P^\dagger_{t'}$ from c to b^+. The upshot of this argument is that there are more blocks on Q and R (or Q') that are not on $P^\dagger$ than there are blocks on $P^\dagger$ after b'' that are generated at or before time t (or up to b^+, if we consider Q'). As before, this gives a contradiction to the fact that $h \in H^{N_C, N_\ell, \delta, \delta_C, T_{\max}}$.

Therefore, our initial assumption was wrong and we conclude that b is on P^*_t. Therefore, it precedes the last common block b' on both P^*_t and $P^*_{t'}$. Since we have shown the two paths coincide until b', it follows that $\mathcal{L}_k(G_i^{h(t)}) = \mathcal{L}_k(G_j^{h(t')})$. ◀

▶ **Proposition 17** (Colordag ledger growth). *If $(N_C, N_\ell, \delta, \delta_C, T_{max})$ is suitable for ε and $\alpha < 1/2$, then for all rounds t and t' such that $t' - t \geq N_\ell/\delta_C$, if all but at most one miner is honest in $h \in H_i^{N_C, N_\ell, \delta, \delta_C, T_{max}}$, then $|\mathcal{L}^{cd}(G_i^{h(t')})| \geq |\mathcal{L}^{cd}(G_i^{h(t)})| + 1$.*

Proof. Suppose that $h \in H^{N_C, N_\ell, \delta, \delta_C, T_{\max}}$. Consider rounds t and t' such that $t'-t \geq 2N_\ell/\delta_C$. Since $t'-t \geq 2N_\ell/\delta_C$ and $h \in H_i^{N_C, N_\ell, \delta, \delta_C, T_{\max}}$ there are $K \geq 2N_\ell$ blocks of color $\hat{c}$ generated in this interval. Because h is safe, more than $K/2 \geq N_\ell$ of these blocks are honest and not naturally forked. Let $P^\dagger$ be a path that includes all of these blocks. Let P^*_t denote the canonical longest path of color $\hat{c}$ up to time t. Let b be the last block on P^*_t that is on $P^\dagger$. Let M_0 be the length of P^*_t up to and including b. Suppose that there are M blocks on P^*_t following b, and M' blocks on $P^\dagger$ following b that are generated before time t. Thus, the length of P^*_t is $M_0 + M$. Note that $M \geq M'$ (since P^*_t is a longest path) and

$$M + M' < N_\ell \implies M < N_\ell \tag{11}$$

(otherwise, fewer than half the blocks generated between the time that b was generated and t are honest and not naturally forked, despite the fact that at least N_ℓ blocks are generated in that interval). Now the subpath of $P^\dagger$ up to time t' has length greater than $M_0 + M' + K/2 \geq M_0 + M' + N_\ell$, so the canonical path up to time t' must have at least this length. Thus, for the canonical path up to time t' we have

$$|\mathcal{L}^{\mathrm{cd}}(G_i^{h(t')})| \geq M_0 + M' + N_\ell \geq M_0 + N_\ell \overset{\text{Eq. 11}}{>} M_0 + M = |\mathcal{L}^{\mathrm{cd}}(G_i^{h(t)})|$$ ◀

▶ **Proposition 18** (Colordag ledger quality)**.** *If $(N_C, N_\ell, \delta, \delta_C, T_{max})$ is suitable for ε and $\alpha < 1/2$ then for all rounds t and t' such that $t' - t \geq 2N_\ell/\delta_C$, and all $h \in H_i^{N_C,N_\ell,\delta,\delta_C,T_{max}}$, at least two of the blocks of color $\hat{c}$ added to $\mathcal{L}(G_i^{h(t')})$ in the interval $[t, t']$ are generated by honest miners.*

Proof. As we argued in the proof of Proposition 17, since $t' - t \geq 2N_\ell/\delta_C$, there are at least $2N_\ell$ blocks of color $\hat{c}$ in the interval (by SH3), so we must have at least N_ℓ blocks that are honest and not naturally forked (by SH1 and SH2). Let $P^*_{t'}$ be the canonical longest path up to time t' and let $P^\dagger$ be a path that includes all the honest blocks of color $\hat{c}$ that are not naturally forked up to time t'. Let b be the last honest block that is not naturally forked on $P^*_{t'}$ that is generated prior to time t (b is the genesis block if no other honest blocks on $P^*_{t'}$ are generated prior to time t). We claim that there must be at least two honest blocks that are not naturally forked on $P^*_{t'}$ that come after b. First suppose that there are none. Then there are at least as many blocks on $P^*_{t'}$ that are generated after b as there are on $P^\dagger$ that are generated after b, so, as before, we get a contradiction to the fact that $h \in H_i^{N_C,N_\ell,\delta,\delta_C,T_{\max}}$.

Next suppose that there is only one block, say b', on $P^*_{t'}$ that is generated after b that is honest and not naturally forked (see Figure 4). Note that there are more than N_ℓ blocks on $P^\dagger$ after b and hence more than N_ℓ on $P^*_{t'}$ after b (since $P^*_{t'}$ is a longest path). Consider the subpath of $P^*_{t'}$ strictly between b and b' and the subpath of $P^\dagger$ strictly between b and b'. If the total number of blocks on these subpaths is at least N_ℓ, then property SH1 does not hold and we have a contradiction to $h \in H_i^{N_C,N_\ell,\delta,\delta_C,T_{\max}}$. If not, then the total number of blocks on the subpath of $P^\dagger$ strictly after b and the subpath of $P^*_{t'}$ strictly after b must be at least N_ℓ, so again we get a contradiction to $h \in H_i^{N_C,N_\ell,\delta,\delta_C,T_{\max}}$. ◀

Certified Round Complexity of Self-Stabilizing Algorithms

Karine Altisen
Université Grenoble Alpes, CNRS, Grenoble INP,[1] VERIMAG, 38000 Grenoble, France

Pierre Corbineau
Université Grenoble Alpes, CNRS, Grenoble INP,[1] VERIMAG, 38000 Grenoble, France

Stéphane Devismes
Université de Picardie Jules Verne, MIS, 80039 Amiens, France

Abstract

A proof assistant is an appropriate tool to write sound proofs. The need of such tools in distributed computing grows over the years due to the scientific progress that leads algorithmic designers to consider always more difficult problems. In that spirit, the *PADEC* Coq library has been developed to certify self-stabilizing algorithms. Efficiency of self-stabilizing algorithms is mainly evaluated by comparing their stabilization times in *rounds*, the time unit that is primarily used in the self-stabilizing area. In this paper, we introduce the notion of rounds in the PADEC library together with several formal tools to help the certification of the complexity analysis of self-stabilizing algorithms. We validate our approach by certifying the stabilization time in rounds of the classical Dolev *et al*'s self-stabilizing Breadth-first Search spanning tree construction.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Theory of computation → Logic and verification

Keywords and phrases Certification, proof assistant, Coq, self-stabilization, round complexity

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.2

Funding This work has been partially funded by the ANR project SkyData (ANR-22-CE25-0008-01).

1 Introduction

Proving the correctness and analyzing the time complexity of distributed algorithms, especially fault-tolerant ones, is usually complex and subtle due to the many uncertainties we have to face, *e.g.*, locality of information, asynchrony of communications, faults, topological changes, just to quote a few. In this context, *certification* is an appropriate method to increase confidence of algorithmic designers in the functional and non-functional properties of their solutions. Indeed, the certification consists in formally writing proofs using a proof assistant, a software solution such as *Coq* [40, 7] or *Isabel/HOL* [34] that allows to develop formal proofs interactively and *mechanically check* them.

It is important to note that to guarantee the soundness of proofs, a proof assistant requires a level of detail that is drastically higher than in paper-and-pencil proofs and often necessitates a full reengineering of the initial proof. As a consequence, importing a paper-and-pencil proof into a proof assistant is usually an intricate task. However, to circumvent this difficulty, many libraries have been developed to facilitate the work of proof designers, *e.g.*, [8, 5, 1]. Such libraries mainly tackle two orthogonal goals: (1) they help to write formal proofs to prevent bugs while (2) keeping them readable and understandable for a non-expert in certification.

[1] Institute of Engineering Univ. Grenoble Alpes

PADEC [1] is a library for the certification of distributed self-stabilizing algorithms written in the *atomic-state model* [21], the most commonly used model in the self-stabilizing area. This library is based on the proof assistant Coq. It contains formal definitions and tools whose suitability has been demonstrated through several relevant use cases from the literature.

Self-stabilization is a versatile and lightweight fault-tolerant paradigm of distributed computing [21, 4]. A self-stabilizing algorithm enables a distributed system to resume a correct behavior within finite time, regardless its initial configuration; and therefore also after a finite number of transient faults place it in an arbitrary configuration. It is worth noting that self-stabilization makes no hypotheses on the nature (*e.g.*, memory corruption or topological changes) or extent of transient faults that could hit the system, and self-stabilizing systems recover from the effects of those faults in a unified manner. Such versatility comes at a price, *e.g.*, after transient faults cease, there is a finite period of time, called the *stabilization phase*, during which the safety properties of the system are violated. Hence, self-stabilizing algorithms are mainly compared according to their *stabilization time*, the worst-case duration of the stabilization phase.

To evaluate (stabilization) time, three main units are used in the atomic-state model: moves, (atomic) steps, and rounds. A move corresponds to a local state update at some process. Actually, it is rather a unit of work since it captures the amount of computations an algorithm needs. Steps essentially captures the same information as moves: a step is a global transition in an execution. *Rounds* [22, 13] evaluate the execution time according to the speed of the slowest processes. It is a non-atomic unit contrary to the two previous ones. Essentially, it is the adaptation to the atomic-state model of the notion of *time units* used in the message-passing model [39]. Roughly speaking, from a given configuration, a round is over as soon as all processes get a chance to move at least once.

The concept of steps have been already imported in PADEC [2]. Yet, as for moves, complexities in steps somehow neglect the parallel aspects of the distributed algorithm they evaluates. As a matter of fact, worst-case executions in steps are most of the time sequential; see, *e.g.*, [3, 2]. Perhaps, this is why the rounds are the most commonly used time units in the self-stabilizing area.

Contribution. In this paper, we enrich the PADEC library with the concept of rounds. Certifying complexities, especially stabilization times, in rounds is a major concern since it allows to increase confidence in the soundness of claimed bounds. As a matter of fact, paper-and-pencil proven complexity bounds are sometimes inaccurate due to implicit assumptions and a lack of details. For example, the stabilization time of Huang and Chen's Breadth-first Search (BFS) spanning tree construction [27] was conjectured to stabilize in $O(\mathcal{D})$ rounds, where $\mathcal{D}$ is the network diameter. Now, this algorithm is actually made of two non-mutually exclusive rules and the absence of priority on those rules leads to a possible execution that stabilizes in $\Omega(n)$ rounds, where $n > \mathcal{D}$ is the number of processes [19].

We add the formal definition of rounds in PADEC together with several companion formal tools whose aim is to ease the certification by making it as close as possible to the paper-and-pencil round complexity analyses one can find in the self-stabilizing area. To achieve this, we provide several certified meta-theorems consisting in general proof patterns allowing the users to mimic the usual way round complexities are proven. Thus, they can focus on the true difficulty of the result instead of drowning the proof in tedious details requested by the proof assistant.

We validate our approach and illustrate the usefulness of our general formal tools by certifying the stabilization time in rounds of the straightforward translation into the atomic-state model of Dolev *et al*'s algorithm [22], which was initially written in the Read/Write atomicity model. This latter constructs a BFS spanning tree in a rooted bidirectional connected network. This task is fundamental in the self-stabilizing area since it is widely used as a basic building block of more complex self-stabilizing solutions; see, *e.g.*, [24, 17]. Notice that we also certify the self-stabilization of our use case assuming a weakly fair daemon.

Beyond the certification in Coq, our work leads to a better understanding of the intrinsic nature of non-atomic time units such as rounds.

Related Work. Many formal approaches have been used in the context of distributed computing. There exist exhaustive tool suites to validate a given distributed algorithm, such as the TLA+ toolbox [32]. Synthesis [9, 23] aims at automatically constructing algorithms based on a given specification, a fixed topology, and sometimes a restricted scheduling (*e.g.*, synchronous execution); this technique is now often based on SMT-solvers. Verification using model-checking [41, 31] is also fully automated and requires to fix settings similarly to synthesis. Both synthesis and model-checking only succeed with small topologies, due to computation limits. Notice also that model checking has been also successfully used to prove impossibility results applying on small-scale distributed systems [20]. In contrast, a proof assistant allows to validate a given algorithm for arbitrary-sized topologies, but is only semi-automated and may require heavy development for each algorithm, justifying then the development of helpful libraries.

The correctness of several non fault-tolerant distributed algorithms have been certified; *e.g.*, Castéran and Filou [10] consider distributed algorithms written in the local model, and a certified proof of Lamport's Bakery algorithm is given in [26]. Certification of fault-tolerant, yet non self-stabilizing, distributed systems has been addressed using various proof assistants, *e.g*, in Isabel/HOL [28, 12, 11, 29], TLA+ [16, 18], Coq [38], and Nuprl [36, 37]. This so-called *robust* fault tolerance approach aims at masking the effect of faults, whereas self-stabilization is non-masking by essence. Hence, the techniques used for these two approaches are widely different. In the robust context, many certification results are related to agreement problems, such as consensus or state-machine replication, in fully connected networks. Overall, most of these aforementioned works only certify the safety property of the considered problem [16, 18, 36, 37, 28, 38]. However, both liveness and safety properties are certified in [12, 11, 29]. To the best of our knowledge, the certification of time complexity of robust fault-tolerant algorithms has never been addressed. Finally, robust fault tolerance has been also considered in the context of mobile robot computing: using the PACTOLE Coq framework, impossibility results for swarm robotics that are subjected to Byzantine faults have been certified [6, 15]. Once again, to the best of our knowledge, certification of time complexity has never been addressed in the robot context.

Several frameworks to certify self-stabilizing algorithms using the Coq proof assistant have been proposed, *e.g.*, [14, 1]. In particular, the PADEC framework has already been used to certify the exact stabilization time in *steps* of the first Dijkstra's self-stabilizing token ring algorithm [2]. Certification of the correctness (safety and liveness) of the first Dijkstra's token ring algorithm has been previously achieved using various proof assistants, *i.e.*, PVS [35, 25, 30] and Isabel/HOL [33]. Interestingly, Fokkink *et al.* [25] have certified a quantitative property; precisely they show that the minimum number of states per node the algorithm needs to converge in any sequential execution is $N - 1$, where N is the number of nodes. However, overall among these works, only PADEC addresses time complexity issues.

Coq Development. The development for this contribution represents about 11,000 lines of Coq code (loc, as measured by `coqwc`), precisely `#loc: spec = 2,698; proof = 7,892; comments = 484`. The Coq development related to the paper is available as an online browsing documentation at `http://www-verimag.imag.fr/~altisen/PADEC`. We encourage the reader to visit this webpage for a deeper understanding of our work.

Roadmap. The rest of the paper is organized as follows. In Section 2, we present our use case and the PADEC framework. Section 3 is devoted to the formalization of rounds in PADEC. In Section 4, we illustrate how to use the general tools given in the previous section to certify the round complexity of our use case. We make concluding remarks in Section 5.

2 A BFS Spanning Tree Algorithm and its Certification

In this section, we present an algorithm, denoted by $\mathcal{BFS}$, which will be used as the common use case all along the paper. Algorithm $\mathcal{BFS}$ allows us to define self-stabilization, the atomic-state model, and its semantics. We also use this algorithm as an illustrative example to introduce the PADEC framework and the method to certify a self-stabilizing algorithm.

2.1 Algorithm Definition and Informal Model

$\mathcal{BFS}$ is a self-stabilizing distributed algorithm that computes a BFS spanning tree in an arbitrary rooted, connected, and bidirectional network. By "bidirectional", we mean that each node can both transmit and acquire information from its adjacent nodes in the network topology, *i.e.*, its neighbors. The algorithm being distributed, these are the only possible direct communications. "Rooted" indicates that a particular node, called the root and denoted by r, is distinguished in the network. As in the present case, algorithms for rooted networks are (usually) semi-anonymous: all nodes have the same code except the root.

Algorithm 1 Algorithm $\mathcal{BFS}$, code for each node p.

```
Constant Local Input:
  $p.neighbors \subseteq Channels$; $p.root \in \{true, false\}$
  /* p.neighbors as well as other sets below are implemented as lists */
Local Variables:
  $p.d \in \mathbb{N}$; $p.par \in Channels$
Macros:
  $Dist_p = \min\{q.d + 1, q \in p.neighbors\}$
  $Par_{dist}$ returns the first channel in the list $\{q \in p.neighbors, q.d + 1 = p.d\}$
Action for the root, i.e., for $p$ such that $p.root = true$
  Action $Root$:   if $p.d \neq 0$ then $p.d := 0$
Actions for any non-root node, i.e., for $p$ such that $p.root = false$
  Action $CD$:   if $p.d \neq Dist_p$ then $p.d := Dist_p$
  Action $CP$:   if $p.d = Dist_p$ and $p.par.d + 1 \neq p.d$ then $p.par := Par_{dist}$
```

Algorithm $\mathcal{BFS}$ is written in the *atomic-state model*, where nodes communicate through locally shared variables: a node can read its variables and those of its neighbors, but can only write to its own variables. Every node can access its neighbors (to read its variables) through (local) channels.

The network is locally defined at each node p using constant inputs. The fact that the network is rooted is implemented using a constant Boolean input called $p.root$ which is false for every node except r. The input $p.neighbors$ is the set of channels linking p to its neighbors. When it is clear from the context, we do not distinguish a neighbor from the channels to that neighbor.

$\mathcal{BFS}$ is the straightforward translation into the atomic-state model of Dolev *et al*'s algorithm [22], which was initially written in the Read/Write atomicity model. Its code is given in Algorithm 1 as a set of three locally-mutually-exclusive actions. Each action is of the form: **if** *condition* **then** *statement*. In the following, we say that an action is *enabled* when its condition is true. By extension, a node is said to be enabled when at least one of its actions is enabled.

The semantics of the system is defined as follows. The current system *configuration* is given by the current value of all variables at each node. If no node is enabled in the current configuration, then the configuration is said to be *terminal* and the execution is over. Otherwise, a *step* is performed: a *daemon* (an oracle that models the asynchronism of the system) *activates* a non-empty set of enabled nodes. Each activated node then *atomically executes* the statement of its enabled action, leading the system to a new configuration, and so on and so forth.

Assumptions can be made about the daemon. Here, we assume that the daemon is *weakly fair* meaning that every continuously enabled nodes is eventually chosen by the daemon. More precisely, this means that every enabled node is eventually either activated or *neutralized*. A node p is neutralized in the step from configuration γ to configuration γ' if p is enabled in γ but not in γ' while being not activated during that step. Such situation occurs when a node is made disabled by the activation of some of its neighbors.

In Algorithm $\mathcal{BFS}$, each node p maintains two variables. First, each node p evaluates in $p.d$ its distance to the root. Then, each non-root node p maintains the pointer $p.par$ to designate as *parent* a neighbor that is closest to the root (*n.b.*, $r.par$ is meaningless). Algorithm $\mathcal{BFS}$ is a self-stabilizing BFS spanning tree construction in the sense that, regardless the initial configuration, it makes the system converge to a terminal configuration where par-variables describe a BFS spanning tree rooted at r. To that goal, nodes first compute into their d-variable their distance to the root. The root simply forces the value of $r.d$ to be 0; see Action $Root$. Then, the d-variables of other nodes are gradually corrected: every non-root node p maintains $p.d$ to be the minimum value of the d-variables of its neighbors incremented by one; see $Dist_p$ and Action CD. In parallel, each non-root node p chooses as parent a neighbor q such that $q.d = p.d - 1$ when $p.d$ is locally correct (*i.e.*, $p.d = Dist_p$) but $p.par$ is not correctly assigned (*i.e.*, $p.par.d$ is not equal to $p.d - 1$); see Action CP.

2.2 The PADEC Library

PADEC [1] is a general framework for the certification in Coq [7] of self-stabilizing algorithms. It includes the definition of the atomic-state model, tools for the definition of the algorithms and their properties, lemmas for common proof patterns, and case studies. The atomic-state model is carefully defined in PADEC to be as close as possible to the standard usage of the self-stabilizing community. Moreover, it is made general enough to encompass every usual hypothesis (*e.g.*, about topology or scheduling). First, a finite network is described using types `Node` and `Channel`, which respectively represent the nodes and the links between nodes. Then, the distributed algorithm is defined by providing a local algorithm at each node. This latter is defined using a type `State` that represents the local state of a node (*i.e.*, the values of its local variables) and a function `run` that encodes the local algorithm itself. Function `run` computes a new state depending on the current state of the node and that of its neighbors.

The model semantics defines a *configuration* as a function of type `Env := Node` $\rightarrow$ `State` that provides the (local) state of each node. An *atomic step* of the distributed algorithm is encoded as a binary relation over configurations that checks the conditions given in the informal model; see Section 2.1. An *execution* `e` is a finite or infinite *stream* of configurations, which models a *maximal* sequence of configurations where any two consecutive configurations are linked by the step relation. “Maximal” means that `e` is finite if and only if its last configuration is terminal. We use the coinductive[1] type `Exec` to represent an execution stream and the coinductive predicate `is_exec: Exec` $\rightarrow$ **Prop**[2] to check the above condition.

Daemons are also defined as predicates over executions using Linear Time Logic (LTL) operators provided in the PADEC library. For example, the fact that an execution is scheduled according to a weakly fair daemon is expressed by the following property: for every node `n`, it is `Always` (*a.k.a.* **G**lobally) the case that if `n` is enabled, then `Eventually` (*a.k.a.* **F**inally) `n` is activated or neutralized.

The semantics that uses the step relation is referred to as the *relational* semantics. As a way to strengthen the framework, PADEC also defines a *functional* semantics, which produces traces (*i.e.*, finite prefixes) of executions; those two semantics are proven to be equivalent.

Self-stabilization in PADEC is defined according to the usual practice: the property is formalized as a predicate `self_stabilization SPEC` that depends on the predicate `SPEC: Exec` $\rightarrow$ **Prop**, the specification of the algorithm. An algorithm is *self-stabilizing w.r.t. the specification* `SPEC` if there exists a set of legitimate configurations, encoded by some property `Leg: Env` $\rightarrow$ **Prop**, that satisfies the following three properties in every execution `e`:

- if `e` starts in a legitimate configuration (*i.e.*, if `Leg (H e)` holds, where `H e` is the first configuration of `e`), then `e` only contains legitimate configurations (*Closure*);
- `e` eventually reaches a legitimate configuration (*Convergence*); and
- if `Leg (H e)` holds, then `e` satisfies the intended specification, *i.e.*, `SPEC e` holds (*Specification*).

An algorithm is *silent* when each of its executions eventually reaches a terminal configuration; in such a case, the set of legitimate configurations is chosen to be the set of terminal configurations. Again, the closure, convergence, and silent properties use the LTL predicates `Always` and `Eventually`.

2.3 The Formal Algorithm

The formal algorithm is encoded in PADEC as a straightforward faithful translation in Coq of Algorithm 1. Together with its formal code, we have developed several technical results to facilitate the formal proof and complexity analysis of Algorithm $\mathcal{BFS}$.

We also had to encode the specification of $\mathcal{BFS}$ into PADEC. To that goal, we have defined the network in PADEC using the PADEC types `Node` and `Channel` as well as the predicate `is_channel: Node` $\rightarrow$ `Channel` $\rightarrow$ `Node` $\rightarrow$ **Prop**, where `is_channel n c n'` means that a channel `c` connects a node `n` to another `n'`. This network encodes the following graph relation: `R_Net := fun n n': Node =>` $\exists$ `c: Channel, is_channel n c n'`. Namely, an edge from Node `n` to Node `n'` exists in the graph if and only if a channel `c` connects `n` to `n'`.

Then, the BFS spanning tree specification states that the algorithm should output a subgraph T of `R_Net` such that T is a locally-defined[3] BFS spanning tree of `R_Net` rooted at a given root node r.

[1] Coinduction allows to define and reason about potentially infinite objects.
[2] Predicates in Coq have type **Prop**.
[3] In our context, locally-defined means that each non-root node should be endowed with a pointer designating its parent in T.

To express the rooted BFS spanning tree, we have defined several tools about trees, distances, and diameter. In particular, `dist: Node` $\rightarrow$ `Node` $\rightarrow$ `nat` is a constructive distance function between nodes and $\mathcal{D}$`: nat` computes the diameter of the graph. We have also introduced a few graph properties; in particular the one expressing that a graph is a subgraph of another one using inclusion of relations. We also needed to introduce the notion of DAG (Directed Acyclic Graph) and rooted trees. A DAG is a directed graph that contains no (directed) cycle or equivalently, its transitive closure is not reflexive. A graph is a (directed) tree rooted at r if it is a DAG such that (1) every node has at most one outgoing edge (the out-neighbor, if it exists, is the parent of the node), and (2) for every non-root node x, there exists a path from x to r. Finally, the relation T is defined as a BFS spanning tree rooted at r of `R_Net` if T is (1) a spanning tree, *i.e.*, a subgraph of `R_Net` containing all nodes and a tree rooted at r, and (2) BFS, *i.e.*, the distance from every node to r is the same in T and `R_Net`.

Proving the self-stabilization and silence of Algorithm $\mathcal{BFS}$ for this specification then consists in proving that all its executions eventually reach a terminal configuration where parent pointers describe a BFS spanning tree T rooted at r, provided that `R_Net` is a connected and bidirectional graph rooted at r and the daemon is weakly fair.

3 Rounds

3.1 Rounds in the Atomic-state Model

In computer science, the time complexity is the computational measure that describes the amount of computer time an algorithm uses to solve a problem. Time complexity is estimated by counting the number of transitions performed by the algorithm, *i.e.*, the number of operations that are considered to be elementary in the computational model where the time complexity is evaluated. Of course, such operations are assumed to take a fixed amount of time to be performed. For example, in sequential algorithmics, it is commonly assumed that basic operations, such as divisions or multiplications, are elementary (despite their actual implementations are often not) and so take a constant amount of time.

Here, we are interested in the complexity measure called "round" which is accurately defined using natural language in the self-stabilizing community but requires mental gymnastics since by essence, rounds (*i*) are not atomic in the computational model they are considered (*i.e.*, the atomic-state model) and (*ii*) may be infinite in certain particular cases which – to some extent – should not be taken into account in the complexity evaluation.

We should underline that there exist other non-atomic complexity measures in the literature. For example, in message-passing systems, time complexity is often evaluated in terms of *time units* [39]. To define this later, it is assumed that the message transmission time is at most one time unit and the node execution time is zero. Now, the local algorithm at each node is made of several instructions and may contain loops. In particular, in case of bug, the node may get stuck in an infinite loop. Overall, this means that in general the correctness and the complexity analysis should be studied independently, following the separation of concerns principle: once the correctness has been established, some assumptions can be made for the purpose of defining the time complexity.

Evaluation of time complexity in rounds requires first to explain how a round is built from an execution in the atomic-state model (see Subsubsection 3.1.1), and then to define what it means to achieve a given property within a given amount of rounds (see Subsubsection 3.1.3).

3.1.1 Natural Language Definition

In the atomic-state model, every execution `e` of a given algorithm is split into rounds as follows. Let `U` be the set of enabled nodes in Configuration `H e`, the first configuration of `e`. The first round of `e` terminates at the first configuration `gr` where every node in `U` has been neutralized or activated. If no such a configuration exists in `e`, then the round is infinite and actually consists in the whole (infinite) execution `e`. Otherwise, the second round of `e` is the first round of the suffix of `e` starting from `gr`; and so on and so forth.

3.1.2 Infinite Rounds

It is worth noting that the existence of an infinite round is due to a starvation generated by fairness issues. For example, imagine a situation where the activation of some node x makes another node y enable and conversely; another node z may stay continuously enabled without being ever activated by the daemon making the current round infinite. Such a situation occurs when the daemon is unfair[4] and the algorithm is actually unable to enforce fairness between enabled nodes. In contrast, when the daemon is weakly fair, fairness is guaranteed by definition. So, in every execution under the weakly fair daemon assumption, every round is finite. Remark also that when an execution contains an infinite round, it is the last one and the execution actually only contains a finite number of rounds. Conversely, if every round of an execution is finite, then the execution contains infinitely many rounds. This is in particular true for finite execution. In this latter case, infinitely many empty rounds are defined from the terminal configuration, by definition.

3.1.3 Amount of Rounds to achieve a Property

A round is a unit of time, *i.e.*, it is used to evaluate how many time is required to achieve a given property. Consider an execution `e` where a property `P: Exec` $\rightarrow$ **Prop** is eventually satisfied, *i.e.*, `e` has a suffix that satisfies `P`. The goal is then to evaluate in how many rounds `P` becomes satisfied. For example, evaluating the stabilization time in rounds of an execution of some self-stabilizing algorithm consists in counting the number of initiated rounds before a legitimate configuration is reached in the execution. If `P` is true at the first configuration of `e`, then `P` is satisfied in 0 round. Otherwise, `P` is satisfied in i rounds, where i is the index of the first round of `e` containing a configuration from which `P` is true.

As in the present paper, time complexity proofs often cope with upper bounds rather than exact ones. So, we need to express the fact that an execution *requires at most j rounds to reach* `P`. Let `e` be an execution *containing at least j rounds.* We say that `e` *requires at most j rounds to reach* `P` if `e` contains a suffix which starts before the end of the j-th round and satisfies `P`.

Remark that the assumption on `e` is necessary to cope with the possible existence of an infinite round, in which case the total number of rounds in the execution is finite and maybe smaller than j. However, contrary to more usual cases, where each unit of time is assumed to be finite, our assumption here is weaker: we only require the existence of at least j rounds, so `e` can contain an infinite round as far as it is preceded by at least $j - 1$ finite rounds.

We naturally extend the definition to all executions by fixing the property to true for each execution containing less than j rounds. Therefore, to falsify this extended property, one needs to exhibit an execution in which `P` is still not satisfied despite j rounds have elapsed.

[4] Unfair means that no fairness is imposed to the daemon, except the activation of at least one enabled node at each step.

3.2 Rounds in PADEC

We now formally express the previous definitions so that they could be encoded in PADEC.

3.2.1 Set of Unsatisfied Nodes

To compute a round, from its beginning to its end (which may never occur), we use the set `U`, called the *set of unsatisfied nodes*, which is computed as follows:

- At the beginning of the round (say at Configuration `gr`), `U` is initialized to the set of enabled nodes in `gr`; using Function `UNSAT_init gr`.
- Then, at each step (say from `g` to `g'`), the set is updated by removing the nodes that have been activated or neutralized during the step; using Function `UNSAT_update g' g`.
- The current round ends, say at Configuration `g"`, when `U` becomes empty. In this case, `U` is refilled at configuration `g"` with `UNSAT_init g"` since the next round begins.

Notice that sets of nodes are represented in PADEC using Boolean functions: `Node` $\to$ `bool`. We have defined the `PADEC.BoolSet` library to provide tools that handle sets of elements, in particular set operations (such as union, intersection, set difference, ...). The library also provides decidability results in case the set of elements is finite, which is the assumption we made for Type `Node`.

3.2.2 Predicate `At_most_rounds`

Predicate `at_most_rounds P n e` defines the fact that an execution `e` requires at most n rounds to reach a predicate `P`. It is based on an intermediate predicate, called `at_most_rounds_aux`, that has an additional parameter `U`, a set of unsatisfied nodes. Thus, `at_most_rounds` is defined as follows: `at_most_rounds P` n `e := at_most_rounds_aux P (UNSAT_init (H e))` n `e`, where the set of unsatisfied nodes (the second parameter) is initialized at the beginning of the computation of the rounds by `UNSAT_init (H e)`, *i.e.*, the set of enabled nodes in the first configuration of `e`.

We now give more details about the predicate `at_most_rounds_aux`. Recall the informal definition: "if `e` contains at least n rounds, then `e` contains a suffix which starts before the end of the n-th round and satisfies `P`". Of course, we will perform a single traversal of the execution to check both the "if" and "then" parts of the sentence. Actually, this is the role of `at_most_rounds_aux`. Since rounds may be infinite, this predicate is typically coinductive. The predicate `at_most_rounds_aux` is evaluated thanks to the following three rules. At each step of the execution, one of the rules applies.

Rule 1. The first rule, `rnd_here`, detects that the targeted predicate `P` is reached, *i.e.*, if for some execution `e`, `e` satisfies `P`, then `at_most_rounds_aux P U` n `e` holds for any values of `U` and n (even $n = 0$). Indeed, since `P e` holds, `e` requires at most n rounds to reach `P`, for any $n \geq 0$.

The two other rules achieve a traversal of the execution and update the set of unsatisfied nodes meanwhile, according to the informal definition of a round.

Rule 2. The second rule, `rnd_in`, applies when going through the current round but the round is not over (*i.e.*, right after its update, the set of unsatisfied nodes is still not empty). In this case, we decompose the execution as `g` $\bullet$ `e`, where `g` is its first configuration and `e` the subsequent suffix. Then, to satisfy `at_most_rounds_aux P U` n `(g` $\bullet$ `e)` – which claims that, given the current set of unsatisfied nodes `U`, the execution requires at most n rounds to reach `P` –

we need that `at_most_rounds_aux P U'` n `e` holds, where $n > 0$ and `U' := UNSAT_update (H e) g` is not empty, meaning that, given the updated set of unsatisfied nodes `U'`, the execution `e` requires at most n rounds to reach `P`. Indeed,

- the non-empty set `U'` is obtained by removing the nodes from `U` that have been activated or neutralized during the step from `g` to `H e`;
- and the rule applies during the current round, so the number of rounds n is positive and does not change.

Rule 3. The last rule, `rnd_chge`, applies at the end of a round. So, the number of elapsed rounds increases by one. In this case, to satisfy `at_most_rounds_aux P U` $(n+1)$ `(g • e)` – which claims that, given the current set of unsatisfied nodes `U`, the execution requires at most $n+1$ rounds to reach `P` – we need that `at_most_rounds_aux P U"` n `e` holds, where `U" := UNSAT_init(H e)`. Indeed, this time, the set of unsatisfied nodes, `UNSAT_update (H e) g`, becomes empty during the step from `g` to `H e`, so one round has passed. Consequently, the new set of unsatisfied nodes `U"` should be filled with the enabled nodes of Configuration `H e` and, given `U"`, `P` should be satisfied within at most n rounds in `e` (In particular, if $n = 0$, `e` should satisfy `P`; see Rule 1).

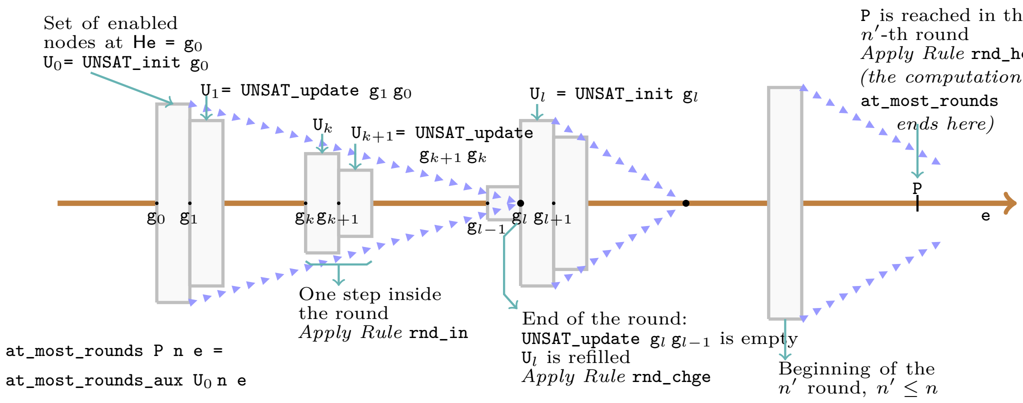

Set of enabled nodes at He = g_0
U_0= UNSAT_init g_0
U_1= UNSAT_update g_1 g_0
U_k
U_{k+1}= UNSAT_update g_{k+1} g_k
U_l = UNSAT_init g_l
P is reached in the n'-th round
Apply Rule rnd_here
(the computation of at_most_rounds *ends here)*
P
g_0 g_1
g_k g_{k+1}
g_{l-1}
g_l g_{l+1}
e
One step inside the round
Apply Rule rnd_in
End of the round:
UNSAT_update g_l g_{l-1} is empty
U_l is refilled
Apply Rule rnd_chge
Beginning of the n' round, $n' \leq n$
at_most_rounds P n e =
at_most_rounds_aux U_0 n e

Figure 1 Round principle: evaluation of `at_most_rounds` using the rules of `at_most_rounds_aux`.

Illustration. Figure 1 shows how the rules of `at_most_rounds_aux` apply to evaluate `at_most_rounds`. The horizontal line represents the execution `e` = g_0 g_1 ... starting from g_0 = `H e`. First, to evaluate Predicate `at_most_rounds P n e`, the set of unsatisfied nodes U_0 is initialized to `UNSAT_init` g_0; see the leftmost part of the figure. Then, along the steps inside the round, Rule `rnd_in` is applied and the set of unsatisfied nodes is monotonically nonincreasing. Moreover, sometimes its cardinal decreases; see *e.g.*, the step from g_k to g_{k+1} where U_{k+1} := `UNSAT_update` g_{k+1} g_k. At the end of the round, *i.e.*, at Configuration g_l, Rule `rnd_chge` applies since the update of the set of unsatisfied nodes using `UNSAT_update` g_l g_{l-1} produces an empty set. The set of unsatisfied nodes is then refilled using U_l := `UNSAT_init` g_l. Finally, `P` becomes satisfied during the `n'`-th round with `n'`$\leq$`n`. When it happens, Rule `rnd_here` applies: the evaluation stops and `at_most_rounds P n e` is satisfied.

Remark that if we remove the rightmost part of the figure, *i.e.*, if `P` is never satisfied along `e`, Rule `rnd_here` is never applied. In this case, the evaluation never stops, which is allowed since `at_most_rounds` is coinductive. Note also that the other rules may never be applied. Rule `rnd_chge` may never be applied in case the first (and last) round is infinite. Rule `rnd_in` is never applied when the execution is synchronous, *i.e.*, at each step all enabled nodes are activated.

Straightforward properties of `at_most_rounds`. We can prove that the predicate has several basic, yet interesting and useful, properties, *e.g.*:

- If `at_most_rounds P` n `e` holds, then for every $n' \geq n$, `at_most_rounds P` n' `e`.
- If Predicate $\texttt{P}_1$ implies Predicate $\texttt{P}_2$ all along the execution `e`, then `at_most_rounds` $\texttt{P}_1$ n `e` implies `at_most_rounds` $\texttt{P}_2$ n `e`.

No need to detail the proofs of these properties: they are simple coinductive proofs that directly use the definition of `at_most_rounds_aux`.

3.2.3 Functional Definition (Computation)

We also provide a functional definition, denoted by `count_rounds P e`, which *returns* in how many rounds of `e` the predicate `P` is reached (for the first time). Obviously, this function requires assumptions which enable its actual computation (precisely, which guarantees the computation eventually stops): it requires the assumption that `e` eventually reaches `P`, this assumption being expressed using a property that actually allows the computation to detect whether `P` is satisfied. This means that the reachability of `P` is encoded by an assumption – denoted by `FP` – that can be used in the function to compute its result. For `FP`, we use a computable inductive predicate which is satisfied whenever the execution actually reaches `P`. Then, `count_rounds` is defined as a fixpoint using a structural induction over the `FP` assumption. In this function, we deal separately with the case where zero is returned: when the assumption `FP` claims that `P` is (immediately) reached, `count_rounds P e` returns 0. Otherwise, it returns the result computed by an auxiliary inductive function that requires one more parameter: the accumulator parameter that encodes the set of unsatisfied nodes. This auxiliary function computes the successive values of the set of unsatisfied nodes (as explained in Subsubsection 3.2.1) until `P` is reached. We have two cases:

- Either the assumption `FP` claims that `P` is reached, hence the auxiliary function returns one round (to count the current round).
- Or the set of unsatisfied nodes is refreshed by removing both activated and neutralized nodes. Moreover, if this new set is empty, the function starts a new round: it adds one to the current result and resets the set of unsatisfied nodes with the enabled nodes of the current configuration.

As a matter of fact, we can prove that the functional and relational definitions are related as expected. Namely, for every execution `e`, every number of rounds n, and every predicate `P`, we have

`at_most_rounds P` n `e` $\longleftrightarrow$ **`exists`** n', $n' \leq n \wedge$ `count_rounds P e =` n'.

Note that this latter property is an equivalence and implies that if `count_rounds P e =` n', then $\forall\ n \geq n'$, `at_most_rounds P` n `e`. Again, there is no need to detail the proof of this property as it is directly obtained by induction on the `FP` assumption.

3.2.4 Induction Scheme

During the development of this round library, we paid a particular attention on facilitating, as much as possible, the use of the round predicate in users' own proofs. To do so, we have developed tools to avoid coinductive proofs which are particularly tricky in Coq. Actually, the fact that an execution requires at most n rounds to reach `P` is usually proven using induction on n. In that spirit, we have developed an induction scheme that follows the classical way inductions on rounds are written in paper-and-pencil proofs.

The particular induction scheme we propose is as follows: assume that we want to prove that an execution requires at most B rounds to reach `P`. Assume also that we have a family of predicate $\texttt{P}_n$ (indexed on natural numbers n) such that $\texttt{P}_B$ implies `P`. We can prove the following lemma.

▶ **Lemma 1** (Lemma `schema_round_induction`). *Assume an execution* `e` *satisfies the following two properties:*

- `e` *satisfies* $\mathtt{P}_0$ *(Base Case).*
- *All along the execution* `e`*, and for every value $n < B$, if* `e` *has a suffix* `c` *satisfying* $\mathtt{P}_n$*, then* `c` *requires at most one round to reach* $\mathtt{P}_{n+1}$*,* i.e., `at_most_rounds` $\mathtt{P}_{n+1}$ 1 `c` *holds (Induction Step).*

Then, `e` *requires at most B rounds to reach* $\mathtt{P}_B$ *and so* `P`*.*

The proof of this induction schema is shown by induction on the parameter n. The base case is immediate from the first property on $\mathtt{P}_0$ (*Base Case*). The induction step of the proof uses the second property and is a direct application of the next lemma.

▶ **Lemma 2** (Lemma `schema_round_step`). *Let* `e` *be an execution and n be a number of rounds. Let* $\mathtt{P}_1$ *and* $\mathtt{P}_2$ *be two predicates over executions. Assume*

(A) `e` *requires at most n rounds to reach* $\mathtt{P}_1$ *and*

(B) *all along the execution* `e`*, if* `e` *has a suffix* `c` *satisfying* $\mathtt{P}_1$*, then* `c` *requires at most one round to reach* $\mathtt{P}_2$*,* i.e., `at_most_rounds` $\mathtt{P}_2$ 1 `c` *holds.*

Then, `e` *requires at most $n+1$ rounds to reach* $\mathtt{P}_2$*.*

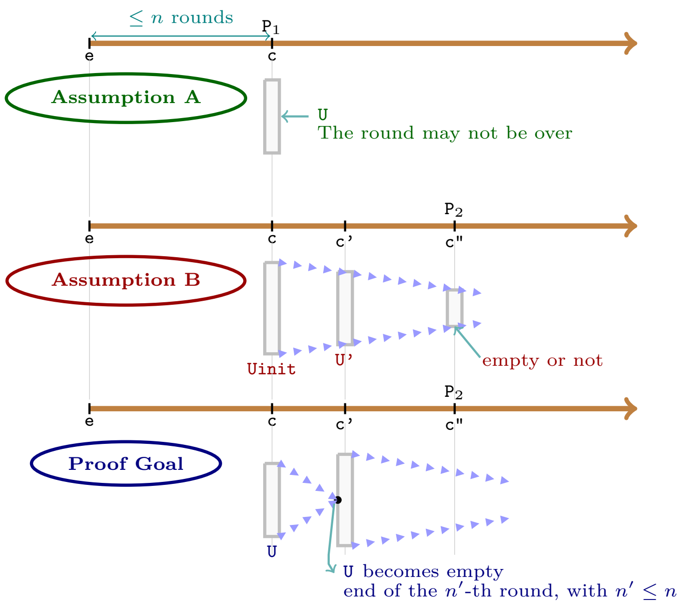


Figure 2 Proof principle of Lemma `schema_round_step`.

The key point to prove Lemma `schema_round_step` (and the fact that makes the result non-obvious) is that the set of unsatisfied nodes may change from one assumption to the other. Indeed (see Figure 2), using Assumption (A), `e` eventually reaches $\mathtt{P}_1$, *i.e.*, there is some suffix `c` of `e` where $\mathtt{P}_1$ `c` holds. Let `U` be the current set of unsatisfied nodes when it happens (for the first time). Using Assumption (B), we know that $\mathtt{P}_1$ `c` holds. Hence, `c` requires at most 1 round to reach $\mathtt{P}_2$. Now, the computation of this assertion uses `UNSAT_init (H c)` as set of unsatisfied nodes instead of `U`. To obtain that `e` requires at most one more round to reach $\mathtt{P}_2$ from its beginning, we need that `c` requires at most 1 round to reach $\mathtt{P}_2$ using `U` and not `UNSAT_init (H c)`. Obviously, there is no reason for `U` to be equal to `UNSAT_init (H c)`. Nevertheless, we can compare them: since unsatisfied nodes are enabled, `U` is included into `UNSAT_init (H c)`.

Before presenting an overview of the proof of Lemma `schema_round_step`, we have two remarks.

▶ Remark 1. Let `U` and `U'` be two sets of unsatisfied nodes such that `U` ⊆ `U'`. If `e` requires at most one round to reach `P` with the current set `U`, then this is also true with the set `U'`. Namely, if `at_most_rounds_aux P U 1 e`, then `at_most_rounds_aux P U' 1 e`.

Remark 1 is a fairly (easy to prove) intuitive property, since we can prove that the inclusion between the two families of sets – built from `U` and `U'`, respectively – remains; see Figure 3.

▶ Remark 2. Using the same notations as above, if `U` ⊆ `U'` ⊆ `UNSAT_init (H e)` and `at_most_rounds_aux P U' 1 e` holds, then `at_most_rounds_aux P U 2 e` also holds.

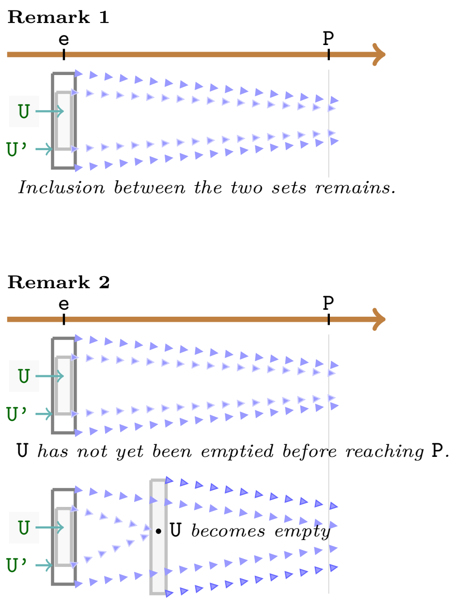

Remark 1
e
P
U
U'
Inclusion between the two sets remains.
Remark 2
e
P
U
U'
U has not yet been emptied before reaching P.
U
U'
U becomes empty

Figure 3 Principles for Remarks 1 and 2.

Proof Overview of Remark 2. (See the sketch given in Figure 3 for an illustration.) Starting the computation of rounds at `e` with `U'`:

- Either `U` does not become empty (and so neither do `U'`) before reaching `P`, hence using `U` as set of unsatisfied nodes, `P` is reached within at most one round of `e`.
- Or `U` becomes empty before reaching `P`, say at Configuration g_c. So, using `U` as set of unsatisfied nodes, one round of `e` is over and `U` is refilled. Let $\texttt{U}_{init}$ be the value of `U` after being refilled at g_c. Let $\texttt{U'}_c$ be the value of `U'` at g_c. As `U'` was included in `UNSAT_init (H e)`, it still contains enabled nodes only. So, $\texttt{U'}_c$ is included into $\texttt{U}_{init}$. We can then apply Remark 1: since `P` is reached from g_c using $\texttt{U'}_c$ in at most one round, this is also the case from g_c using $\texttt{U}_{init}$. Overall, this means that using `U` as set of unsatisfied nodes, `P` is reached within at most two rounds in `e`. ◀

We can now conclude with the proof of Lemma `schema_round_step`.

Proof Overview of Lemma `schema_round_step`. The proof of Lemma `schema_round_step` uses coinduction. For the sake of explanation, we summarize the proof using the following two scenarios.

First scenario. if `e` contains less than $n+1$ rounds, then the result immediately holds.

Second scenario. Assume `e` contains at least $n+1$ rounds (*n.b.*, the $(n+1)$-th may be infinite). Then, `e` contains at least n rounds and the first n rounds of `e` are finite. By Assumption (A), `e` actually reaches $\texttt{P}_1$ within at most n rounds. So, `e` consumes at most n rounds to reach $\texttt{P}_1$ at the first configuration of some suffix `c`. Let `U` be the set of unsatisfied nodes at `H c`.

Assumption (B) ensures that in `c`, $\texttt{P}_2$ is actually reached in at most 1 round with set of unsatisfied nodes `UNSAT_init (H c)`. Then, we have two cases. If the n^{th} round terminates at `H c`, `U` $=$ `UNSAT_init (H c)` and we are done. Otherwise, at most $n' < n$ rounds have terminated at Configuration `H c`. Now, since `U` only contains nodes that are enabled in `H c`, we have `U` $\subseteq$ `UNSAT_init (H c)` and we can apply Remark 2 with `U' = UNSAT_init (H c)`. So, with `U` as set of unsatisfied nodes, $\texttt{P}_2$ is reached within at most two more rounds, and we are done. ◀

4 Round Complexity of the Algorithm

We now illustrate how to use the previous tools by sketching the certification of the stabilization time in rounds of Algorithm $\mathcal{BFS}$. Precisely, we show that $\mathcal{BFS}$ requires at most $\mathcal{D}+2$ rounds to reach a terminal configuration starting from an arbitrary configuration. The full certified proof is detailed in Appendix B. Here, we focus on generic formal tools and show how to apply them. In particular, through out the section, we will introduce two additional useful general tools.

Another goal of this section is to convince the reader that our formalization allows to write certified proofs that are close to the standard usages in the self-stabilizing community. In that spirit, the Coq proof outline given below broadly follows the approach proposed in [4].

The proof is split into the following two main parts. First, we prove (Part A) that $\mathcal{BFS}$ requires at most $\mathcal{D}+1$ rounds to reach a configuration from which the d-variables are correctly assigned forever, *i.e.*, for every node p, $p.d$ is (forever) equal to the distance from p to the root. Then, we prove that once d-variables are correctly assigned forever, $\mathcal{BFS}$ requires at most one more round to reach a terminal configuration (Part B).

To prove Part A (and according to [4]) we use the induction scheme given in **Lemma** `schema_round_induction` with the predicate P_k `e := Always (check_dist` k`) e` and the value B `:=` $\mathcal{D}+1$, where Predicate `check_dist` k `e` holds iff every node p satisfies the condition `CD` k `e` p. This latter condition checks that `(dist` p r `<` k $\wedge$ $p.d$ `= dist` p r`)` $\vee$ `(dist` p r $\geq$ k $\wedge$ $p.d$ $\geq$ k`)` holds in `H e`, the first configuration of the execution `e`.

To apply **Lemma** `schema_round_induction`, we have to establish the assumptions of this lemma, namely the *Base Case* and the *Induction Step*. To ease this proof, we have introduced two other generic results in the PADEC Round library. The goal of the first one is to simplify the proof when dealing with local predicates such as `check_dist` and P_k. Actually, P_k checks properties that are local at each node. Precisely, it checks that every node p satisfies the local property `CD` k `e` p all along `e`. For such a local property `Q: Node` $\to$ `Exec` $\to$ **Prop**, we can prove that

```
at_most_rounds (Always (fun e' => ∀p, Q p e')) k e ⟷
∀p, at_most_rounds (Always (Q p)) k e
```

Namely, the universal quantifier over the nodes can be shifted to the outer border of the formula, which is easier to handle, since the proof can now be done using a single node. This result, proven by a simple coinduction, is now provided in the PADEC Round library.

The second generic result is the following proof scheme:

▶ **Lemma 3** (Lemma `at_most_rounds_scheme_per_node`). *Let* `P: Node` $\rightarrow$ `Exec` $\rightarrow$ **Prop** *be a predicate and* p *be a node. If*

- *either* `(P` p `e)` *holds when* p *is disabled in* `(H e)`
- *or* `(P` p`)` *becomes true whenever* p *is activated or neutralized during some step of* `e`*,*

then at most one round is required to reach a configuration where `(P` p`)` *holds.*

Indeed, either the node p is disabled at the beginning of the round and then `P` p holds (due to the first assumption); or p is enabled and so belongs to the set of unsatisfied nodes. Now, as this set is empty at the end of the round, the node has been removed because it has been activated or neutralized meanwhile and so `P` p has become true during the round (second assumption). Overall, we obtain that `P` p requires at most one round to be reached.

Using these tools, we have obtained Part A by proving that for every execution `e`,

- the *Base Case* holds, namely, the property P_0 `e := Always (check_dist 0) e` is satisfied;
- and the *Induction Step* also holds, *i.e.*, for every value of $k < \mathcal{D} + 1$, if `e` has a suffix `c` satisfying P_k, then `c` requires at most one round to reach P_{k+1}.

The detailed proof (which is based on a combinatorial study of possible values for the d-variables) is given in Appendix B. Notice however that, by successfully applying our generic proof schemes, we were able to make the certification of the main proof simpler and really close to the one in [4].

Part B and Final Result. Part A proves that Algorithm $\mathcal{BFS}$ requires at most $\mathcal{D} + 1$ rounds to reach a configuration from which `Always (check_dist` $(\mathcal{D}+1)$`) e` is achieved (*i.e.*, the d-variables are correctly assigned forever). After that, at most one more round is required to reach a configuration where the par-variables are correctly set and so to achieve the silence. The proof (Part B) uses the same mechanism as before and mainly relies on the following two simple facts:

- As `check_dist` $(\mathcal{D}+1)$ `e` holds forever, the d-variables no more change.
- Furthermore, when `check_dist` $(\mathcal{D}+1)$ `e` holds, a node can be activated only once (using Action CP): the action, and so the node too, is then disabled forever.

Afterwards, we have merged Parts A and B using Lemma `schema_round_step` to conclude that every execution `e` requires at most $\mathcal{D} + 2$ rounds to reach a terminal configuration, concluding then the proof of the round complexity for Algorithm $\mathcal{BFS}$.

To complete this result, we have also proven that Algorithm $\mathcal{BFS}$ is silent and self-stabilizing w.r.t. its specification. For the convergence property, we can easily show, using the definitions of `weakly_fair` and `at_most_rounds`, that if an execution has been scheduled using a weakly fair daemon and requires at most B rounds to converge to some property `P`, then the execution eventually reaches `P`, *i.e.*, the execution converges to `P` within finite time. Thus, we can deduce that under the weakly fair daemon, Algorithm $\mathcal{BFS}$ converges to a legitimate configuration. The rest of the proof, *i.e.*, Algorithm $\mathcal{BFS}$ satisfies both the specification and closure part of the self-stabilization property, is proven by induction; see Appendix A for details.

Overall, we have certified the following theorem:

▶ **Theorem 4.** *Let* G *be a connected bidirectional network rooted at some node* r*. Under a weakly fair daemon,* $\mathcal{BFS}$ *is a silent self-stabilizing BFS spanning tree construction whose stabilization time is at most* $\mathcal{D} + 2$ *rounds, where* $\mathcal{D}$ *is the diameter of* G*.*

5 Conclusion

Certification is an important tool to increase confidence of algorithmic designers in the correctness of their solutions. This is even more important in fault-tolerant distributed algorithmic, where models, algorithms, and intended specifications are most of the time both complex and subtle. In this context, the PADEC library has been proposed to help the certification (in Coq) of self-stabilizing distributed algorithms written in the atomic-state model. This library encompasses all necessary formal tools to establish the correctness (especially the convergence) and the time complexity of monolithic, as well as composite, self-stabilizing algorithms. The usefulness of all these formal tools has been validated thanks to many non-trivial use cases from the literature. The last contribution, presented here, has been to import in PADEC the most commonly used time complexity measure in the self-stabilizing area: the rounds. The main encountered difficulties were due to the non-atomic nature of the rounds that made them not compositional. The definition of rounds has been provided in PADEC together with many formal companion tools, *e.g.*, `Lemmas schema_round_induction`, `schema_round_step`, `at_most_rounds_scheme_per_node`. The suitability of these general tools has been demonstrated with an appropriate use case from the literature: we have certified the stabilization time of Dolev *et al*'s algorithm [22]. Although the intrinsic nature of rounds implies a coinductive definition, the companion tools provided in the library avoid the user to deal with coinductive proofs, which may be tricky in Coq. Our use case is convincing in this sense since applying the companion tools allows to prevent the use of coinduction in its certified proof. Actually, the only (rather simple) proofs requiring coinduction are due to the `Always (check_dist` *k*`)` property which is itself coinductive.

References

1 Karine Altisen, Pierre Corbineau, and Stéphane Devismes. A framework for certified self-stabilization. *Log. Methods Comput. Sci.*, 13(4), 2017. `doi:10.23638/LMCS-13(4:14)2017`.

2 Karine Altisen, Pierre Corbineau, and Stéphane Devismes. Certification of an exact worst-case self-stabilization time. In *ICDCN '21: International Conference on Distributed Computing and Networking, Virtual Event, Nara, Japan, January 5-8, 2021*, pages 46–55. ACM, 2021. `doi:10.1145/3427796.3427832`.

3 Karine Altisen, Alain Cournier, Stéphane Devismes, Anaïs Durand, and Franck Petit. Self-stabilizing leader election in polynomial steps. *Inf. Comput.*, 254:330–366, 2017. `doi:10.1016/j.ic.2016.09.002`.

4 Karine Altisen, Stéphane Devismes, Swan Dubois, and Franck Petit. *Introduction to Distributed Self-Stabilizing Algorithms*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2019. `doi:10.2200/S00908ED1V01Y201903DCT015`.

5 Cédric Auger, Zohir Bouzid, Pierre Courtieu, Sébastien Tixeuil, and Xavier Urbain. Certified impossibility results for byzantine-tolerant mobile robots. In Teruo Higashino, Yoshiaki Katayama, Toshimitsu Masuzawa, Maria Potop-Butucaru, and Masafumi Yamashita, editors, *Stabilization, Safety, and Security of Distributed Systems - 15th International Symposium, SSS 2013, Osaka, Japan, November 13-16, 2013. Proceedings*, volume 8255 of *Lecture Notes in Computer Science*, pages 178–190. Springer, 2013. `doi:10.1007/978-3-319-03089-0_13`.

6 Cédric Auger, Zohir Bouzid, Pierre Courtieu, Sébastien Tixeuil, and Xavier Urbain. Certified impossibility results for byzantine-tolerant mobile robots. In Teruo Higashino, Yoshiaki Katayama, Toshimitsu Masuzawa, Maria Potop-Butucaru, and Masafumi Yamashita, editors, *Stabilization, Safety, and Security of Distributed Systems - 15th International Symposium, SSS 2013, Osaka, Japan, November 13-16, 2013. Proceedings*, volume 8255 of *Lecture Notes in Computer Science*, pages 178–190. Springer, 2013.

7 Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

8 Frédéric Blanqui and Adam Koprowski. Color: a coq library on well-founded rewrite relations and its application to the automated verification of termination certificates. *Mathematical Structures in Computer Science*, 21(4):827–859, 2011. doi:10.1017/S0960129511000120.

9 Roderick Bloem, Nicolas Braud-Santoni, and Swen Jacobs. Synthesis of self-stabilising and byzantine-resilient distributed systems. In *Computer Aided Verification - 28th International Conference, CAV 2016*, pages 157–176, 2016.

10 Pierre Castéran and Vincent Filou. Tasks, types and tactics for local computation systems. *Stud. Inform. Univ.*, 9(1):39–86, 2011.

11 Bernadette Charron-Bost, Henri Debrat, and Stephan Merz. Formal verification of consensus algorithms tolerating malicious faults. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 120–134, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

12 Bernadette Charron-Bost and Stephan Merz. Formal verification of a consensus algorithm in the heard-of model. *Int. J. Software and Informatics*, 3(2-3):273–303, 2009.

13 Alain Cournier, Ajoy K. Datta, Franck Petit, and Vincent Villain. Snap-Stabilizing PIF Algorithm in Arbitrary Networks. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, pages 199–206, 2002.

14 Pierre Courtieu. Proving self-stabilization with a proof assistant. In *16th International Parallel and Distributed Processing Symposium (IPDPS 2002), 15-19 April 2002, Fort Lauderdale, FL, USA, CD-ROM/Abstracts Proceedings*, volume 1, page 8pp. IEEE Computer Society, 2002.

15 Pierre Courtieu, Lionel Rieg, Sébastien Tixeuil, and Xavier Urbain. Impossibility of gathering, a certification. *Inf. Process. Lett.*, 115(3):447–452, 2015.

16 Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernán Vanzetto. TLA + proofs. In *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, pages 147–154, 2012.

17 Ajoy Kumar Datta, Stéphane Devismes, Karel Heurtefeux, Lawrence L. Larmore, and Yvan Rivierre. Competitive self-stabilizing k-clustering. *Theor. Comput. Sci.*, 626:110–133, 2016. doi:10.1016/j.tcs.2016.02.010.

18 Carole Delporte-Gallet, Hugues Fauconnier, Eli Gafni, and Leslie Lamport. Adaptive register allocation with a linear number of registers. In *Distributed Computing - 27th International Symposium, DISC 2013*, 2013.

19 Stéphane Devismes and Colette Johnen. Silent self-stabilizing BFS tree algorithms revisited. *J. Parallel Distributed Comput.*, 97:11–23, 2016. doi:10.1016/j.jpdc.2016.06.003.

20 Stéphane Devismes, Anissa Lamani, Franck Petit, Pascal Raymond, and Sébastien Tixeuil. Optimal grid exploration by asynchronous oblivious robots. In *Stabilization, Safety, and Security of Distributed Systems - 14th International Symposium, SSS 2012, Toronto, Canada, October 1-4, 2012. Proceedings*, volume 7596 of *Lecture Notes in Computer Science*, pages 64–76. Springer, 2012.

21 Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974. doi:10.1145/361179.361202.

22 Shlomi Dolev, Amos Israeli, and Shlomo Moran. Self-stabilization of dynamic systems assuming only Read/Write atomicity. *Distributed Computing*, 7(1):3–16, 1993.

23 Fathiyeh Faghih, Borzoo Bonakdarpour, Sébastien Tixeuil, and Sandeep S. Kulkarni. Specification-based synthesis of distributed self-stabilizing protocols. In *Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016*, 2016.

24 Lin Fei, Sun Yong, Ding Hong, and Ren Yizhi. Self stabilizing distributed transactional memory model and algorithms. *Journal of Computer Research and Development*, 51(9):2046, 2014.

25 Wan Fokkink, Jaap-Henk Hoepman, and Jun Pang. A note on k-state self-stabilization in a ring with k=n. *Nordic Journal of Computing*, 12(1):18–26, 2005.

26 Wim H. Hesselink. Mechanical verification of lamport's bakery algorithm. *Science of Computer Programming*, 78(9):1622–1638, 2013.

27 Shing-Tsaan Huang and Nian-Shing Chen. A self-stabilizing algorithm for constructing breadth-first trees. *Information Processing Letters*, 41(2):109–117, 1992.

28 Mauro Jaskelioff and Stephan Merz. Proving the correctness of disk paxos. *Archive of Formal Proofs*, 2005, 2005.

29 Philipp Küfner, Uwe Nestmann, and Christina Rickmann. Formal verification of distributed algorithms - from pseudo code to checked proofs. In Jos C. M. Baeten, Thomas Ball, and Frank S. de Boer, editors, *Theoretical Computer Science - 7th IFIP TC 1/WG 2.2 International Conference, TCS 2012, Amsterdam, The Netherlands, September 26-28, 2012. Proceedings*, volume 7604 of *Lecture Notes in Computer Science*, pages 209–224, 2012.

30 S. S. Kulkarni, J. Rushby, and N. Shankar. A case-study in component-based mechanical verification of fault-tolerant programs. In *19th IEEE International Conference on Distributed Computing Systems*, pages 33–40, 1999.

31 Marta Kwiatkowska, Gethin Norman, and David Parker. Probabilistic verification of herman's self-stabilisation algorithm. *Form. Asp. Comput.*, 24(4–6):661–670, July 2012. `doi:10.1007/s00165-012-0227-6`.

32 Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

33 Stephan Merz. On the verification of a self-stabilizing algorithm. Technical report, University of Munich, 1998.

34 Lawrence C. Paulson. Natural deduction as higher-order resolution. *J. Log. Program.*, 3(3):237–258, 1986. `doi:10.1016/0743-1066(86)90015-4`.

35 S. Qadeer and N. Shankar. Verifying a self-stabilizing mutual exclusion algorithm. In *International Conference on Programming Concepts and Methods (PROCOMET '98) 8–12 June 1998, Shelter Island, New York, USA*, pages 424–443, Boston, MA, 1998. Springer US.

36 Vincent Rahli, David Guaspari, Mark Bickford, and Robert L. Constable. Formal specification, verification, and implementation of fault-tolerant systems using eventml. *ECEASST*, 72, 2015.

37 Vincent Rahli, David Guaspari, Mark Bickford, and Robert L. Constable. Eventml: Specification, verification, and implementation of crash-tolerant state machine replication systems. *Sci. Comput. Program.*, 148:26–48, 2017.

38 Vincent Rahli, Ivana Vukotic, Marcus Völp, and Paulo Jorge Esteves Veríssimo. Velisarios: Byzantine fault-tolerant protocols powered by coq. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, pages 619–650, 2018.

39 Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2 edition, 2000. `doi:10.1017/CBO9781139168724`.

40 The Coq Development Team. *The Coq Proof Assistant Documentation*, June 2012. URL: `http://coq.inria.fr/refman/`.

41 Tatsuhiro Tsuchiya, Shin'ichi Nagano, Rohayu Bt Paidi, and Tohru Kikuno. Symbolic model checking for self-stabilizing algorithms. *IEEE TPDS*, 12(1):81–95, 2001. `doi:10.1109/71.899941`.

A Specification and Closure of Algorithm $\mathcal{BFS}$

Using the definitions given in Section 2.2, we now sketch the proof of the specification part of the self-stabilization of Algorithm $\mathcal{BFS}$. Precisely, we have to prove that the specification of the algorithm is satisfied in any terminal configuration, *i.e.*, the *par*-variables of non-root nodes shape a BFS tree rooted at r that spans the graph `R_Net`. The proof actually follows the one given in [4] for a bounded-memory version of $\mathcal{BFS}$.

Consider a terminal configuration `g`. For a node `p`, `g p` provides the local state of `p` in configuration `g`. Furthermore, let `d (g p)` (resp. `par (g p)`) be the value of `p.`d (resp. `p.`par) in `g`. The proof begins by establishing that in `g`, d-variables are not underestimated:

$$\forall \texttt{ p, d (g p)} \geq \texttt{dist p } r$$

The proof is a simple case analysis; see below.

Proof Overview. Let `p` be a node. If we have `d (g p)` $\geq$ `dist p` r then we are done. Otherwise, `dist p` r `> d (g p)` and pick a node `pmin` satisfying this inequality with the smallest d-value (we can access `pmin` by filtering the list of all nodes with the *ad hoc* criteria). Then, we can prove that every neighbor `q` of `pmin` satisfies `d (g pmin) < 1 + d (g q))`.

- Indeed, if `dist q` r $\leq$ `d (g q)`, then we are done since `dist pmin` r $\leq$ `1 + dist q` r (by definition) and `d (g pmin) < dist pmin` r (by hypothesis).
- Otherwise, `dist q` r `> d (g q)`. But, in this case, `d (g pmin)` $\leq$ `d (g q)` since `pmin` is a node with minimum `d`-value among the nodes satisfying the inequality, and we are done.

As `pmin` is not the root (indeed `dist pmin` r `> d (g pmin)` $\geq$ `0`), Action CD is enabled at `pmin`: indeed, `pmin.d` is not equal to `q.d + 1` for some neighbor `q`, since for any of them, we have proved that `d (g pmin) < 1 + d (g q)`. Overall, this proves that `pmin` is enabled, which contradicts the fact that `g` is terminal. ◀

We can now show that for every node p, its d-variable in g (*i.e.*, `d(g p)`) is actually equal to its distance to the root (*i.e.*, `dist g p`).

Proof Overview. The proof is done by induction on the distance from nodes to the root.

- *Base case:* Let `p` be a node such that `dist p` r `= 0`. Then, `p` is the root and as r is disabled, we have $r.d$ `= 0`.
- *Step case:* Let `d` $\geq$ `0`. Assume the property is satisfied by every node at distance `d` from the root. Let `q` be a node at distance `d + 1` from the root. Obviously, `q` is not the root. Then, by definition of `dist`, `q` has a neighbor, say `p`, at distance `d` from the root. By induction assumption, `d (g p) = d`. As `q` is disabled, we just have to prove that $Dist_q$ `= d + 1`, *i.e.*, `d + 1` is the minimum value in the list `{ x.d + 1, x in q.neighbors }`. Now, `p` is a neighbor of `q` that satisfies `d + 1 = d (g p) + 1`. Moreover, for every other neighbor `p'` of `q`, we have `d + 1` $\leq$ `d (g p') + 1`. Indeed, by definition, `p'` is at distance `d`, `d + 1`, or `d + 2` from the root and we have seen that `d (g p')` is not underestimated, *i.e.*, `d (g p')` $\geq$ `dist p'` r $\geq$ `d`. Hence, we obtain that $Dist_q$ `= d + 1 = d (g q)`, and we are done. ◀

Using Action CP and the fact that d-variables are correctly evaluated in the terminal configuration `g`, we now show that the par-variables define a BFS spanning tree rooted at r in `g`. To that goal, we first define `Par_Rel g n n'` as the relation describing the spanning tree in `g`: `Par_Rel g n n'` holds iff the node `n` is not the root and `par (g n)` is the channel that leads to the node `n'`. By definition of the algorithm and since `g` is terminal, for every non-root node `p`, `Par_Rel g p q` holds for some node `q` such that (p,q) is an edge in `R_Net` and `d (g p) = d (g q) + 1` (remember that these values are also the distances from the nodes to r, hence `dist p` r `= dist q` r `+ 1`). Therefore, we have the following properties:

- `Par_Rel g` is a subgraph of `R_Net`.
- r has no link to some other node using `Par_Rel g`.
- `Par_Rel g` does not contain any cycle, hence it is a DAG.
  Indeed, along any path of `Par_Rel g`, the distances from nodes to the root decreases.
- By definition of `Par_Rel g`, every node has a single parent.

- There is a path from any node `p` to the root in `Par_Rel g` and the length of this path is exactly `dist p` r.
  Notice that this latter property requires to explicitly build the witness path. So, we prove that for every node at distance `d` from the root, there exists a path of length `d` from this node to root in `Par_Rel g`. The proof is done by induction on `d`.
  - The base case (for root node) is trivial.
  - Assume that the property holds for some `d` $\geq$ `0` and consider a node `p` at distance `d + 1` from the root. The parent of `p` using `Par_Rel g`, say `q`, is at distance `d` to the root. Hence, we can apply the induction hypothesis to `q` and then add the edge from `p` to `q` to the path to obtain a path from `p` to r which exists in `Par_Rel g`.

Based on the previous properties, `Par_Rel g` is BFS spanning tree rooted at r. In particular, the distances to the root in `Par_Rel g` are exactly those in `R_Net`. To show this latter fact, we use the last property (there exists a path from every node to root in `Par_Rel g` whose length is the distance to the root) and the fact that the path between any two nodes is unique in a tree (`Path_Rel g` is a tree since it is a DAG with single parent links at each non-root nodes).

Hence the specification of the problem holds in any terminal configuration: the relation `Par_Rel g` (built from the variables computed at each node) is a BFS spanning tree of `R_Net` rooted at r. This concludes the *specification* part of the proof of self-stabilization. Indeed, recall that the set of legitimate configurations is actually the set of terminal configurations.

Finally, since terminal configurations are closed by definition, the *closure* part of the proof is trivially satisfied.

B Detailed Proof of the Round Complexity of Algorithm $\mathcal{BFS}$

We detail here the proof that Algorithm $\mathcal{BFS}$ requires at most $\mathcal{D} + 2$ rounds to reach a terminal configuration, starting from an arbitrary configuration. Under the weakly fair daemon, this property implies convergence. Hence, thanks to results of Appendix A, we can conclude that $\mathcal{BFS}$ is self-stabilizing under the weakly fair daemon and its stabilization time is at most $\mathcal{D} + 2$ rounds.

The proof is split into the following two main parts:

Part A: First, we prove that $\mathcal{BFS}$ requires at most $\mathcal{D} + 1$ rounds to reach a configuration from which the d-variables are correctly assigned forever, *i.e.*, for every node p, $p.d$ is forever equal to the distance from p to the root (**Theorem** `BFS_rounds_CD` in the certified proof).

Part B: Then, we prove that once d-variables are correctly assigned forever, $\mathcal{BFS}$ requires at most one more round to reach a terminal configuration (**Lemma** `last_round_action_CP` in the certified proof).

B.1 Part A

First, we recall the definition of Predicate `check_dist` k `e`, where k is a natural number and `e` is an execution. Following [4], this predicate holds iff for every node p, one of the following two conditions is satisfied in `H e`:

(a) either `CD_a` k `e` p `:= dist` p r `<` k $\wedge$ $p.d$ `= dist` p r,

(b) or `CD_b` k `e` p `:= dist` p r $\geq$ k $\wedge$ $p.d$ $\geq$ k.

Then, we have the following straightforward, yet useful, properties:

1. By definition, `check_dist 0 e` holds, for every execution `e`.

2. `check_dist` $(\mathcal{D}+1)$ `e` holds iff the d-variables in `H e` are correctly assigned.
   Indeed, Case (b) of the definition does not apply in this case.
3. We can easily prove, by checking the rules of the algorithm, that for every execution `e` and every `k`, `check_dist` k `e` is suffix-closed, meaning that once it is satisfied, it holds forever in `e`.
   More formally, `check_dist` k `e` implies that `check_dist` k `c` holds for every suffix `c` of `e`.

We now use the induction scheme given in **Lemma** `schema_round_induction` with the predicate P_k `e := Always (check_dist` k`) e` and the value B `:=` $\mathcal{D}+1$ to prove that any execution `e` requires at most $\mathcal{D}$+1 rounds to reach a configuration where P_B `:= Always (check_dist(`$\mathcal{D}+1$`)) e` holds, *i.e.*, a configuration from which the d-variables are forever correctly assigned. To apply **Lemma** `schema_round_induction`, we have to establish the assumptions of this lemma, namely the *Base Case* and the *Induction Step*. We now consider an arbitrary execution `e` and detail the proof of these two goals.

B.1.1 Base Case

The base case, P_0 `:= Always (check_dist 0)`, is trivial: indeed, `check_dist 0 e` holds, by Property (1), and then we can conclude, by Property (3).

B.1.2 Induction Step

We have to prove that all along the execution `e`, and for every value of $k < \mathcal{D}+1$, if `e` has a suffix `c` satisfying P_k, then `c` requires at most one round to reach P_{k+1}.

First, remark that P_k actually checks local properties at each node since it checks that for every node p, p satisfies all along `e` the local property `CD_a` k `e` $\vee$ `CD_b` k `e`. We use the generic tool to transform the predicate and place the universal quantifier over nodes at the outer border of the formula: we obtain that for any execution `e`, the fact that `e` achieves P_k in at most n rounds is equivalent to the fact that for every node p, `e` reaches in at most n rounds a configuration from which the local property `CD_a` k `e` p $\vee$ `CD_b` k `e` p is satisfied forever. We now consider any node p and split the proof depending on whether or not k is null. In turns, each case is subdivided in two subcases that separately prove `CD_a` $(k+1)$ or `CD_b` $(k+1)$ for p.

Case $k = 0$. We have to prove that at most one more round is required so that `check_dist 1` becomes true. Then again, Property (3) allows to conclude.

(a) *Proof for* `CD_a 1`. Here, by definition of `CD_a`, p can be nothing but the root since the only node at distance less than one from the root is the root itself. Hence, we must prove that at most one round is required so that $r.d$ becomes 0. To that goal, we apply **Lemma** `at_most_rounds_scheme_per_node` and conclude that at most one round is required so that the d-variable of the root becomes 0. Indeed, the assumptions of the lemma are satisfied since:
- if r is disabled, then $r.d = 0$, and
- $r.d$ becomes 0 when r is activated or neutralized (*n.b.*, this latter disjunction is equivalent to "r is activated" since r cannot be neutralized).

(b) *Proof of* `CD_b 1`. We must prove that at most one round is required for every non-root node to have a positive d-variable. Indeed, this case concerns nodes at positive distance from the root, by definition of `CD_b 1`. Again, we can apply **Lemma** `at_most_rounds_scheme_per_node`. Indeed,

- if p is disabled in (`H e`), then $p.d = Dist_p > 0$; and
- when p is activated (resp. neutralized), $p.d$ is set (resp. becomes equal) to $Dist_p > 0$.

Case $1 \leq k < \mathcal{D} + 1$. We assume that `check_dist` k `e` holds and use the same mechanisms as previously to prove that at most one round is required to reach a configuration where `CD_a` $(k+1)$ or `CD_b` $(k+1)$ holds for p. The conclusion will be that at most one more round is required to reach `Always (check_dist` $(k+1)$`)`.

(a) *Proof of* `CD_a` $(k+1)$. Here we assume that `dist` p r $< k+1$. If `dist` p r $< k$, then by applying the induction hypothesis (`check_dist` k `e` holds), we get that $p.d$ = `dist` p r forever. Otherwise `dist` p r = k and we apply **Lemma** `at_most_rounds_scheme_per_node` again:
  - If p is disabled in (`H e`), then $p.d = Dist_p = p.par.d + 1$. Then, as `dist` p r = k, p has a neighbor, q, at distance $k-1$ from the root. Using the induction hypothesis, we deduce that $q.d = k - 1$. Hence $p.d \leq k$, by definition of $Dist_p$. Consider now the node pointed by $p.par$. As it is a neighbor of p, it is at distance $k-1$, k, or $k+1$ from the root. Cases k and $k+1$ are impossible. Indeed, using the induction hypothesis, we would get that $p.par.d \geq k$ and so $p.d \geq k+1$, a contradiction.
    So, `dist` $p.par$ r = $k-1$. In this case, $p.par.d = k-1$, by induction hypothesis, and so $p.d = k$.
  - When p is activated (resp. neutralized), $p.d$ is set to (resp. becomes) $Dist_p$. So, we have to show that $Dist_p$ = k, *i.e.*, $k-1$ is the smallest value in the d-variables of p's neighbors. Now, as the distance from p to r is k, every neighbor q of p is at distance $k-1$, k, or $k+1$ from the root. By applying the induction hypothesis, we obtain that the neighbors at distance greater than $k-1$ from the root have their d-variables greater than $k-1$; moreover, those at distance $k-1$ from the root have their d-variable equal to $k-1$. As p has at least one neighbor at distance $k-1$ from the root, we obtain that $p.d$ = $Dist_p$ = k = `dist` p r.

(b) *Proof of* `CD_b` $(k+1)$. Here we assume that `dist` p r $\geq k+1$. In this case, every neighbor q of p is at distance at least k from r. Hence, by induction hypothesis, we obtain that $d.q \geq k$. Using this property and the definition of $Dist_p$, we can easily show that the two conditions of **Lemma** `at_most_rounds_scheme_per_node` are fulfilled to establish that at most one more round is required to reach a configuration where the property $p.d \geq k+1$ is satisfied.

B.2 Part B

After `check_dist` $(\mathcal{D}+1)$ `e` is achieved (*i.e.*, once the d-variables are correctly assigned forever), at most one more round is required to reach a configuration where the par-variables are correctly set and so to achieve the silence. The proof uses the same mechanism as before and mainly relies on the following two simple facts:

- As `check_dist` $(\mathcal{D}+1)$ `e` holds forever, the d-variables no more change.
- Furthermore, when `check_dist` $(\mathcal{D}+1)$ `e` holds, a node can be activated only once (using Action CP): the action, and so the node too, is then disabled forever.

B.3 Final Result

Afterwards, Parts A and B are merged using Lemma `schema_round_step` to conclude that every execution `e` requires at most $\mathcal{D}+2$ rounds to reach a terminal configuration. This conclude the proof of the round complexity for Algorithm $\mathcal{BFS}$.

Network Agnostic Perfectly Secure MPC Against General Adversaries

Ananya Appan[1]
University of Illinois at Urbana Champaign, USA

Anirudh Chandramouli[1]
Bar-Ilan University, Ramat Gan, Israel

Ashish Choudhury
International Institute of Information Technology, Bangalore, India

Abstract

In this work, we study *perfectly-secure multi-party computation* (MPC) against general (*non-threshold*) adversaries. Known protocols are secure against $\mathcal{Q}^{(3)}$ and $\mathcal{Q}^{(4)}$ adversary structures in a synchronous and an asynchronous network respectively. We address the existence of a *single* protocol which remains secure against $\mathcal{Q}^{(3)}$ and $\mathcal{Q}^{(4)}$ adversary structures in a *synchronous* and in an *asynchronous* network respectively, where the parties are *unaware* of the network type. We design the *first* such protocol against general adversaries. Our result generalizes the result of Appan, Chandramouli and Choudhury (PODC 2022), which presents such a protocol against *threshold* adversaries.

2012 ACM Subject Classification Security and privacy → Information-theoretic techniques; Theory of computation → Distributed algorithms; Theory of computation → Cryptographic protocols; Theory of computation → Communication complexity

Keywords and phrases Verifiable Secret Sharing, Byzantine Agreement, Perfect Security

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.3

Related Version *Full Version*: `https://arxiv.org/abs/2208.06223`

Funding *Anirudh Chandramouli*: Supported by the Israel Science Foundation (grant No. 2439/20).
Ashish Choudhury: This research is an outcome of the R&D work undertaken in the project under the Visvesvaraya PhD Scheme of Ministry of Electronics & Information Technology, Government of India, being implemented by Digital India Corporation (formerly Media Lab Asia). The author is also thankful to the Electronics, IT & BT Government of Karnataka for supporting this work under the CIET project.

1 Introduction

Secure *multi-party computation* (MPC) [40, 27, 10] is one of the central pillars in modern cryptography. Informally, an MPC protocol allows a set of mutually distrusting parties, $\mathcal{P} = \{P_1, \ldots, P_n\}$, to securely perform any computation over their private inputs *without* revealing anything additional about their inputs. In any MPC protocol, the distrust is modelled by a centralized *adversary* $\mathcal{A}$, who can corrupt and control a subset of the parties during the protocol execution. We aim for *perfect security*, where $\mathcal{A}$ is a *computationally unbounded* byzantine adversary who can force the corrupt parties to behave *arbitrarily* during protocol execution and where all security guarantees are achieved *without* any error.

[1] Work done as a student at IIIT Bangalore

Traditionally, the *corruption capacity* of $\mathcal{A}$ is modelled through a publicly-known *threshold* t, where it is assumed that $\mathcal{A}$ can corrupt *any* subset of up to t parties [10, 15, 38]. A more generic and fine-grained form of corruption capacity is the *general-adversary* model (also known as the *non-threshold* setting) [28]. Here, $\mathcal{A}$ is characterized by a publicly-known monotone *adversary structure* $\mathcal{Z} \subset 2^{\mathcal{P}}$, which enumerates *all possible* subsets of potentially corrupt parties, where $\mathcal{A}$ can select any one subset from $\mathcal{Z}$ for corruption. Notice that a *threshold* adversary is a *special* type of *non-threshold* adversary, where $\mathcal{Z}$ consists of all subsets of $\mathcal{P}$ of size up to t. It is well-known that modelling $\mathcal{A}$ through $\mathcal{Z}$ allows for more flexibility, especially when $\mathcal{P}$ is small [28, 29]. The downside is that the complexity of the resultant protocols is polynomial in the size of $\mathcal{Z}$, which could be exponential in n in the worst case.

Traditionally, MPC protocols are designed assuming either a *synchronous* or *asynchronous* communication model. In a *synchronous* MPC (SMPC) protocol, the communication channels between the parties are assumed to be *synchronized*, and every message is assumed to be delivered within some *known* time Δ. Unfortunately, maintaining such time-outs in real-world networks like the Internet is extremely challenging. Asynchronous MPC (AMPC) protocols operate assuming an *asynchronous* communication network with eventual message delivery, where the messages can be arbitrarily, yet finitely delayed. Designing AMPC protocols is inherently *more challenging* when compared to SMPC protocols. This is because, due to the lack of an upper bound on message delays, parties won't know how long to wait for an expected message, since the corresponding sender party may be *corrupt* and may not send the message in the first place. Consequently, to avoid an endless wait, a party can consider messages from only a subset of parties for processing but, in the process, messages from potentially slow but honest parties may get ignored. In fact, in *any* AMPC protocol, it is *impossible* to ensure that the inputs of *all* honest parties are considered for computation, since the wait may turn out to be endless.

Against *threshold* adversaries, perfectly-secure SMPC and AMPC can tolerate up to $t_s < n/3$ [10] and $t_a < n/4$ [9] corrupt parties respectively. Following the notion of [29], given an adversary structure $\mathcal{Z}$ and a subset of parties $\mathcal{P}' \subseteq \mathcal{P}$, we say that $\mathcal{Z}$ satisfies the $\mathcal{Q}^{(k)}(\mathcal{P}', \mathcal{Z})$ condition if the union of any k subsets from $\mathcal{Z}$ *does not* cover $\mathcal{P}'$. That is, for *every* $Z_{i_1}, Z_{i_2}, \ldots, Z_{i_k} \in \mathcal{Z}$, the following holds:

$$\mathcal{P}' \not\subseteq Z_{i_1} \cup \ldots \cup Z_{i_k}.$$

SMPC and AMPC against general adversaries is possible, provided the underlying adversary structure $\mathcal{Z}$ satisfies the $\mathcal{Q}^{(3)}(\mathcal{P}, \mathcal{Z})$ [29] and $\mathcal{Q}^{(4)}(\mathcal{P}, \mathcal{Z})$ condition [32] respectively.

Our Motivation and Results. In an MPC protocol, it is usually *assumed* that the parties will be knowing if the underlying network is synchronous or asynchronous *beforehand.* Suppose that the parties are *not aware* of the network type. We aim to design a *single* MPC protocol that is capable of adapting to the exact timing behaviour of the underlying network while offering the best possible security guarantees in either network. We call such a protocol a *best-of-both-worlds* (BoBW) or a network-agnostic MPC protocol. Recently, [2] presented a BoBW *perfectly-secure* MPC protocol against *threshold* adversaries which could tolerate up to t_s and t_a corruptions in a *synchronous* and *asynchronous* network respectively, for any $t_a < t_s$ where $t_a < n/4$ and $t_s < n/3$, provided $3t_s + t_a < n$ holds. We aim to generalize this result against *general* adversaries, and ask the following question:

Let $\mathcal{A}$ be an adversary characterized by adversary structures $\mathcal{Z}_s$ and $\mathcal{Z}_a$ in a synchronous network and asynchronous network respectively, where $\mathcal{Z}_s \neq \mathcal{Z}_a$. Then, is there a BoBW perfectly-secure MPC protocol which is secure against $\mathcal{A}$, irrespective of the network type?

No prior work has addressed the above question. We present a BoBW perfectly-secure MPC protocol provided *all* the following conditions hold, which we refer to throughout as Con.[2]

▶ **Condition 1** ($\mathsf{Con}(\mathcal{Z}_s, \mathcal{Z}_a)$)**.** *$\mathcal{Z}_s$ and $\mathcal{Z}_a$ satisfy the following conditions.*

- *$\mathcal{Z}_s \neq \mathcal{Z}_a$, and $\mathcal{Z}_s, \mathcal{Z}_a$ satisfy the $\mathbb{Q}^{(3,1)}(\mathcal{P}, \mathcal{Z}_s, \mathcal{Z}_a)$ condition, meaning that the union of any 3 subsets from $\mathcal{Z}_s$ and any one subset from $\mathcal{Z}_a$, does not cover $\mathcal{P}$.*
- *Every subset in $\mathcal{Z}_a$ is a subset of some subset in $\mathcal{Z}_s$.*

The computation and communication complexity of our protocol is polynomial in n and $|\mathcal{Z}_s|$.

Significance of Our Result. We focus on the case where $\mathcal{Z}_s \neq \mathcal{Z}_a$ as, otherwise, the question is *trivial* to solve [3]. Let $\mathcal{P} = \{P_1, \ldots, P_8\}$, $\mathcal{Z}_s = \{\{P_1, P_2, P_3\}, \{P_2, P_3, P_4\}, \{P_3, P_4, P_5\}, \{P_4, P_5, P_6\}, \{P_7\}, \{P_8\}\}$ and $\mathcal{Z}_a = \{\{P_1, P_3\}, \{P_2, P_4\}, \{P_3, P_5\}, \{P_4, P_6\}\}$. Since $\mathcal{Z}_s$ and $\mathcal{Z}_a$ satisfy $\mathbb{Q}^{(3)}(\mathcal{P}, \mathcal{Z}_s)$ and $\mathbb{Q}^{(4)}(\mathcal{P}, \mathcal{Z}_a)$ conditions respectively, it follows that *existing* SMPC and AMPC protocols can tolerate $\mathcal{Z}_s$ and $\mathcal{Z}_a$ respectively. However, we show that *even* if the parties are *not aware* of the exact network type, then using our protocol, one can *still* achieve security against $\mathcal{Z}_s$ if the network is *synchronous* or against $\mathcal{Z}_a$ if the network is *asynchronous*. The above example demonstrates the flexibility offered by the non-threshold adversary model, in terms of tolerating *more* faults. In the *threshold* model, using the protocol of [2], one can tolerate up to $t_s = 2$ and $t_a = 1$ faults, in a *synchronous* and *asynchronous* network respectively. In the *non-threshold* model, our protocol can tolerate subsets of size larger than the maximum t_s and t_a allowed in a synchronous and asynchronous network.

We compare the communication complexity of our network-agnostic MPC protocol with the most efficient existing synchronous and asynchronous MPC protocols in Table 1.[4] Here, $(\mathbb{K}, +, \cdot)$ denotes the finite ring (or field) over which the computations are performed.

Table 1 Amortized communication complexity per multiplication of different perfectly-secure MPC protocols against general adversaries.

Setting	Reference	Condition	Communication Complexity (in bits)
Synchronous	[30]	$\mathbb{Q}^{(3)}(\mathcal{P}, \mathcal{Z})$	$\mathcal{O}(\lvert\mathcal{Z}\rvert^2 \cdot (n^5 \log \lvert\mathbb{K}\rvert + n^6) + \lvert\mathcal{Z}\rvert \cdot (n^7 \log \lvert\mathbb{K}\rvert + n^8))$
Asynchronous	[3]	$\mathbb{Q}^{(4)}(\mathcal{P}, \mathcal{Z})$	$\mathcal{O}(\lvert\mathcal{Z}\rvert^2 \cdot n^7 \log \lvert\mathbb{K}\rvert + \lvert\mathcal{Z}\rvert \cdot n^9 \log n)$
Network Agnostic	This work	$\mathsf{Con}(\mathcal{Z}_s, \mathcal{Z}_a)$	$\mathcal{O}(\lvert\mathcal{Z}_s\rvert^2 \cdot n^5 (\log \lvert\mathbb{K}\rvert + \log \lvert\mathcal{Z}_s\rvert + \log n))$

1.1 Technical Overview

Like in any generic MPC protocol [27, 10, 38], we assume that the underlying computation (which the parties want to perform securely) is modelled as some publicly-known function f, abstracted by some arithmetic circuit cir, over some algebraic structure $\mathbb{K}$, consisting of linear and non-linear (multiplication) gates. The problem of secure computation then reduces to secure *circuit-evaluation*, where the parties jointly and securely "evaluate" cir in a *secret-shared* fashion, such that all the values during the circuit-evaluation remain *verifiably secret-shared* and where the shares of the corrupt parties *fail* to reveal the exact

[2] Conditions Con imply that $\mathcal{Z}_s$ and $\mathcal{Z}_a$ satisfy the $\mathbb{Q}^{(3)}(\mathcal{P}, \mathcal{Z}_s)$ and $\mathbb{Q}^{(4)}(\mathcal{P}, \mathcal{Z}_a)$ conditions respectively.

[3] If $\mathcal{Z}_s = \mathcal{Z}_a$, then AMPC is possible only if *even* $\mathcal{Z}_s$ satisfies the $\mathbb{Q}^{(4)}(\mathcal{P}, \mathcal{Z}_s)$ condition. *Any* existing perfectly-secure AMPC protocol (with appropriate time-outs) [32, 18, 3] will work *even* in the synchronous network, with the guarantee that the inputs of *all honest* parties are considered for the computation

[4] Conventionally, the communication complexity of any generic MPC protocol is measured in terms of the number of bits communicated to evaluate a single multiplication gate in the underlying circuit.

underlying value. The secret-sharing used is typically *linear* [20], thus allowing the parties to evaluate the linear gates *locally* (non-interactively). On the other hand, non-linear gates are evaluated by deploying the standard Beaver's method [8] using random, secret-shared *multiplication-triples* which are generated in a circuit-independent *preprocessing phase*. Then, once all the gates are securely evaluated, the parties publicly reconstruct the secret-shared circuit-output. Apart from *verifiable secret-sharing* (VSS) [16], the parties also need to run instances of a *Byzantine agreement* (BA) protocol [37] to ensure that all the parties are on the "same page" during the various stages of the circuit-evaluation. The above framework for shared circuit-evaluation is defacto used in *all* generic perfectly-secure SMPC and AMPC protocols. Unfortunately, there are several obstacles while adapting the framework if the parties are *unaware* of the network type.

First Obstacle – A BoBW BA Protocol. Informally, a BA protocol [37] allows parties with private inputs to reach an agreement on a *common* output, even if a subset of the parties behave maliciously. In the *non-threshold* setting, one can design perfectly-secure BA protocol against $\mathcal{Q}^{(3)}$ adversary structures *irrespective* of the network type [25, 17]. However, the *termination* (also called *liveness*) guarantees are *different* for *synchronous* BA (SBA) and *asynchronous* BA (ABA) protocols. The (deterministic) SBA protocols ensure that all honest parties obtain their output after some fixed time (*guaranteed liveness*) [37]. On the other hand, to circumvent the FLP impossibility result [24], ABA protocols are *randomized* and provide *almost-surely liveness* [1, 7, 17], where the parties terminate the protocol asymptotically with a probability of 1. Known SBA protocols become insecure in an *asynchronous* network even if one expected message from an honest party gets arbitrarily delayed, while existing ABA protocols can provide *only* almost-surely liveness in a *synchronous* network.

The *first* obstacle is to get a BoBW BA protocol against non-threshold adversaries, which provides the security guarantees of SBA and ABA in a *synchronous* and an *asynchronous* network respectively. We present such a BA protocol which is secure against $\mathcal{Q}^{(3)}$ adversary structures. The protocol is obtained by generalizing the BoBW BA protocol of [2] which is secure against *threshold* adversaries and tolerates $t < n/3$ faults.

Second Obstacle – A BoBW VSS Protocol. In a VSS protocol, a designated *dealer* $\mathsf{D} \in \mathcal{P}$ has some private input s. The goal is to let D "verifiably" distribute shares of s such that the adversary does not learn anything additional about s, if D is *honest* (*privacy*). In a *synchronous* VSS (SVSS), every (honest) party obtains its shares after some *known* time-out (*correctness*). *Verifiability* guarantees that even a *corrupt* D shares some value "consistently" within the known time-out (*commitment* property). Perfectly-secure SVSS is possible, provided the underlying adversary structure $\mathcal{Z}_s$ satisfies $\mathcal{Q}^{(3)}$ condition [34, 30].

For an *asynchronous* VSS (AVSS) protocol, *correctness* guarantees that for an *honest* D, the secret s is eventually secret-shared. However, a *corrupt* D *may not* invoke the protocol in the first place, in which case the honest parties may not obtain any shares. Hence, the *commitment* property of AVSS guarantees that if D is *corrupt and* if some honest party computes a share (implying that D has invoked the protocol), then all honest parties eventually compute their shares. Perfectly-secure AVSS is possible, provided the underlying adversary structure $\mathcal{Z}_a$ satisfies the $\mathcal{Q}^{(4)}$ condition [18, 3].

Existing SVSS protocols become insecure in an asynchronous network, even if a single expected message from an *honest* party is *delayed*. On the other hand, existing AVSS protocols are insecure against $\mathcal{Q}^{(3)}$ adversary structures (which SVSS protocols can tolerate). Since, in our setting, the parties will *not* be knowing the exact network type, to maintain *privacy* during the shared circuit-evaluation, we need to ensure that each value remains secret-shared with respect to $\mathcal{Z}_s$ rather than *not* $\mathcal{Z}_a$, *even* if the network is *asynchronous*.[5] The *second* obstacle to perform shared circuit-evaluation in our setting is to get a perfectly-secure VSS protocol which is secure with respect to $\mathcal{Z}_s$ and $\mathcal{Z}_a$ in a *synchronous* and *asynchronous* network respectively and where *privacy always holds* with respect to $\mathcal{Z}_s$, *irrespective* of the network type. We are not aware of any VSS protocol with these guarantees. Hence, we present a BoBW VSS protocol satisfying the required properties.

Our BoBW VSS protocol is obtained by carefully and non-trivially "stitching" together the SVSS and AVSS protocols of [34] and [18] respectively. Both these protocols are further based on the classic *additive* secret-sharing protocol of Ito et al [31] (designed against passive adversaries). The secret is shared using a *sharing specification* $\mathbb{S}_{\mathcal{Z}}$ corresponding to a given adversary structure $\mathcal{Z}$, where $\mathbb{S}_{\mathcal{Z}}$ is the collection of "set-complements" of the subsets in $\mathcal{Z}$. That is, if $\mathcal{Z} = \{Z_1, \ldots, Z_{|\mathcal{Z}|}\}$, then $\mathbb{S}_{\mathcal{Z}} = (S_1, \ldots, S_{|\mathcal{Z}|})$ where $S_m = \mathcal{P} \setminus Z_m$, for $m = 1, \ldots, |\mathcal{Z}|$. The idea behind the secret-sharing of [31] is then to share a secret s through a *random* vector of shares $(s_1, \ldots, s_{|\mathcal{Z}|})$ which sum up to s, where all (honest) parties in the group S_m hold the share s_m. Since one of the subsets in $\mathbb{S}_{\mathcal{Z}}$ consists of *only honest* parties, it would be ensured that if D is *honest*, then the probability distribution of the shares learnt by the adversary is *independent* of s. The SVSS and AVSS protocols of [34] and [18] ensure that the underlying secret is indeed shared as per the above semantics, even in the presence of *malicious* corruptions, including a potentially corrupt D. We next briefly discuss these protocols individually and then give a high-level overview of how we combine them.

- **SVSS Against $\mathcal{Q}^{(3)}$ Adversary Structures [34]:** Consider an arbitrary adversary structure $\mathcal{Z}_s$ satisfying the $\mathcal{Q}^{(3)}(\mathcal{P}, \mathcal{Z}_s)$ condition, and let $\mathbb{S}_{\mathcal{Z}_s} = (S_1, \ldots, S_{|\mathcal{Z}_s|})$ be the corresponding sharing specification. The protocol is executed as a sequence of *phases*. To share s, during the *first* phase, D picks a random vector of shares $(s_1, \ldots, s_{|\mathcal{Z}_s|})$, such that $s = s_1 + \ldots + s_{|\mathcal{Z}_s|}$. Then all parties in every group $S_m \in \mathbb{S}_{\mathcal{Z}_s}$ are given share s_m by D. To check whether a potentially *corrupt* D has given the same share to all the (honest) parties in S_m, the parties in S_m perform a *pairwise consistency* check of their supposedly common share during the *second* phase, and publicly broadcast the results during the *third* phase, using a *synchronous* reliable broadcast protocol. If any party in S_m publicly complains about an inconsistency, then during the *fourth* phase, D makes public the share s_m corresponding to S_m by broadcasting it. This *does not* violate the privacy for an *honest* D, since a complaint for inconsistency from S_m implies that S_m has at least one *corrupt* party and so, the adversary will already know s_m. If D *does not* "resolve" any complaint during the fourth phase (implying D is *corrupt*), then D is *publicly discarded*, and everyone takes a default sharing of 0 on the behalf of D. Clearly, the protocol ensures that by the end of the *fourth* phase, *all honest* parties in S_m have the *same* share, and the sum of these shares across all the S_m sets is the value shared by D.
- **AVSS Against $\mathcal{Q}^{(4)}$ Adversary Structures [18]:** Consider an arbitrary adversary structure $\mathcal{Z}_a$ satisfying the $\mathcal{Q}^{(4)}(\mathcal{P}, \mathcal{Z}_a)$ condition, and let $\mathbb{S}_{\mathcal{Z}_a} = (S_1, \ldots, S_{|\mathcal{Z}_a|})$ be the corresponding sharing specification. The AVSS protocol of [18] closely follows the SVSS protocol of [34]. However, the phases are *no longer* synchronized. Moreover, during the pairwise consistency phase, the parties *cannot* afford to wait to know the status of the consistency checks between all pairs of parties, since potentially *corrupt* parties may *never* respond. Instead, corresponding to every S_m, the parties check for the existence of a set

[5] Since we are assuming that every subset in $\mathcal{Z}_a$ is a subset of some subset in $\mathcal{Z}_s$, privacy will be maintained *irrespective* of the network type if each value remains secret-shared with respect to $\mathcal{Z}_s$.

of "core" parties $\mathcal{C}_m \subseteq S_m$, with $S_m \setminus \mathcal{C}_m \in \mathcal{Z}_a$, which publicly confirmed that they are pairwise consistent. To ensure that all the parties agree on the core sets, D is assigned the task of identifying the core sets and broadcasting them (where the broadcast now happens through an *asynchronous* reliable broadcast protocol). The protocol proceeds *only* upon the receipt of core sets from D and their verification. While an *honest* D will eventually find and broadcast valid core sets, a *corrupt* D may *not* do so, in which case the parties obtain no shares. Once the core sets are identified and verified, it is guaranteed that all the (honest) parties in each core set $\mathcal{C}_m$ have received the same share from D. The goal is then to ensure that even the (honest) parties "outside" $\mathcal{C}_m$ (namely, the parties in $S_m \setminus \mathcal{C}_m$) get this common share. Since $\mathcal{Z}_a$ satisfies the $\mathcal{Q}^{(4)}(\mathcal{P}, \mathcal{Z}_a)$ condition, the "majority" of the parties in $\mathcal{C}_m$ are *honest* [6]. Hence, the parties in $S_m \setminus \mathcal{C}_m$ can "extract" the common share held by the parties in $\mathcal{C}_m$, by applying the "majority rule" on the shares received from the parties in $\mathcal{C}_m$, during the pairwise consistency tests.

- **Our BoBW VSS Protocol:** In our VSS protocol, the parties first start executing the steps of the above SVSS protocol, *assuming* a *synchronous* network, where all the instances of broadcast happen by executing an instance of a BoBW reliable broadcast protocol Π_{BC}, designed as part of our BoBW BA protocol. Let T_{BC} be the time taken by the protocol Π_{BC} to produce the output in a *synchronous* network. If indeed the network is *synchronous*, then within time $2\Delta + T_{\mathsf{BC}}$, the results of pairwise consistency tests should be publicly available, where Δ is the upper bound on message delay in a *synchronous* network. Moreover, if any inconsistency is reported, then within the time $2\Delta + 2T_{\mathsf{BC}}$, the dealer D should have resolved all those inconsistencies by making the "disputed" shares public. However, unlike the SVSS protocol, the parties *cannot* afford to discard D if it fails to resolve any inconsistency within time $2\Delta + 2T_{\mathsf{BC}}$. This is because the network could be *asynchronous*, and D's responses may be arbitrarily *delayed*, even if D is *honest*. A bigger challenge is that in an *asynchronous* network, some honest parties, say $\mathcal{H}_1$, *might* be seeing the inconsistencies being reported within local time $2\Delta + T_{\mathsf{BC}}$, *as well as* D's responses within the local time $2\Delta + 2T_{\mathsf{BC}}$. And there might be another set of honest parties, say $\mathcal{H}_2$, who *might not* be seeing these inconsistencies and D's responses within these timeouts. This may result in the parties in $\mathcal{H}_1$ considering the shares made public by D, while the parties in $\mathcal{H}_2$ may think that the network is *asynchronous* and wait for the core sets of parties to be made public by D (as done in the AVSS). However, this gives a *corrupt* D an opportunity to *violate* the *commitment* property in an *asynchronous* network. In more detail, consider a set S_m for which pairwise *inconsistency* is reported, and for which D also finds a set of core parties $\mathcal{C}_m$. Then, it might be possible that the parties in $\mathcal{C}_m$ have received the common share s_m from D, but in response to the inconsistencies reported for S_m, D broadcasts the share s'_m, where $s'_m \neq s_m$. This will lead to a situation where the parties in $\mathcal{H}_1$ consider s'_m as the share for the group S_m after the timeout of $2\Delta + 2T_{\mathsf{BC}}$. On the other hand, the parties in $\mathcal{H}_2$ *may not* see the inconsistencies and s'_m within the timeout of $2\Delta + 2T_{\mathsf{BC}}$, but eventually see $\mathcal{C}_m$ and extract the share s_m corresponding to S_m.

  To deal with the above challenge, apart from resolving the inconsistencies reported for *any* set S_m, the dealer D *also* finds and broadcasts a core set of parties $\mathcal{C}_m$, who have confirmed receiving the same share from D corresponding to *all* the sets S_m, such that

[6] Since the $\mathcal{Q}^{(4)}(\mathcal{P}, \mathcal{Z}_a)$ condition is satisfied, the conditions $\mathcal{Q}^{(3)}(S_m, \mathcal{Z}_a)$ and, consequently, $\mathcal{Q}^{(2)}(\mathcal{C}_m, \mathcal{Z}_a)$ are also satisfied. Thus, the $\mathcal{Q}^{(1)}(\mathcal{C}_m \setminus Z^\star, \mathcal{Z}_a)$ condition is satisfied, where $Z^\star$ is the actual set of corrupt parties, implying that the set of honest parties form a "majority".

$S_m \setminus \mathcal{C}_m \in \mathcal{Z}_s$. *Additionally*, if there is any inconsistency reported for S_m, then *apart* from D, *every* party in S_m *also* makes public its version of the share corresponding to S_m received from D. Now, at time $2\Delta + 2T_{\mathsf{BC}}$, the parties check if D has broadcasted a core set $\mathcal{C}_m$ for each S_m. Moreover, if any inconsistency has been reported corresponding to S_m, the parties check if "sufficiently many" parties from $\mathcal{C}_m$ have made public the same share which D made public. This *prevents* a *corrupt* D from making public a share that is *different* from the share which it distributed to the parties in $\mathcal{C}_m$.

If the network is *asynchronous*, then different parties may have *different* "opinions" regarding whether D has broadcasted "valid" core sets $\mathcal{C}_m$. Hence, at time $2\Delta + 2T_{\mathsf{BC}}$, the parties run an instance of our BoBW BA protocol to decide what the case is. If the parties find that D has broadcasted valid core sets $\mathcal{C}_m$ corresponding to each S_m, then the parties in S_m proceed to compute their share as follows: if D has made public the share for S_m in response to any inconsistency, then it is taken as the share for S_m. If no share has been made public for S_m, then the parties check if "sufficiently many" parties have reported the same share during the pairwise consistency test within time 2Δ, which we show should have happened if the network is *synchronous*, and if the parties maintain sufficient timeouts. If none of these conditions holds, then the parties proceed to filter out the common share, held by the parties in $\mathcal{C}_m$, through the "majority rule".

On the other hand, if the parties find that D has *not* made public core sets within time $2\Delta + 2T_{\mathsf{BC}}$, then either the network is *asynchronous* or D is *corrupt*. So the parties resort to the steps used in AVSS. Namely, D finds and broadcasts a set of core parties $\mathcal{E}_m$ corresponding to each S_m, where $S_m \setminus \mathcal{E}_m \in \mathcal{Z}_a$. [7] Then, the parties filter out the common share, held by the parties in $\mathcal{E}_m$, through majority rule (see Section 4 for details).

Best-of-Both-Worlds Secure Multiplication. Apart from BoBW VSS and BA, another key component in our MPC protocol is a BoBW multiplication protocol against general adversaries. This is again obtained by carefully stitching together the synchronous and asynchronous multiplication protocol of [34] and [18] respectively. The protocol takes as input secret-shared a and b, both shared with respect to $\mathcal{Z}_s$, and securely outputs a secret-sharing of $a \cdot b$ with respect to $\mathcal{Z}_s$, *irrespective* of the network type. Let $(a_1, \ldots, a_{|\mathcal{Z}_s|})$ and $(b_1, \ldots, b_{|\mathcal{Z}_s|})$ be the vector of shares, corresponding to a and b respectively. The idea here is to securely generate a secret-sharing of each of the summands $a_l \cdot b_m$, where $l, m \in \{1, \ldots, |\mathcal{Z}_s|\}$. The linearity property (see Definition 2) of the secret-sharing then guarantees that a secret-sharing of $a \cdot b$ can be obtained from the secret-sharing of the summands $a_l \cdot b_m$.

To generate a secret-sharing of $a_l \cdot b_m$, the parties do the following: let $\mathcal{I}_{l,m}$ be the set of parties who have both a_l and b_m. Since $\mathbb{Q}^{(3,1)}(\mathcal{P}, \mathcal{Z}_s, \mathcal{Z}_a)$ condition is satisfied, *irrespective* of the network type, $\mathcal{I}_{l,m}$ *will* have *at least* one honest party. Each party in $\mathcal{I}_{l,m}$ is asked to independently secret-share $a_l \cdot b_m$ through an instance of our BoBW VSS protocol. To avoid an endless wait, the parties *cannot* afford for *all* the parties in $\mathcal{I}_{l,m}$ to secret-share their "versions" of $a_l \cdot b_m$, even if the network would have been *synchronous*. Hence the parties run instances of our BoBW BA to agree on a common subset of parties $\mathcal{R}_{l,m}$ from $\mathcal{I}_{l,m}$, where $\mathcal{I}_{l,m} \setminus \mathcal{R}_{l,m} \in \mathcal{Z}_s$, who have shared some version of $a_l \cdot b_m$ through VSS instances. However, we take special care to ensure that *irrespective* of the network type, the set $\mathcal{R}_{l,m}$ has at least one *honest* party from $\mathcal{I}_{l,m}$, who has indeed shared the summand $a_l \cdot b_m$. Note that achieving this goal is *not* a challenge for the *synchronous* multiplication protocol of [34], since

[7] $\mathcal{E}_m$ (not to be confused with $\mathcal{C}_m$) is the core set of parties corresponding to S_m which D finds in case it is unable to find and make public valid core sets $\mathcal{C}_m$ "on time" for each S_m.

$\mathcal{R}_{l,m} = \mathcal{I}_{l,m}$ holds.[8] Similarly, the goal is *easily* achievable in the *asynchronous* multiplication protocol of [18].[9] To ensure that the $\mathcal{R}_{l,m}$ has at least one *honest* party, we carefully run instances of our BoBW BA and decide the timeouts of the parties in these BA instances (see Section 5 for the exact details). Once the set $\mathcal{R}_{l,m}$ is decided, the parties then check if *all* the parties in $\mathcal{R}_{l,m}$ have shared the same version of $a_l \cdot b_m$. If all the versions are the same, then any one of these is taken as a secret-sharing of $a_l \cdot b_m$. Else at least one party from $\mathcal{R}_{l,m}$ has behaved maliciously and so the parties publicly reconstruct the shares a_l and b_m and compute a default secret-sharing of $a_l \cdot b_m$.[10]

Comparison of Our Results with [2]. Even though our BoBW BA protocol is an easy generalization of the BoBW BA protocol of [2] against *threshold* adversaries, our VSS protocol and the multiplication protocol are relatively simpler and based on completely different ideas. For instance, the BoBW VSS protocol of [2] is based on the properties of symmetric bivariate polynomials of degree t_s in two variables over a finite field, where the underlying secret is embedded in the constant term of the polynomial and the share for each party is a distinct univariate polynomial, lying on the bivariate polynomial (this is a two-dimensional extension of the classical Shamir's secret-sharing [39]). The bivariate polynomials help to verify whether a potentially *corrupt* D has distributed shares consistently. However, verifying the same in the BoBW setting is quite challenging. As a result, the VSS protocol of [2] is quite involved and is further based on a "weaker" primitive, called *weak polynomial-sharing* (WPS) [36, 5], which ensures that if the dealer is *corrupt*, then *only* a subset of the *honest* parties receive their designated shares.[11] On the contrary, our BoBW VSS protocol is much *simpler* and *not* based on any WPS protocol. Intuitively this is because the "sharing-semantics" of the underlying secret-sharing is *different* for VSS against the threshold and non-threshold adversaries. While the former is based on polynomial interpolation, the latter deploys additive secret-sharing. Consequently, there is more "redundancy" available to verify whether D has consistently shared its secret, compared to bivariate polynomials, since each candidate share is now available with multiple parties. To the best of our knowledge, the idea of designing VSS based on WPS has been used *only* against *threshold* adversaries and it is *not* known whether the idea can be generalized against *non-threshold* adversaries.

Similarly, the multiplication protocol of [2] is quite involved and based on the framework of [19], which further involves a lot of subprotocols and deploys properties of polynomial evaluation and interpolation over finite fields. In contrast, our multiplication protocol is relatively simpler and straightforward and *does not* involve multiple sub-protocols.

1.2 Other Related Work

All existing works in the domain of BoBW protocols focus only on *threshold* adversaries. The works of [12, 14, 21] show that the condition $2t_s + t_a < n$ is necessary and sufficient for BoBW *cryptographically-secure* BA and MPC, tolerating *computationally bounded* adversaries. Using

[8] In a synchronous network, a and b are secret-shared with respect to a set $\mathcal{Z}$ satisfying $\mathcal{Q}^{(3)}(\mathcal{P}, \mathcal{Z})$ condition. This ensures that $\mathcal{Z}$ satisfies the $\mathcal{Q}^{(1)}(\mathcal{I}_{l,m}, \mathcal{Z})$ condition and hence contains at least one honest party. Moreover, in a *synchronous* network, the VSS instances of *all* the parties in $\mathcal{I}_{l,m}$ get over within a known time bound and hence $\mathcal{R}_{l,m} = \mathcal{I}_{l,m}$ holds.

[9] In an asynchronous network, a and b are secret-shared with respect to a set $\mathcal{Z}$ satisfying $\mathcal{Q}^{(4)}(\mathcal{P}, \mathcal{Z})$ condition. This ensures that $\mathcal{Z}$ satisfies the $\mathcal{Q}^{(2)}(\mathcal{I}_{l,m}, \mathcal{Z})$ condition. Consequently, $\mathcal{I}_{l,m} \setminus \mathcal{R}_{l.m} \in \mathcal{Z}$ will hold, implying that $\mathcal{Z}$ satisfies the $\mathcal{Q}^{(1)}(\mathcal{R}_{l,m}, \mathcal{Z})$ condition and $\mathcal{R}_{l,m}$ contains at least one honest party.

[10] The vector of shares $(s, 0, \ldots, 0)$ can be considered as a default sharing of a publicly known value s.

[11] It is *not* known how to *directly* design a BoBW VSS protocol, *without* deploying any WPS.

the same condition, [13] presents a BoBW *cryptographically-secure* atomic broadcast protocol. The work of [35] studies Byzantine fault tolerance and state machine replication protocols for multiple thresholds, including t_s and t_a. The work of [26] presents a BoBW protocol for the task of approximate agreement using the condition $2t_s + t_a < n$. The same condition has been used to design a BoBW distributed key-generation (DKG) protocol in [6]. A recent work [22] has studied the problem of perfectly-secure message transmission (PSMT) [23] over *incomplete* graphs, in the BoBW setting. Along with the results of [2], they note that BoBW perfectly-secure MPC over incomplete networks is possible as long as $3t_s + t_a < n$ *and* $t_s + 2t_a < N$, where N is the connectivity of the graph modelling the underlying network.

1.3 Open Problems

We do not know whether the conditions Con are indeed necessary for any BoBW perfectly-secure MPC protocol. In fact, it is not known whether the corresponding condition $3t_s+t_a < n$ is necessary for any BoBW perfectly-secure MPC against *threshold* adversaries. We conjecture that these conditions are indeed necessary for the respective adversarial model, for any BoBW perfectly-secure MPC. The main aim of this work (and [2]) is to show the feasibility of BoBW perfectly-secure MPC against general adversaries over complete networks. We do not know if an equivalent result for MPC over incomplete networks can be shown as in [22]. Improving the efficiency of these protocols is also left for future work.

2 Preliminaries and Definitions

The parties in $\mathcal{P}$ are assumed to be connected by pair-wise secure channels. The underlying communication network can be either synchronous or asynchronous, with parties being *unaware* about the exact type. In a *synchronous* network, every sent message is delivered within a *known* time Δ. In an *asynchronous* network, messages can be delayed arbitrarily, but finitely, with every message sent being delivered *eventually*. The distrust among $\mathcal{P}$ is modelled by a *malicious* (byzantine) adversary $\mathcal{A}$, who can corrupt a subset of the parties in $\mathcal{P}$ and force them to behave in any arbitrary fashion during the execution of a protocol. For simplicity, we assume the adversary to be *static*, it decides the set of corrupt parties at the beginning of the protocol execution. However, our protocols can be proved secure even against a more powerful *adaptive* adversary that can decide the set of corrupt parties at run time.

Adversary $\mathcal{A}$ can corrupt any one subset of parties from $\mathcal{Z}_s$ and $\mathcal{Z}_a$ in *synchronous* and *asynchronous* networks respectively. The adversary structures are *monotone*, implying that if $Z \in \mathcal{Z}_s$ ($Z \in \mathcal{Z}_a$ resp.), then every subset of Z also belongs to $\mathcal{Z}_s$ (resp. $\mathcal{Z}_a$). We say that $\mathcal{Z}_s$ and $\mathcal{Z}_a$ satisfy the $\mathcal{Q}^{(k,k')}(\mathcal{P}, \mathcal{Z}_s, \mathcal{Z}_a)$ condition if the union of any k subsets from $\mathcal{Z}_s$ and any k' subsets from $\mathcal{Z}_a$, *does not* cover $\mathcal{P}$. That is, for every $Z_{i_1}, \ldots, Z_{i_k} \in \mathcal{Z}_s$ and every $Z_{j_1}, \ldots, Z_{j_{k'}} \in \mathcal{Z}_a$, the condition $\mathcal{P} \not\subseteq Z_{i_1} \cup \ldots \cup Z_{i_k} \cup Z_{j_1} \cup \ldots \cup Z_{j_{k'}}$ holds.

In our VSS and MPC protocols, all computations are done over a finite algebraic structure $(\mathbb{K}, +, \cdot)$, which could be a ring or a field. Without loss of generality, we assume that each P_i has an input $x_i \in \mathbb{K}$, and the parties want to securely compute a function $f : \mathbb{K}^n \rightarrow \mathbb{K}$, represented by an arithmetic circuit cir over $\mathbb{K}$, consisting of linear and non-linear (multiplication) gates, where cir has c_M multiplication gates and a multiplicative depth of D_M.

Termination Guarantees of Our Sub-Protocols. As done in [2], for simplicity, we will *not* be specifying any *termination* criteria for our sub-protocols. The parties will keep on participating in these sub-protocol instances even *after* computing their outputs. The termination criteria of our MPC protocol will ensure the termination of *all* underlying sub-protocol instances. We will be using an existing *randomized* ABA protocol [17] which ensures that the honest parties (eventually) obtain their respective output *almost-surely* with probability 1, where the probability is over the random coins of the honest parties and adversary in the protocol. The property of almost-surely obtaining an output carries over to the "higher" level protocols, where ABA is used as a building block.

We next discuss the syntax and semantics of the secret-sharing used in our VSS.

▶ **Definition 2** ([34]). *Let $\mathbb{S} = (S_1, \ldots, S_{|\mathbb{S}|})$ be a set called the sharing specification where, for $m = 1, \ldots, |\mathbb{S}|$, each $S_m \subseteq \mathcal{P}$. Then a value $s \in \mathbb{K}$ is said to be secret-shared with respect to $\mathbb{S}$ if there exist shares $s_1, \ldots, s_{|\mathbb{S}|} \in \mathbb{K}$ such that $s = s_1 + \ldots + s_{|\mathbb{S}|}$ and, for $m = 1, \ldots, |\mathbb{S}|$, the share s_m is available to every (honest) party in S_m.*

A secret-sharing of s is denoted by $[s]$, where $[s]_m$ denotes the m^{th} share. The above secret-sharing is *linear* as $[c_1 s_1 + c_2 s_2] = c_1[s_1] + c_2[s_2]$ holds for publicly-known $c_1, c_2 \in \mathbb{K}$. Hence, the parties can *non-interactively* compute any linear function over secret-shared inputs. For our protocols, we will consider the sharing specification $\mathbb{S} = \{S_m : S_m = \mathcal{P} \setminus Z_m \text{ and } Z_m \in \mathcal{Z}_s\}$.

2.1 Existing Asynchronous Primitives

Asynchronous Reliable Broadcast (Acast). An Acast protocol allows a designated *sender* $\mathsf{S} \in \mathcal{P}$ to send its input $m \in \{0,1\}^\ell$ *identically* to all the parties, even if S is potentially corrupt. An Acast protocol Π_{ACast} is presented in [33], provided $\mathcal{Z}$ satisfies the $\mathcal{Q}^{(3)}(\mathcal{P}, \mathcal{Z})$ condition. The protocol also provides certain guarantees in a *synchronous* network, as stated in Lemma 8 (Appendix A). The protocol, along with the proof of Lemma 8 and various terminologies associated with Π_{ACast} are available in the full version of this paper [4].

Public Reconstruction of a Secret-Shared Value. Let $s \in \mathbb{K}$ be a value, which is secret-shared with respect to $\mathbb{S} = \{S_m : S_m = \mathcal{P} \setminus Z_m \text{ and } Z_m \in \mathcal{Z}_s\}$. To publicly reconstruct s, we use the reconstruction protocol $\Pi_{\mathsf{Rec}}(s, \mathbb{S})$ of [34]. In a *synchronous* network, the protocol will take Δ time, while in an *asynchronous* network, the parties eventually output s. The protocol incurs a communication of $\mathcal{O}(|\mathcal{Z}_s| \cdot n^2 \log |\mathbb{K}|)$ bits; see [4] for the details.

3 Best-of-Both-Worlds Byzantine Agreement (BA)

We begin with the definition of BA, which is adapted from [14, 2].

▶ **Definition 3** (BA). *Let Π be a protocol for $\mathcal{P}$ where every P_i has input $b_i \in \{0,1\}$ and a possible output from $\{0,1,\bot\}$. Let $\mathcal{A}$ be an adversary, characterized by adversary structure $\mathcal{Z}$, where $\mathcal{A}$ can corrupt any set of parties from $\mathcal{Z}$ during the execution of Π.*

- **$\mathcal{Z}$-Guaranteed Liveness:** *All honest parties obtain an output.*
- **$\mathcal{Z}$-Almost-Surely Liveness:** *Almost-surely, all honest parties obtain some output.*
- **$\mathcal{Z}$-Validity:** *If all honest parties input b, every honest party with an output outputs b.*
- **$\mathcal{Z}$-Weak Validity:** *If all honest parties input b, every honest party with an output outputs b or $\bot$.*
- **$\mathcal{Z}$-Consistency:** *All honest parties with an output output the same value (may be $\bot$).*
- **$\mathcal{Z}$-Weak Consistency:** *All honest parties with an output output a common $v \in \{0,1,\bot\}$.*

A $\mathcal{Z}$-perfectly-secure synchronous BA (SBA) protocol Π *has $\mathcal{Z}$-guaranteed liveness, $\mathcal{Z}$-validity, and $\mathcal{Z}$-consistency in a synchronous network. A $\mathcal{Z}$-perfectly-secure asynchronous BA (ABA)* Π *has $\mathcal{Z}$-almost-surely liveness, $\mathcal{Z}$-validity and $\mathcal{Z}$-consistency in an asynchronous network.*[12]

To design our BoBW BA protocol, we will need a special broadcast protocol. Hence, we next review the definition of broadcast, adapted from [14, 2].

▶ **Definition 4 (Broadcast).** *Let* Π *be a protocol where a sender* $\mathsf{S} \in \mathcal{P}$ *has input* $m \in \{0,1\}^\ell$*, and parties obtain an output. Let* $\mathcal{A}$ *be an adversary characterized by adversary structure* $\mathcal{Z}$*.*

- **$\mathcal{Z}$-Liveness:** *All honest parties obtain some output.*
- **$\mathcal{Z}$-Validity:** *If* S *is honest, then every honest party with an output outputs* m*.*
- **$\mathcal{Z}$-Weak Validity:** *If* S *is honest, every honest party with an output outputs* m *or* $\bot$*.*
- **$\mathcal{Z}$-Consistency:** *If* S *is corrupt, every honest party with an output outputs a common value.*
- **$\mathcal{Z}$-Weak Consistency:** *If* S *is corrupt, every honest party with an output outputs a common* $m^\star \in \{0,1\}^\ell$ *or* $\bot$*.*

Π *is a $\mathcal{Z}$-perfectly-secure broadcast protocol if it has $\mathcal{Z}$-Liveness, $\mathcal{Z}$-Validity, and $\mathcal{Z}$-Consistency.*[13]

We give an overview of how to generalize the BoBW BA protocol of [2] and defer to the full version of the paper [4] for the details. The protocol is based on three components.

Component I: SBA with Asynchronous Guaranteed Liveness. We require a $\mathcal{Z}$-perfectly-secure SBA protocol Π_{SBA} with $\mathcal{Q}^{(3)}(\mathcal{P},\mathcal{Z})$ condition, which *also* provides $\mathcal{Z}$-guaranteed liveness in an *asynchronous* network. We design a candidate for Π_{SBA} by generalizing the simple SBA protocol of [11], which was designed to tolerate $t < n/3$ corruptions. The protocol requires at most $3n$ rounds in a *synchronous* network and hence, within time $T_{\mathsf{SBA}} \overset{def}{=} 3n \cdot \Delta$, all honest parties will get an output in a *synchronous* network. The protocol incurs a communication of $\mathcal{O}(n^3\ell)$ bits if the inputs of the parties are of size ℓ bits. To achieve $\mathcal{Z}$-guaranteed liveness in an *asynchronous* network, the parties can run Π_{SBA} till time T_{SBA}, and then output $\bot$ if no "valid" output is computed as per the protocol at the time T_{SBA}; see the full version of this paper [4] for the details.

Component II: ABA with Synchronous Guarantees. We deploy the ABA protocol Π_{ABA} of [17], where $\mathcal{Z}$ satisfies the $\mathcal{Q}^{(3)}(\mathcal{P},\mathcal{Z})$ condition and where each party has an input bit. The protocol has the following liveness guarantees in an *asynchronous* network.

- If the inputs of all *honest* parties are the same, then Π_{ABA} achieves $\mathcal{Z}$-guaranteed liveness. Else, Π_{ABA} achieves $\mathcal{Z}$-almost-surely liveness.

Protocol Π_{ABA} also achieves $\mathcal{Z}$-validity, $\mathcal{Z}$-consistency, and the following liveness guarantees in a *synchronous* network.

- If all *honest* parties have the same input, then Π_{ABA} achieves $\mathcal{Z}$-guaranteed liveness, and all honest parties obtain output within time $T_{\mathsf{ABA}} = k \cdot \Delta$, for some known constant k.
- Else, Π_{ABA} achieves $\mathcal{Z}$-almost-surely liveness and requires $\mathcal{O}(\text{poly}(n) \cdot \Delta)$ expected time.

[12] The *weak validity* and *weak consistency* properties are defined here for the sake of completeness. Looking ahead, our BoBW BA protocol will be using BA protocol(s) with these "weaker" properties.

[13] Similar to BA, the weak validity and consistency properties are defined here for the sake of completeness, since we will be designing a broadcast protocol with these weaker properties in our BoBW BA protocol.

Irrespective of the network type, Π_{ABA} incurs a communication of $\mathcal{O}(|\mathcal{Z}| \cdot n^5 \log |\mathbb{F}| + n^6 \log n)$ bits, if all honest parties have the same input bit. Else, it incurs an expected communication of $\mathcal{O}(|\mathcal{Z}| \cdot n^7 \log |\mathbb{F}| + n^8 \log n)$ bits. Here $\mathbb{F}$ is a finite field such that $|\mathbb{F}| > n$ holds.

Component III: Synchronous Broadcast with Asynchronous Guarantees. We assume the existence of a broadcast protocol Π_{BC}, which is a $\mathcal{Z}$-perfectly-secure broadcast protocol in a *synchronous* network, and which also provides $\mathcal{Z}$-Liveness, $\mathcal{Z}$-Weak Validity and $\mathcal{Z}$-Weak Consistency in an *asynchronous* network. We present a candidate for Π_{BC} by generalizing the broadcast protocol of [2] with similar guarantees. The protocol incurs a communication of $\mathcal{O}(n^3\ell)$ bits, where S participates with input $m \in \{0,1\}^\ell$. The idea is to carefully "stitch" together protocol Π_{ACast} with the protocol Π_{SBA}. In the protocol, all *honest* parties have some output at the (local) time $T_{\mathsf{BC}} = 3\Delta + T_{\mathsf{SBA}}$. Depending upon the network type and corruption status of S, the output is -

- *Synchronous Network and Honest* S*:* m for *all* honest parties.
- *Synchronous Network and Corrupt* S*:* a common $m^\star \in \{0,1\}^\ell \cup \{\perp\}$ for *all* honest parties.
- *Asynchronous Network and Honest* S*:* either m or $\perp$ for each honest party.
- *Asynchronous Network and Corrupt* S*:* a common $m^\star \in \{0,1\}^\ell$ or $\perp$ for each honest party.

Protocol Π_{BC} also gives the parties who output $\perp$ at time T_{BC} an option to switch their output to some ℓ-bit string if the parties keep running the protocol beyond time T_{BC} and if certain "conditions" are satisfied for those parties. We stress that this switching provision is *only* for those who output $\perp$ at time T_{BC}. While this provision is not "useful" and not used while designing BA, it comes in handy when Π_{BC} is used to broadcast values in our VSS protocol. Notice that the output-switching provision will *not* lead to a violation of consistency and hence honest parties will *not* end up with different ℓ-bit outputs. Following the terminology of [2], we call the process of computing output at time T_{BC} and beyond time T_{BC} as the *regular mode* and *fallback mode* of Π_{BC} respectively. We refer to Appendix A for the terminologies associated with the protocol Π_{BC}.

Π_{BC}+Π_{ABA} $\Rightarrow$ BoBW BA. We combine protocols Π_{BC} and Π_{ABA} to get Π_{BA} by generalizing the idea used in [2] against *threshold* adversaries. In the protocol, every party first broadcasts its input bit (for the BA protocol) through an instance of Π_{BC}. If the network is *synchronous*, then all honest parties should have received the inputs of all the (honest) sender parties from the corresponding broadcast instances through regular mode by time T_{BC}. Consequently, at time T_{BC}, the parties decide an output for *all* the n instances of Π_{BC}. Based on these outputs, the parties decide their respective inputs for the Π_{ABA} protocol. Specifically, if "sufficiently many" outputs from the Π_{BC} instances are found to be *same*, then the parties consider this output value as their input for the Π_{ABA} instance. Else, they stick to their original inputs. The overall output for Π_{BA} is then set to be the output from Π_{ABA}. For the formal description of Π_{BA} and the proof of Theorem 5, see [4].

▶ **Theorem 5.** *Let $\mathcal{Z}$ satisfy the $\mathcal{Q}^{(3)}(\mathcal{P}, \mathcal{Z})$ condition. Then Π_{BA} achieves the following.*

- *In a synchronous network, the protocol is a $\mathcal{Z}$-perfectly-secure SBA protocol, where all honest parties obtain an output within time $T_{\mathsf{BA}} = T_{\mathsf{BC}} + T_{\mathsf{ABA}}$. The protocol incurs a communication of $\mathcal{O}(|\mathcal{Z}| \cdot n^5 \log |\mathbb{F}| + n^6 \log n)$ bits.*
- *In an asynchronous network, the protocol is a $\mathcal{Z}$-perfectly-secure ABA protocol, with an expected communication of $\mathcal{O}(|\mathcal{Z}| \cdot n^7 \log |\mathbb{F}| + n^8 \log n)$ bits.*

Protocol $\Pi_{\mathsf{VSS}}(\mathsf{D}, s, \mathbb{S} = (S_1, \ldots, S_{|\mathcal{Z}_s|}))$

- **Phase I – Share Distribution:** D randomly selects $s^{(1)}, \ldots, s^{(|\mathcal{Z}_s|)} \in \mathbb{K}$ such that $s = s^{(1)} + \ldots + s^{(|\mathcal{Z}_s|)}$. For $m = 1, \ldots, |\mathcal{Z}_s|$, it then sends $s^{(m)}$ to every party in the set S_m.
- **Phase II – Pairwise Checks:** For $m = 1, \ldots, |\mathcal{Z}_s|$, each $P_i \in S_m$ does the following.
  - On receiving $s_i^{(m)}$ from D, wait till the local time is a multiple of Δ. Send $s_i^{(m)}$ to each $P_j \in S_m$.
  - On receiving $s_j^{(m)}$ from any $P_j \in S_m$, wait till the local time is a multiple of Δ. Do the following.
    - If a share $s_i^{(m)}$ corresponding to S_m has been received from D, then, broadcast $\texttt{OK}(m, i, j)$ if $s_i^{(m)} = s_j^{(m)}$ holds. Else, broadcast $\texttt{NOK}(m, i)$.
    - If $s_j^{(m)}$ and $s_k^{(m)}$ have been received from any P_j and P_k respectively, belonging to S_m such that $s_j^{(m)} \neq s_k^{(m)}$, then broadcast $\texttt{NOK}(m, i)$.
- **Local Computation – Constructing Consistency Graphs:** Each $P_i \in \mathcal{P}$ constructs undirected *consistency graphs* $G_i^{(1)}, \ldots, G_i^{(|\mathcal{Z}_s|)}$, where $G_i^{(m)}$ is over the parties in S_m and where the edge (P_j, P_k) is included in $G_i^{(m)}$ if P_i has received $\texttt{OK}(m, j, k)$ and $\texttt{OK}(m, k, j)$ from the broadcast of P_j and P_k respectively, either through regular or fallback mode.
- **Phase III – Resolving Complaints and Broadcasting Core Sets Based On $\mathcal{Z}_s$:** Each $P_i \in \mathcal{P}$ (including D) does the following at time $2\Delta + T_{\mathsf{BC}}$.
  - If $\texttt{NOK}(m, j)$ is received from the broadcast of any $P_j \in S_m$ through regular-mode corresponding to any $m \in \{1, \ldots, |\mathcal{Z}_s|\}$, then do the following:
    - **If** $P_i = \mathsf{D}$**:** Broadcast $\texttt{Resolve}(m, s^{(m)})$.
    - **If** $P_i \neq \mathsf{D}$**:** Broadcast $\texttt{Resolve}(m, s_i^{(m)})$, if $P_i \in S_m$ and P_i has received $s_i^{(m)}$ from D.
  - **(If** $P_i = \mathsf{D}$**):** For $m = 1, \ldots, |\mathcal{Z}_s|$, check if there exists a $\mathcal{C}_m \subseteq S_m$ which constitutes a clique in graph $G_{\mathsf{D}}^{(m)}$, such that $S_m \setminus \mathcal{C}_m \in \mathcal{Z}_s$. If $\mathcal{C}_1, \ldots, \mathcal{C}_{|\mathcal{Z}_s|}$ are found, then broadcast them.
- **Local Computation – Verifying and Accepting Core sets:** Each party $P_i \in \mathcal{P}$ (including D) does the following at time $2\Delta + 2T_{\mathsf{BC}}$.
  - If $\mathcal{C}_1, \ldots, \mathcal{C}_{|\mathcal{Z}_s|}$ are received from the broadcast of D through regular mode, *accept* these if:
    - For $m = 1, \ldots, |\mathcal{Z}_s|$, the set $\mathcal{C}_m$ constitutes a clique in the consistency graph $G_i^{(m)}$ at time $2\Delta + T_{\mathsf{BC}}$. In addition, $S_m \setminus \mathcal{C}_m \in \mathcal{Z}_s$.
    - For $m = 1, \ldots, |\mathcal{Z}_s|$, if $\texttt{NOK}(m, j)$ was received from the broadcast of any $P_j \in S_m$ through regular mode at time $2\Delta + T_{\mathsf{BC}}$, then the following must hold at time $2\Delta + 2T_{\mathsf{BC}}$.
      - $\texttt{Resolve}(m, s^{(m)})$ is received from the broadcast of D through regular-mode.
      - $\texttt{Resolve}(m, s^{(m)})$ is received from the broadcast of a set of parties $\mathcal{C}'_m$ through regular-mode, where $\mathcal{C}'_m \subseteq \mathcal{C}_m$, and $\mathcal{C}_m \setminus \mathcal{C}'_m \in \mathcal{Z}_s$.
- **Phase IV – Deciding Whether Core Sets Based on $\mathcal{Z}_s$ have Been Accepted by Any Honest Party:** At time $2\Delta + 2T_{\mathsf{BC}}$, each $P_i \in \mathcal{P}$ participates in an instance of Π_{BA} with input $b_i = 1$ if it has accepted sets $\mathcal{C}_1, \ldots, \mathcal{C}_{|\mathcal{Z}_s|}$, else, with input $b_i = 0$, and waits for time T_{BA}.

Figure 1 Best-of-both-worlds VSS protocol: Part I.

4 Best-of-Both-Worlds VSS Protocol

The goal of our BoBW VSS protocol (Fig 1 and Fig 2) is to enable a *dealer* $\mathsf{D} \in \mathcal{P}$ to "verifiably" generate a secret-sharing of its private input $s \in \mathbb{K}$ with respect to the specification $\mathbb{S} = \{S_m : S_m = \mathcal{P} \setminus Z_m \text{ and } Z_m \in \mathcal{Z}_s\}$, *irrespective* of the network type. An overview of the protocol has been given in Section 1. In the protocol, broadcast is instantiated through Π_{BC} with respect to $\mathcal{Z}_s$ (see the terminologies associated with Π_{BC} in Appendix A).

Theorem 6 states the properties of Π_{VSS} and is proven in the full version of the paper [4].

▶ **Theorem 6.** *Protocol* Π_{VSS} *achieves the following.*

Protocol $\Pi_{\mathsf{VSS}}(\mathsf{D}, s, \mathbb{S} = (S_1, \ldots, S_{|\mathcal{Z}_s|}))$ Contd ...

- **Local Computation – Computing Shares Through Core Sets Based on $\mathcal{Z}_s$:** If the output of Π_{BA} is 1, then each party $P_i \in \mathcal{P}$ does the following.
  - If $\mathcal{C}_1, \ldots, \mathcal{C}_{|\mathcal{Z}_s|}$ are not received yet, then wait to receive them from the broadcast of D. Then for $m = 1, \ldots, |\mathcal{Z}_s|$, compute the share $s_i^{(m)}$ corresponding to S_m as follows, if $P_i \in S_m$.
    - If at time $2\Delta + 2T_{\mathsf{BC}}$, $\texttt{Resolve}(m, s^{(m)})$ was received from D's broadcast and from a set of parties $\mathcal{C}'_m \subseteq \mathcal{C}_m$ through regular-mode, where $\mathcal{C}_m \setminus \mathcal{C}'_m \in \mathcal{Z}_s$, then output $s_i^{(m)} = s^{(m)}$.
    - Else, if a common value, say $s^{(m)}$, was received from a set of parties $\mathcal{C}''_m \subseteq \mathcal{C}_m$ at time 2Δ where $\mathcal{C}_m \setminus \mathcal{C}''_m \in \mathcal{Z}_s$, then output $s_i^{(m)} = s^{(m)}$.
    - Else wait till there exists a subset of parties $\mathcal{C}'''_m \subseteq \mathcal{C}_m$ where $\mathcal{C}_m \setminus \mathcal{C}'''_m \in \mathcal{Z}_a$, such that a common value, say $s^{(m)}$, is received from all the parties in $\mathcal{C}'''_m$. Output $s_i^{(m)} = s^{(m)}$.
- **Phase V – Broadcasting Core Sets Based on $\mathcal{Z}_a$:** If the output of Π_{BA} is 0, then for $m = 1, \ldots, |\mathcal{Z}_s|$, dealer D does the following in its graph $G_{\mathsf{D}}^{(m)}$.
  - Check if there exists a subset of parties $\mathcal{E}_m \subseteq S_m$, which constitutes a clique in the graph $G_{\mathsf{D}}^{(m)}$, such that $S_m \setminus \mathcal{E}_m \in \mathcal{Z}_a$. Upon finding $\mathcal{E}_1, \ldots, \mathcal{E}_{|\mathcal{Z}_s|}$, broadcast them.
- **Local Computation – Computing Shares Through Core Sets Based on $\mathcal{Z}_a$:** If the output of Π_{BA} is 0, then each party $P_i \in \mathcal{P}$ does the following.
  - Participate in any instance of Π_{BC} invoked by D for broadcasting $\mathcal{E}_1, \ldots, \mathcal{E}_{|\mathcal{Z}_s|}$, only after time $2\Delta + 2T_{\mathsf{BC}} + T_{\mathsf{BA}}$. Wait till $\mathcal{E}_1, \ldots, \mathcal{E}_{|\mathcal{Z}_s|}$ are received from the broadcast of D. Upon receiving, *accept* these sets if each set $\mathcal{E}_m$ constitutes a clique in the graph $G_i^{(m)}$ and $S_m \setminus \mathcal{E}_m \in \mathcal{Z}_a$. Upon accepting, compute the share $s_i^{(m)}$ corresponding to every S_m where $P_i \in S_m$ as follows.
    - If $P_i \in \mathcal{E}_m$, then output $s_i^{(m)}$ received from D.
    - Else, wait till there exists a subset $\mathcal{E}'_m \subseteq \mathcal{E}_m$, where $\mathcal{E}_m \setminus \mathcal{E}'_m \in \mathcal{Z}_s$, such that there exists a common value, say $s^{(m)}$, received from all the parties in $\mathcal{E}'_m$. Output $s_i^{(m)} = s^{(m)}$.

Figure 2 Best-of-both-worlds VSS protocol: Part II.

If D *is honest, then the following hold.*

- **$\mathcal{Z}_s$-correctness:** *In a synchronous network, s is secret-shared with respect to $\mathbb{S}$ at time* $T_{\mathsf{VSS}} = 2\Delta + 2T_{\mathsf{BC}} + T_{\mathsf{BA}}$.
- **$\mathcal{Z}_a$-correctness:** *In an asynchronous network, almost-surely, s is eventually secret-shared with respect to $\mathbb{S}$.*
- **Privacy:** *Adversary's view remains independent of s in any network.*

If D *is corrupt, either no honest party obtains an output or there exists an $s^\star \in \mathbb{K}$, such that:*

- **$\mathcal{Z}_a$-commitment:** *In an asynchronous network, almost-surely, $s^\star$ is eventually secret-shared with respect to $\mathbb{S}$.*
- **$\mathcal{Z}_s$-commitment:** *In a synchronous network, $s^\star$ is shared with respect to $\mathbb{S}$, such that:*
  - *If any honest party outputs its shares at time T_{VSS}, then all honest parties output their shares at time T_{VSS}.*
  - *If any honest party outputs its shares at time $T > T_{\mathsf{VSS}}$, then every honest party outputs its shares by time $T + 2\Delta$.*

Communication Complexity: *The protocol incurs a communication of $\mathcal{O}(|\mathcal{Z}_s| \cdot n^4 (\log |\mathbb{K}| + \log |\mathcal{Z}_s| + \log n) + n^5 \log n)$ bits, and invokes one instance of Π_{BA}.*

Π_{VSS} for L Secrets. We describe how D can share L secrets with just one instance of Π_{BA} in Appendix B.

5 The Preprocessing Phase Protocol

Our preprocessing phase allows the parties to generate secret-sharing of c_M multiplication-triples, which are random for the adversary and is based on two sub-protocols.[14]

Agreement on a Common Subset (ACS). In protocol Π_{ACS}, there exists a set $\mathcal{P}' \subseteq \mathcal{P}$ such that it will be *guaranteed* that $\mathcal{Z}_s$ and $\mathcal{Z}_a$ *either* satisfy the $\mathcal{Q}^{(1,1)}(\mathcal{P}', \mathcal{Z}_s, \mathcal{Z}_a)$ condition *or* $\mathcal{Q}^{(3,1)}(\mathcal{P}', \mathcal{Z}_s, \mathcal{Z}_a)$ condition [15]. Moreover, each party in $\mathcal{P}'$ will have L values, which it would like to secret-share using Π_{VSS}. As *corrupt* dealers might *not* invoke their instances of Π_{VSS}, the parties can compute outputs from *only* a subset of Π_{VSS} instances corresponding to parties $\mathcal{P}' \setminus Z$, for some $Z \in \mathcal{Z}_s$ (*even* in a *synchronous* network). However, in an *asynchronous* network, *different* parties may compute outputs from Π_{VSS} instances of *different* subsets of $\mathcal{P}' \setminus Z$ parties, corresponding to a *different* $Z \in \mathcal{Z}_s$. Protocol Π_{ACS} allows parties to agree on a *common* subset $\mathcal{CS}$ of parties, where $\mathcal{P}' \setminus \mathcal{CS} \in \mathcal{Z}_s$, such that *all* honest parties will be able to compute their outputs corresponding to the Π_{VSS} instances of the parties in $\mathcal{CS}$. Moreover, in a *synchronous* network, *all honest* parties from $\mathcal{P}'$ are guaranteed to be present in $\mathcal{CS}$.[16] Protocol Π_{ACS} is obtained by generalizing the ACS protocol of [2], which was designed for *threshold* adversaries. The idea is to run n instances of our BA protocol Π_{BA}, one for each party, and decide which of these Π_{VSS} instances will produce an output for everyone. However, we need to take special care to ensure that all honest parties are going to make it to $\mathcal{CS}$ in a *synchronous* network; see the full version of the paper [4] for the details.

The Multiplication Protocol. Protocol Π_{Mult} takes as input secret-shared pairs of values $\{([a^{(\ell)}], [b^{(\ell)}])\}_{\ell=1,\ldots,L}$, and securely generates $\{[c^{(\ell)}]\}_{\ell=1,\ldots,L}$, where $c^{(\ell)} = a^{(\ell)} \cdot b^{(\ell)}$. For simplicity, we discuss the idea when $L = 1$ (a brief overview of the protocol has already been presented in Section 1). Let $[a]$ and $[b]$ be the inputs to the protocol and the goal is to compute $[a \cdot b]$. The parties securely compute secret-shared summands $[a]_l \cdot [b]_m$ and then $[a \cdot b]$ can be computed locally from secret-shared summands $[a]_l \cdot [b]_m$, owing to the linearity property. A secret-sharing of the summand $[a]_l \cdot [b]_m$ is computed as follows: let $\mathcal{I}_{l,m} = S_l \cap S_m$. Then, *irrespective* of the network type, $\mathcal{I}_{l,m}$ is *bound* to have *at least* one honest party, since $\mathcal{Z}_s$ and $\mathcal{Z}_a$ satisfy the $\mathcal{Q}^{(1,1)}(\mathcal{I}_{l,m}, \mathcal{Z}_s, \mathcal{Z}_a)$ condition. Each party in $\mathcal{I}_{l,m}$ is asked to independently secret-share the summand $[a]_l \cdot [b]_m$ through an instance of Π_{VSS}. To avoid an indefinite wait, the parties agree on a common subset of parties $\mathcal{R}_{l,m}$ from $\mathcal{I}_{l,m}$, where $\mathcal{I}_{l,m} \setminus \mathcal{R}_{l,m} \in \mathcal{Z}_s$, who have shared some summand, such that $\mathcal{R}_{l,m}$ has at least one *honest* party, *irrespective* of the network type. For this, the parties execute an instance of the Π_{ACS} protocol. To check if any cheating has occurred, the parties check whether *all* the parties in $\mathcal{R}_{l,m}$ have shared the same "version" of the summand $[a]_l \cdot [b]_m$. Protocol Π_{Mult} and its properties are available in the full version of this paper [4].

The Preprocessing Phase Protocol. Protocol $\Pi_{\mathsf{PreProcessing}}$ has two stages. In the *first* stage, the parties securely generate secret-sharing of c_M pairs of random values, by running an instance of Π_{ACS}, where the input for each party will be c_M pairs of random values. In the *second* stage, a secret-sharing of the product of each pair is computed by executing Π_{Mult}. Protocol $\Pi_{\mathsf{PreProcessing}}$ and its properties are available in the full version of this paper [4].

[14] $([a], [b], [c])$ constitutes a multiplication triple, where $a, b \in \mathbb{K}$ and $c = a \cdot b$ holds.
[15] In our *preprocessing phase* protocol, $\mathcal{P}'$ will be $S_l \cap S_m$ corresponding to some $S_l, S_m \in \mathbb{S}$ and hence, the $\mathcal{P}'^{(1,1)}(\mathcal{Q}, \mathcal{Z}_s, \mathcal{Z}_a)$ condition will be satisfied. In our MPC protocol, $\mathcal{P}'$ will be $\mathcal{P}$ and hence the $\mathcal{Q}^{(3,1)}(\mathcal{P}', \mathcal{Z}_s, \mathcal{Z}_a)$ condition will be satisfied.
[16] This property will be crucial in a *synchronous* network.

6 Best-of-Both-Worlds Circuit-Evaluation Protocol

Protocol Π_{CirEval} for the circuit-evaluation consists of four phases. In the *first* phase, the parties generate secret-sharing of c_M random multiplication-triples through $\Pi_{\mathsf{PreProcessing}}$. Additionally, they invoke Π_{ACS} to generate secret-sharing of their respective inputs for the publicly known function f and agree on a *common* subset of parties $\mathcal{CS}$, where $\mathcal{P} \setminus \mathcal{CS} \in \mathcal{Z}_s$, such that the inputs of the parties in $\mathcal{CS}$ are secret-shared. The inputs of the remaining parties are set to 0. Note that in a *synchronous* network, *all honest* parties will be in $\mathcal{CS}$. In the *second* phase, the parties securely evaluate each gate in the circuit in a secret-shared fashion, after which the parties *publicly* reconstruct the secret-shared output in the *third* phase. The *last* phase is the *termination phase*, where the parties wait till "sufficiently many" parties have obtained the same output, after which they "safely" take that output and terminate the protocol (and all the underlying sub-protocols).

Π_{CirEval} and the proof of Theorem 7 are available in the full version of this paper [4].

▶ **Theorem 7.** *Let $\mathcal{A}$ be an adversary, characterized by adversary structures $\mathcal{Z}_s$ and $\mathcal{Z}_a$ in a synchronous and asynchronous network respectively, satisfying the conditions* Con *(see Condition 1 in Section 1). Moreover, let $f : \mathbb{K}^n \rightarrow \mathbb{K}$ be a function represented by an arithmetic circuit* cir *over $\mathbb{K}$, consisting of c_M number of multiplication gates, with a multiplicative depth of D_M, with each party having an input $x_i \in \mathbb{K}$. Then, Π_{CirEval} incurs a communication cost of $\mathcal{O}(c_M \cdot |\mathcal{Z}_s|^3 \cdot n^5(\log |\mathbb{K}| + \log |\mathcal{Z}_s| + \log n) + |\mathcal{Z}_s|^2 \cdot n^6 \log n)$ bits, invokes $\mathcal{O}(|\mathcal{Z}_s|^2 \cdot n)$ instances of Π_{BA}, and achieves the following for some $\mathcal{CS} \subseteq \mathcal{P}$.*

- *In a synchronous network, all honest parties output $y = f(x_1, \ldots, x_n)$ at time $(30n + D_M + 6k + 38) \cdot \Delta$, where $x_j = 0$ for every $P_j \notin \mathcal{CS}$, such that $\mathcal{P} \setminus \mathcal{CS} \in \mathcal{Z}_s$, and every honest party is present in $\mathcal{CS}$; here k is a constant determined by the protocol Π_{ABA}.*
- *In an asynchronous network, almost-surely, the honest parties eventually output $y = f(x_1, \ldots, x_n)$ where $x_j = 0$ for every $P_j \notin \mathcal{CS}$ and where $\mathcal{P} \setminus \mathcal{CS} \in \mathcal{Z}_s$.*
- *The view of $\mathcal{A}$ remains independent of the inputs of the honest parties in $\mathcal{CS}$.*

References

1 I. Abraham, D. Dolev, and J. Y. Halpern. An Almost-surely Terminating Polynomial Protocol for Asynchronous Byzantine Agreement with Optimal Resilience. In *PODC*, pages 405–414. ACM, 2008.

2 A. Appan, A. Chandramouli, and A. Choudhury. Perfectly-Secure Synchronous MPC with Asynchronous Fallback Guarantees. In *PODC*, pages 92–102. ACM, 2022.

3 A. Appan, A. Chandramouli, and A. Choudhury. Revisiting the Efficiency of Asynchronous Multi Party Computation Against General Adversaries. In *INDOCRYPT*, volume 13774 of *Lecture Notes in Computer Science*, pages 223–248. Springer International Publishing, 2022.

4 Ananya Appan, Anirudh Chandramouli, and Ashish Choudhury. Perfectly secure synchronous mpc with asynchronous fallback guarantees against general adversaries. Cryptology ePrint Archive, Paper 2022/1047, 2022. URL: `https://eprint.iacr.org/2022/1047`.

5 B. Applebaum, E. Kachlon, and A. Patra. The Round Complexity of Perfect MPC with Active Security and Optimal Resiliency. In *FOCS*, pages 1277–1284. IEEE, 2020.

6 R. Bacho, D. Collins, C. Liu-Zhang, and J. Loss. Network-Agnostic Security Comes for Free in DKG and MPC. Cryptology ePrint Archive, Paper 2022/1369, 2022.

7 L. Bangalore, A. Choudhury, and A. Patra. The Power of Shunning: Efficient Asynchronous Byzantine Agreement Revisited. *J. ACM*, 67(3):14:1–14:59, 2020.

8 D. Beaver. Efficient Multiparty Protocols Using Circuit Randomization. In J. Feigenbaum, editor, *CRYPTO*, volume 576 of *Lecture Notes in Computer Science*, pages 420–432. Springer, 1991.

9 M. Ben-Or, R. Canetti, and O. Goldreich. Asynchronous Secure Computation. In *STOC*, pages 52–61. ACM, 1993.

10 M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract). In *STOC*, pages 1–10. ACM, 1988.

11 P. Berman, J. A. Garay, and K. J. Perry. Towards Optimal Distributed Consensus (Extended Abstract). In *FOCS*, pages 410–415. IEEE Computer Society, 1989.

12 E. Blum, J. Katz, and J. Loss. Synchronous Consensus with Optimal Asynchronous Fallback Guarantees. In *TCC*, volume 11891 of *Lecture Notes in Computer Science*, pages 131–150. Springer, 2019.

13 E. Blum, J. Katz, and J. Loss. Tardigrade: An Atomic Broadcast Protocol for Arbitrary Network Conditions. In *ASIACRYPT*, volume 13091 of *Lecture Notes in Computer Science*, pages 547–572. Springer, 2021.

14 E. Blum, C. L. Zhang, and J. Loss. Always Have a Backup Plan: Fully Secure Synchronous MPC with Asynchronous Fallback. In *CRYPTO*, volume 12171 of *Lecture Notes in Computer Science*, pages 707–731. Springer, 2020.

15 D. Chaum, C. Crépeau, and I. Damgård. Multiparty Unconditionally Secure Protocols (Extended Abstract). In *STOC*, pages 11–19. ACM, 1988.

16 B. Chor, S. Goldwasser, S. Micali, and B. Awerbuch. Verifiable Secret Sharing and Achieving Simultaneity in the Presence of Faults (Extended Abstract). In *26th Annual Symposium on Foundations of Computer Science, Portland, Oregon, USA, 21-23 October 1985*, pages 383–395. IEEE Computer Society, 1985.

17 A. Choudhury. Almost-Surely Terminating Asynchronous Byzantine Agreement Against General Adversaries with Optimal Resilience. In *ICDCN*, pages 167–176. ACM, 2023.

18 A. Choudhury and N. Pappu. Perfectly-Secure Asynchronous MPC for General Adversaries (Extended Abstract). In *INDOCRYPT*, volume 12578 of *Lecture Notes in Computer Science*, pages 786–809. Springer, 2020.

19 A. Choudhury and A. Patra. An Efficient Framework for Unconditionally Secure Multiparty Computation. *IEEE Trans. Information Theory*, 63(1):428–468, 2017.

20 R. Cramer, I. Damgård, and U. M. Maurer. General Secure Multi-party Computation from any Linear Secret-Sharing Scheme. In *EUROCRYPT*, volume 1807 of *Lecture Notes in Computer Science*, pages 316–334. Springer Verlag, 2000.

21 G. Deligios, M. Hirt, and C. Liu-Zhang. Round-Efficient Byzantine Agreement and Multi-party Computation with Asynchronous Fallback. In *TCC*, volume 13042 of *Lecture Notes in Computer Science*, pages 623–653. Springer, 2021.

22 G. Deligios and C. Liu-Zhang. Synchronous Perfectly Secure Message Transmission with Optimal Asynchronous Fallback Guarantees. *IACR Cryptol. ePrint Arch.*, page 1397, 2022.

23 D. Dolev, C. Dwork, O. Waarts, and M. Yung. Perfectly Secure Message Transmission. *J. ACM*, 40(1):17–47, 1993.

24 M. J. Fischer, N. A. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32(2):374–382, 1985.

25 M. Fitzi and U. M. Maurer. Efficient Byzantine Agreement Secure Against General Adversaries. In *DISC*, volume 1499 of *Lecture Notes in Computer Science*, pages 134–148. Springer, 1998.

26 D. Ghinea, C. Liu-Zhang, and R. Wattenhofer. Optimal Synchronous Approximate Agreement with Asynchronous Fallback. In *PODC*, pages 70–80. ACM, 2022.

27 O. Goldreich, S. Micali, and A. Wigderson. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In A. V. Aho, editor, *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 218–229. ACM, 1987.

28 Martin Hirt and Ueli Maurer. Complete characterization of adversaries tolerable in secure multi-party computation (extended abstract). In *PODC*, pages 25–34. ACM, 1997.

29 Martin Hirt and Ueli Maurer. Player simulation and general adversary structures in perfect multiparty computation. *Journal of Cryptology*, 13(1):31–60, 2000.

30 Martin Hirt and Daniel Tschudi. Efficient general-adversary multi-party computation. In *ASIACRYPT*, volume 8270 of *Lecture Notes in Computer Science*, pages 181–200. Springer, 2013.

31 M. Ito, A. Saito, and T. Nishizeki. Secret Sharing Schemes Realizing General Access Structures). In *Global Telecommunication Conference, Globecom*, pages 99–102. IEEE Computer Society, 1987.

32 M. V. N. Ashwin Kumar, K. Srinathan, and C. Pandu Rangan. Asynchronous Perfectly Secure Computation Tolerating Generalized Adversaries. In *ACISP*, volume 2384 of *Lecture Notes in Computer Science*, pages 497–512. Springer, 2002.

33 K. Kursawe and F. C. Freiling. Byzantine Fault Tolerance on General Hybrid Adversary Structures. Technical Report, RWTH Aachen, 2005.

34 U. M. Maurer. Secure Multi-party Computation Made Simple. In *SCN*, volume 2576 of *Lecture Notes in Computer Science*, pages 14–28. Springer, 2002.

35 A. Momose and L. Ren. Multi-Threshold Byzantine Fault Tolerance. In *CCS*, pages 1686–1699. ACM, 2021.

36 A. Patra and D. Ravi. On the Power of Hybrid Networks in Multi-Party Computation. *IEEE Trans. Information Theory*, 64(6):4207–4227, 2018.

37 M. C. Pease, R. E. Shostak, and L. Lamport. Reaching Agreement in the Presence of Faults. *J. ACM*, 27(2):228–234, 1980.

38 T. Rabin and M. Ben-Or. Verifiable Secret Sharing and Multiparty Protocols with Honest Majority (Extended Abstract). In *STOC*, pages 73–85. ACM, 1989.

39 A. Shamir. How to Share a Secret. *Commun. ACM*, 22(11):612–613, 1979.

40 A. C. Yao. Protocols for Secure Computations (Extended Abstract). In *FOCS*, pages 160–164. IEEE Computer Society, 1982.

A Broadcast Protocols

A.1 Acast

The properties satsfied by protocol Π_{ACast} [33] in a synchronous and an asynchronous network are given in Lemma 8.

▶ **Lemma 8.** *Let $\mathcal{A}$ be an adversary characterized by an adversary structure $\mathcal{Z}$ satisfying the $\mathcal{Q}^{(3)}(\mathcal{P}, \mathcal{Z})$ condition. Then, for a sender S with input m, Π_{ACast} achieves the following in an asynchronous network.*

- **$\mathcal{Z}$-Liveness:** *If S is honest, then all honest parties eventually have an output.*
- **$\mathcal{Z}$-Validity:** *If S is honest, then each honest P_i with an output, outputs m.*
- **$\mathcal{Z}$-Consistency:** *If S is corrupt and some honest P_i outputs $m^\star$, then all honest parties eventually output $m^\star$.*

Π_{ACast} achieves the following in a synchronous network.

- **$\mathcal{Z}$-Liveness:** *If S is honest, then all honest parties obtain an output within time 3Δ.*
- **$\mathcal{Z}$-Validity:** *If S is honest, then every honest party with an output, outputs m.*
- **$\mathcal{Z}$-Consistency:** *If S is corrupt and some honest party outputs $m^\star$ at time T, then every honest P_i outputs $m^\star$ by the end of time $T + 2\Delta$.*

Communication Complexity: *$\mathcal{O}(n^2\ell)$ bits are communicated by the parties in total.*

A.2 Terminologies Associated with Π_{BC}

▶ **Terminology 9 (Terminologies for Π_{BC}).** *We say that P_i broadcasts m to mean that P_i invokes an instance of Π_{BC} as* S *with input m, and the parties participate in this instance. Similarly, we say that P_j receives m from the broadcast of P_i through regular-mode (resp. fallback-mode), to mean that P_j has the output m at time T_{BC} (resp. after time T_{BC}) during the instance of Π_{BC}.*

B VSS for sharing L secrets

To share L secrets, D can invoke L instances of Π_{VSS}. However, instead of computing and broadcasting $L \cdot |\mathcal{Z}_s|$ core sets, it can compute and broadcast only $|\mathcal{Z}_s|$ core sets, on the behalf of *all* the L instances of Π_{VSS}. The parties will need to execute a *single* instance of Π_{BA} to decide whether D has broadcasted valid core sets. The resultant protocol will incur a communication of $\mathcal{O}(L \cdot |\mathcal{Z}_s| \cdot n^4(\log |\mathbb{K}| + \log |\mathcal{Z}_s| + \log n) + n^5 \log n)$ bits and invokes one instance of Π_{BA}. To avoid repetition, we do not provide the formal details.

One Step Forward, One Step Back: FLP-Style Proofs and the Round-Reduction Technique for Colorless Tasks

Hagit Attiya
Department of Computer Science, Technion, Israel

Pierre Fraigniaud
IRIF – CNRS & Université Paris Cité, France

Ami Paz
LISN – CNRS & Université Paris-Saclay, France

Sergio Rajsbaum
IRIF, École Polytechnique and Instituto de Matemáticas, UNAM, Mexico

Abstract

The paper compares two generic techniques for deriving lower bounds and impossibility results in distributed computing. First, we prove a *speedup theorem* (a-la Brandt, 2019), for wait-free *colorless* algorithms, aiming at capturing the essence of the seminal *round-reduction* proof establishing a lower bound on the number of rounds for 3-coloring a cycle (Linial, 1992), and going by *backward induction*. Second, we consider *FLP-style* proofs, aiming at capturing the essence of the seminal consensus impossibility proof (Fischer, Lynch, and Paterson, 1985) and using *forward induction*.

We show that despite their very different natures, these two forms of proof are tightly connected. In particular, we show that for every colorless task Π, if there is a round-reduction proof establishing the impossibility of solving Π using wait-free colorless algorithms, then there is an FLP-style proof establishing the same impossibility. For 1-dimensional colorless tasks (for an arbitrarily number $n \geq 2$ of processes), we prove that the two proof techniques have exactly the *same* power, and more importantly, both are *complete*: if a 1-dimensional colorless task is *not* wait-free solvable by $n \geq 2$ processes, then the impossibility can be proved by both proof techniques. Moreover, a round-reduction proof can be *automatically* derived, and an FLP-style proof can be automatically generated from it.

Finally, we illustrate the use of these two techniques by establishing the impossibility of solving any colorless *covering* task of arbitrary dimension by wait-free algorithms.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Wait-free computing, lower bounds

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.4

Related Version *Full Version*: `http://arxiv.org/abs/2308.04213`

Funding *Hagit Attiya*: partially supported by the Israel Science Foundation (grants 380/18 and 22/1425).
Pierre Fraigniaud: partially supported by the ANR projects DUCAT (ANR-20-CE48-0006), FREDDA (ANR-17-CE40-0013), and QuData (ANR-18-CE47-0010).
Sergio Rajsbaum: partially supported by the ANR projects DUCAT (ANR-20-CE48-0006).

Acknowledgements The authors thank Faith Ellen and the referees for helpful comments.

1 Introduction

We analyze the relative power of two generic and versatile techniques for establishing lower bounds and impossibility results in asynchronous distributed computing. We focus on solving *tasks*, defined as triples $\Pi = (\mathcal{I}, \mathcal{O}, \Delta)$, where processes start with initial input values defined by $\mathcal{I}$, and decide irrevocably on output values allowed by $\mathcal{O}$ after communicating with each other for some number of steps, respecting the input/output relation Δ; the sets of processes, input values and output values are all finite.

This paper concentrates on the family of *colorless* tasks, including consensus, set agreement [17], loop agreement [25], and various robot and graph agreement tasks [3, 16]. A colorless task is defined only in terms of input and output values, regardless of the number of processes involved, and regardless of which process has a particular input or output value; accordingly, $\mathcal{I}$ and $\mathcal{O}$ consist of sets of values, without process ids. For instance, in the binary consensus task, $\mathcal{I} = \big\{\{0\}, \{1\}, \{0, 1\}\big\}$, meaning that all processes may start with input 0, or all processes may start with input 1, or some processes may start with input 0 while others may start with input 1. In the same task, $\mathcal{O} = \big\{\{0\}, \{1\}\big\}$, meaning that the only valid output configurations are when all processes output 0, or all processes output 1. Finally, consensus specification is captured by $\Delta(\{0\}) = \{0\}$, $\Delta(\{1\}) = \{1\}$, and $\Delta(\{0, 1\}) = \big\{\{0\}, \{1\}\big\}$, meaning that, if there was an initial agreement between the input values then the processes must stick to this agreement and output their input values, and otherwise they are allow to output either 0 or 1, as long as they all agree on the same value.

Colorless tasks are simpler to analyze than general tasks, such as symmetry breaking tasks [15], but they are still undecidable [24]. They have an elegant computability characterization [23, 28], essentially stating that a colorless task is wait-free solvable if and only if there is a continuous map from the geometric realization of $\mathcal{I}$ to that of $\mathcal{O}$ that respects Δ. One major interest of colorless tasks is that, whenever solvable, they can be solved by simple algorithms, referred to as *colorless algorithms* [28] (see also [23, Ch. 4]). Roughly speaking, such algorithms ignore multiplicities of input values and process states, and manipulate only *sets* of values; in contrast, general algorithms manipulate *vectors* of values and take into account which processes possess which value.

The two lower-bound or impossibility techniques at the core of our work are *FLP-style proofs*, named after Fischer, Lynch and Paterson [19], and *round-reduction proofs*, whose first occurrence might be attributed to Linial [33]. The former offers a form of *forward* induction technique, while the latter offers a form of *backward* induction, and they both present some form of *locality.* We recall these two techniques hereafter.

1.1 Stepping Forward: FLP-Style Impossibility Proofs

The celebrated FLP proof technique [19] is perhaps the most used technique for proving impossibility results in distributed computing. It has been used to prove the impossibility of solving consensus and several other problems [6, 18, 34], as well as to derive lower bounds [1, 6, 31]. To prove that a task $\Pi = (\mathcal{I}, \mathcal{O}, \Delta)$ is not wait-free solvable, the FLP technique considers a hypothetical algorithm ALG solving the task, and constructs an infinite sequence $\sigma_0, \sigma_1, \ldots$ of system configurations such that, for every $i \geq 0$,

- σ_{i+1} is a successor of σ_i, and
- ALG cannot output in σ_i.

To initiate this sequence, the prover is allowed to ask the *valencies* of all initial configurations $\sigma \in \mathcal{I}$, where the valency of a configuration σ is the set of values that are output by ALG in all executions starting from σ. Based on these valencies, the prover selects an initial configuration $\sigma_0 \in \mathcal{I}$. Then, given a configuration σ_i, the prover asks the algorithm for the valencies of some successor configurations of σ_i, and one of these configurations is selected to be the next configuration σ_{i+1} in the sequence, and so on. This is a form of *forward induction*, starting with σ_0 and constructing a sequence of configurations one after the other.

If ALG actually solves the task Π, then the prover will fail to construct an infinite sequence, merely because ALG can reveal the actual valencies that it produces for each given configuration. On the other hand, if for every algorithm ALG hypothetically solving Π the prover successfully constructs an infinite sequence, then this establishes that Π is not solvable because the sequence is constructed in such a way that ALG cannot output in σ_i. For instance, the FLP impossibility proof for consensus [19] constructs such an infinite sequence $\sigma_0, \sigma_1, \ldots$ for every algorithm ALG by establishing that (1) there exists an initial *bivalent* configuration σ_0 (i.e., a configuration from which an execution of ALG leads all processes to output 0, and another execution of ALG leads all processes to output 1), and (2) for every bivalent configuration σ_i, there exists a bivalent one-step successor σ_{i+1} of σ_i that is bivalent.

An FLP-style impossibility proof is a simple case of extension-based impossibility proofs [2] and *local proofs* [5], hence if such a proof exists for a task Π then there are also extension-based and local impossibility proofs for it. All these techniques use types of *valency-arguments* [6, Chapter 7], in the sense that they consider output values in executions starting from a given configuration, and then decide on the next configuration. All these techniques work for consensus but fail for set agreement (i.e., set agreement is not solvable, but this impossibility cannot be established by valency arguments), and the exact set of tasks for which each of them applies is not known.

1.2 Stepping Back: Round-Reduction Impossibility Proofs

The round-reduction proof technique in distributed computing can be traced back to the seminal work of Linial [33] establishing a lower bound for coloring the n-node cycle C_n using a (failure-free) synchronous algorithm. In a nutshell, he proved that for every $t \geq 1$, if there exists a t-round algorithm ALG producing a proper k-coloring of C_n, then there exists a $(t-1)$-round algorithm ALG$'$ producing a proper 2^{2^k}-coloring of C_n. Repeating this argument for roughly $\frac{1}{2}\log^* n$ times implies that if there is a $(\frac{1}{2}\log^* n)$-round algorithm for 3-coloring C_n then there exists a 0-round algorithm for $(n-1)$-coloring it, which is impossible. Here, $\log^\star n$ denotes the number of times one should apply the $\log_2$ function to get from n to a value smaller than 1.

This technique was generalized as a *speedup theorem* by Brandt [13]. Such a theorem is based on establishing the existence of a map F transforming any task Π in some class $\mathcal{T}$ of tasks into another task $\Pi' \in \mathcal{T}$ such that, for every $t \geq 1$, if Π is solvable in t rounds by an algorithm ALG from some class $\mathcal{A}$ of algorithms, then $F(\Pi)$ is solvable in $t-1$ rounds by an algorithm ALG$' \in \mathcal{A}$. Whenever such a theorem can be established, we get that for every task $\Pi \in \mathcal{T}$, and for every $t \geq 0$, if the task $F^{(t)}(\Pi)$ obtained by iterating t times F on Π is not solvable in zero rounds by an algorithm in $\mathcal{A}$, then Π is not solvable in t rounds by an algorithm in $\mathcal{A}$, which provides a lower bound on the complexity of Π. In particular, if $F^{(t)}(\Pi)$ is not solvable in zero rounds by an algorithm in $\mathcal{A}$ for all $t \geq 0$, then Π cannot be solved by an algorithm in $\mathcal{A}$. We refer to this technique as a *round-reduction* impossibility or and lower bound proof. In contrast with the forward induction approach of FLP-style proofs, round-reduction is an a-posteriori technique, starting by assuming an algorithm solves a problem in t rounds, claiming another problem is solvable in $t-1$ rounds, and repeating this argument down to 0 rounds; we hence refer to it as backward induction.

A speedup theorem has been established in [13] for solving locally-checkable labeling (LCL) tasks [36] using algorithms running in the anonymous LOCAL model [37]. A speedup theorem has also been established in [20], but for general (colored) tasks and wait-free algorithms running in the *iterated immediate snapshot* (IIS) model [7]. The transformations F_{LOCAL} and F_{IIS} used in [13] and [20] respectively, are of very different natures. Nevertheless, both enabled to establish lower bounds for various tasks, including sink-less orientation and maximal independent set (MIS) in the LOCAL model (see [8,13]), and approximate agreement in the IIS model, and even when the IIS model is enhanced with powerful objects like test&set (see [20]).

Our first contribution is a speedup theorem for wait-free colorless algorithms solving colorless tasks in the IIS model. It is important to note that, although the set of colorless tasks is a subset of the set of general tasks, the speedup theorem in [20] does not imply our speedup theorem, since the transformation F_{IIS} used in [20] does not apply to colorless *algorithms*. Specifically, if Π is solvable in t rounds by n processes running a wait-free colorless algorithm, then $F_{\text{IIS}}(\Pi)$ is indeed solvable in $t-1$ rounds by n processes, but running a wait-free algorithm that *may not be colorless*. As a consequence, the theorem cannot be iterated, which ruins the ability to design a round-reduction proof for colorless tasks. The speedup theorem for colorless tasks we present here uses a transformation that preserves solvability by colorless algorithms. The transformation Cl we define (Definition 4) has the following properties, as shown in Theorem 6 and Theorem 11. Applications of this theorem to approximate agreement and covering tasks can be found in Section 7.

▶ **Theorem A.** *For every $n \geq 2$ and every $t \geq 1$, the transformation* Cl *maps any colorless task Π to a colorless task* $\mathsf{Cl}(\Pi)$ *such that, when considering n processes running a colorless wait-free algorithm in the* IIS *model:*

- *If Π is 1-dimensional, then Π is solvable in t rounds if and only if $\mathsf{Cl}(\Pi)$ is solvable in $t-1$ rounds;*
- *Regardless of the dimension of Π, if Π is solvable in t rounds then $\mathsf{Cl}(\Pi)$ is solvable in $t-1$ rounds.*

A round-reduction impossibility or lower bound proof derived from Theorem A essentially proceeds by computing $\mathsf{Cl}^{(t)}(\Pi)$, and checking whether $\mathsf{Cl}^{(t)}(\Pi)$ is solvable in zero rounds. If the answer is negative for some $t \geq 1$, the proof successfully shows that Π cannot be solved in t rounds, and if it is negative for all $t \geq 1$, the proof shows Π is not solvable; otherwise, the proof fails. While Cl is not the only possible round-reduction operator for colorless tasks, we focus solely on it in this work.

In the full version of this paper, we illustrate the fact that round-reduction does not extend in a straightforward manner to all classes of algorithms. To this end, we consider *comparison-based* algorithms, an important class of algorithms used for studying tasks such as renaming and weak symmetry-breaking. We show that a closure operator similar to the ones considered here and in [20] but restricted to comparison-based algorithms, does not suffice for deriving a speedup theorem for comparison-based algorithms.

1.3 Round-Reduction vs. FLP-Style Impossibility Proofs

Interestingly, as for extension-based proofs, the round-reduction proofs derived from the speedup theorem in [20] work for consensus but fail for set-agreement and the same holds for our transformation Cl. We next show why the fact that both transformations succeed for binary consensus but fail for set-agreement should not come as a surprise, in light of prior results about FLP-style proofs.

Our second contribution shows that, although round-reduction proofs and FLP-style proofs may appear very different in nature, their power in term of establishing impossibility results can be compared. We actually show that the FLP-style proof technique is at least as strong as the round-reduction proof technique.

▶ **Theorem B** (Theorem 13). *For every colorless task Π and $n \geq 2$, if there is a round-reduction proof establishing the impossibility of solving Π by n processes running a wait-free colorless algorithm, then there is an FLP-style proof establishing the same impossibility.*

So, in particular, the fact that there is no round-reduction proof for the impossibility of solving set-agreement wait-free should not come as a surprise, since it is known that set-agreement has no extension-based impossibility proof [2, 5, 14], which is a proof technique at least as strong as FLP-style proofs. Yet, the situation is a bit subtle here: the results in [2, 5] rule out the existence of an FLP-style impossibility proof for solving set-agreement using general algorithms, and [14] proves a similar result for anonymous algorithms (where processes see a multi-set of the values stored in the memory and not a vector), but these works do not necessarily rule out the existence of an FLP-style impossibility proof for solving set agreement using colorless algorithms. The restriction to colorless algorithm allows the claimed algorithm ALG less flexibility regarding the valencies it announces to the prover, so an FLP-style impossibility proof when restricting ALG to be colorless is not ruled out by the previous results.

Nevertheless, we were able to show that, for *1-dimensional* colorless tasks, round-reduction proofs and FLP-style proofs have exactly the same power (Corollary 14). Recall that, in 1-dimensional colorless tasks, $\mathcal{I}$ and $\mathcal{O}$ are graphs, i.e., the n processes can start with at most two different input values and output at most two different values. This family includes binary consensus, approximate agreement with binary inputs, and the colorless version of wait-free checkable tasks [21], but not set-agreement. 1-dimensional colorless tasks are well studied, and they are known to be wait-free solvable if and only if they are 1-resilient solvable, both in the read/write model (with at least two processes) and in the message-passing model (with at least three processes) [9, 12, 27]. In fact, we not only show equivalence between round-reduction and FLP-style proof techniques for 1-dimensional colorless tasks, but we show that both are complete for these tasks (Corollary 12 for the round-reduction technique and Corollary 15 for FLP-style proofs). Hence, if a 1-dimensional colorless task is not wait-free solvable by $n \geq 2$ processes, then this impossibility can be proved by both proof techniques.

▶ **Theorem C.** *The round-reduction and FLP-style proof techniques are both complete for 1-dimensional colorless tasks and wait-free colorless algorithms.*

Theorem C follows from the fact that, for 1-dimensional colorless tasks (and colorless algorithms), our speedup theorem (Theorem A) also provides a *necessary* condition, that is, for every 1-dimensional colorless task Π, every $n \geq 2$, and every $t \geq 1$, Π is solvable in t rounds by n processes running a wait-free colorless algorithm in the IIS model *if and only if* $\mathsf{Cl}(\Pi)$ is solvable in $t-1$ rounds by n processes running a wait-free colorless algorithm in the IIS model. This if-and-only-if condition provides a mechanical way for deciding whether a given 1-dimensional colorless task Π is solvable. Indeed, the transformation Cl used in Theorem A has a desirable property: it preserves the number of input and output values (i.e., the vertices in $\mathcal{I}$ and $\mathcal{O}$), and it may just potentially add some combinations of output values that were not legal in Π but become legal in $\mathsf{Cl}(\Pi)$. As a consequence, iterating Cl starting from Π necessarily leads to a fixed point Π^* for Cl, i.e., $\mathsf{Cl}(\Pi^*) = \Pi^*$, after a bounded number of iterations. It follows that a 1-dimensional colorless task Π is wait-free solvable in the IIS model if and only if Π^* is wait-free solvable in zero rounds, which is decidable. The completeness of the FLP-style proof technique follows. Indeed, if a 1-dimensional colored task Π is not wait-free solvable, then Π has a round-reduction impossibility proof (by computing the fixed point Π^*, and showing that Π^* is not solvable in zero rounds), from which it follows, thanks to Theorem B, that Π has an FLP-style impossibility proof.

1.4 Applications

We illustrate the concepts and results introduced in this paper by applying them to the vast class of *covering tasks*, whose colored version was introduced and studied in [21] under the name *locality-preserving* tasks and further studied in [41]. To get the intuition of such a tasks, it is easier to consider 1-dimensional covering tasks, for which $\mathcal{I}$ and $\mathcal{O}$ are graphs. Recall that for two connected simple (i.e., no self-loops nor multiple edges) graphs G and H, and a function $f : V(H) \to V(G)$, the pair (H, f) is a covering of G if f is an homorphism (i.e., it preserves edges) and, for every $v \in V(H)$, the restriction of f to $N_H[v]$ is a one-to-one mapping $f : N_H[v] \to N_G[f(v)]$. For instance, for $C_3 = (v_0, v_1, v_2)$, $C_6 = (u_0, \dots, u_5)$, and $f : V(C_6) \to V(C_3)$ defined as $f(u_i) = v_{i \bmod 3}$, (C_6, f) is a covering of C_3. A covering $(\mathcal{O}, f)$ of $\mathcal{I}$ induces a task $\Pi = (\mathcal{I}, \mathcal{O}, \Delta)$ where Δ is essentially defined as f^{-1}. For higher dimensional colorless tasks, $\mathcal{I}$ and $\mathcal{O}$ are connected simplicial complexes, and f must be simplicial, but the general idea is the same [39]. A covering $(\mathcal{O}, f)$ of $\mathcal{I}$ is trivial if $\mathcal{I}$ and $\mathcal{O}$ are isomorphic. It is known that no non-trivial covering tasks can be solved wait-free in the IIS model [21]. Here, we consider a colorless version of covering tasks, and study impossibility proofs for solving them using colorless algorithms.

▶ **Theorem D.** *Every non-trivial covering task admits an FLP-style impossibility proof for $n \geq 2$ processes running wait-free colorless algorithms.*

The proof is based on showing that, for every covering task Π, the transformation F used in our speedup theorem satisfies $F(\Pi) = \Pi$, i.e., Π is itself a fixed point for F. As a consequence, since Π is not solvable in zero rounds (unless $\mathcal{I}$ and $\mathcal{O}$ are isomorphic, i.e., the task Π is trivial), there is a round-reduction impossibility proof for Π (Theorem 19), and the existence of an FLP-style impossibility proof for Π then follows from Theorem B. This last fact is of particular interest, as it shows that FLP-style proofs are not limited to cases where $\mathcal{O}$ is disconnected or to problems that are unsolvable even when restricted to a single input simplex and its faces (as is the case for consensus and approximate agreement).

2 Model and Definitions

We consider the wait-free iterated immediate snapshots model (IIS). This model and its variants have been frequently used e.g. [11,32,38] due to its simplicity, while being equivalent to the usual wait-free read/write shared memory model for task solvability [11,22]. Furthermore, it is known that as far as colorless task solvability is concerned, one can assume *colorless computation* without loss of generality [23, 28], by which we mean that in each round of computation processes do no consider which process wrote a value, nor by how many processes it was written. We next provide a brief overview of the model.

Processes communicate through a sequence of shared memory objects, and the computation is split into rounds. In the i-th round, process p writes a value to the p-th location of the i-th object, and then takes a snapshot of all the i-th object. The *view* of a process at the end of a round is the set of values it read from the object, without the information of which

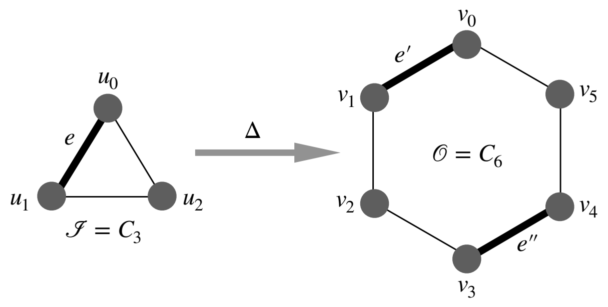


Figure 1 The Hexagone Task. In particular, $\Delta(e)$ is the complex $e' \cup e''$.

process wrote which value, nor by how many processes it was written to the memory. As common in lower bound proofs, we only consider full information protocols: in each round, each process writes its entire state (or equivalently, the view read from the previous object) to the memory.

2.1 Colorless Tasks

In this paper we consider colorless tasks [35]. See [23, Chapter 4] and [28] for an overview.

▶ **Definition 1.** *A* colorless *task* $\Pi = (\mathcal{I}, \mathcal{O}, \Delta)$ *is defined by an input simplicial complex* $\mathcal{I}$*, an output simplicial complex* $\mathcal{O}$*, and an input-output specification* $\Delta : \mathcal{I} \to 2^{\mathcal{O}}$ *mapping every simplex* $\sigma \in \mathcal{I}$ *to a sub-complex* $\Delta(\sigma)$ *of* $\mathcal{O}$ *with dimension at most* $\dim(\sigma)$*.*

The semantics of a colorless task $\Pi = (\mathcal{I}, \mathcal{O}, \Delta)$ is that every vertex of $\mathcal{I}$ is an input value, and every vertex of $\mathcal{O}$ is an output value. A set of processes may start with different input values in $V(\mathcal{I})$, as long as the set σ of these input values belongs to $\mathcal{I}$. To solve the task, it is required that any collection of processes starting with input values forming a set $\sigma \in \mathcal{I}$ outputs only output values in the vertices $V(\mathcal{O})$, as long as the set τ of these output values forms a simplex in $\Delta(\sigma)$.

The next two colorless tasks will serve as running examples in the rest of the paper.

The Set Agreement Task. Set agreement is a well studied relaxation of the classical consensus task. Let $k \geq 2$. *Set-agreement* with input set $[k] = \{1, \ldots, k\}$ is the colorless task $\mathrm{SA}_k = (\mathcal{I}, \mathcal{O}, \Delta)$ where $\mathcal{I} = \{\sigma \subseteq [k] \mid \sigma \neq \varnothing\}$, $\mathcal{O} = \{\tau \subseteq [k] \mid (\tau \neq \varnothing) \wedge (\tau \neq [k])\}$, and, for every $\sigma \in \mathcal{I}$,

$$\Delta(\sigma) = \{\tau \in \mathcal{O} \mid \tau \subseteq \sigma\}.$$

Said differently, $\Delta(\sigma) = \{\tau \subseteq \sigma\}$ if $\dim(\sigma) < k-1$, and $\Delta(\sigma) = \mathcal{O}$ otherwise. SA_k is solvable (in zero rounds) for $n < k$ processes, but not solvable for $n \geq k$ processes [10, 29, 40].

The Hexagone Task. The Hexagone Task is one basic example of the colorless *covering* tasks, thoroughly studied in Section 7.2. For every $t \geq 3$, let C_t denotes the t-node cycle. The *hexagone task* is the task $\mathrm{HX} = (C_3, C_6, \Delta)$ where $C_3 = (u_0, u_1, u_2)$, $C_6 = (v_0, v_1, \ldots, v_5)$ and Δ is defined as follows (see Figure 1). For every $i \in \{0, 1, 2\}$,

$$\Delta(u_i) = \{\{v_i\}, \{v_{i+3}\}\} \text{ and } \Delta(\{u_i, u_{i+1 \bmod 3}\}) = \{\{v_i, v_{i+1}\}, \{v_{i+3}, v_{i+4 \bmod 6}\}\},$$

where formally $\Delta(\{u_i, u_{i+1 \bmod 3}\})$ also contains all the vertices contained it its edges, i.e. $\{v_i\}, \{v_{i+1}\}, \{v_{i+3}\}, \{v_{i+4 \bmod 6}\}$. The map Δ is the inverse of the maps $f : V(C_6) \to V(C_3)$ defined as $f(u_i) = v_{i \bmod 3}$, where (C_6, f) is a covering of C_3.

2.2 Colorless Algorithms

The solvability of a colorless task may depend on the number n of processes involved in the computation. Remarkably, it is enough to consider *colorless computation* for colorless tasks, according to the following result (see, e.g., [28]), that summarizes and formalizes what we need to know about the model of computation[1]. The result holds both for the wait-free read/write memor

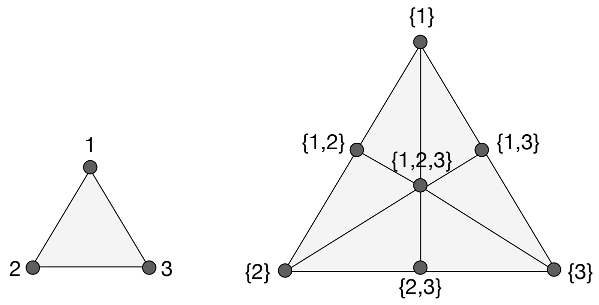


Figure 2 Barycentric subdivision.

▶ **Lemma 2.** *A colorless task $\Pi = (\mathcal{I}, \mathcal{O}, \Delta)$ is read/write solvable by n processes running a wait-free algorithm if and only if there exists $t \geq 0$ and a simplicial map*

$$f : \mathsf{Bary}^{(t)}(\mathsf{Skel}_n(\mathcal{I})) \to \mathcal{O}$$

that agrees with Δ, i.e., for every $\sigma \in \mathcal{I}$, $f(\mathsf{Bary}^{(t)}(\mathsf{Skel}_n(\sigma))) \subseteq \Delta(\sigma)$. Furthermore, Π is 1-round solvable by n processes running a wait-free colorless algorithm in the IIS model if and only if there is a simplicial map

$$f : \mathsf{Bary}(\mathsf{Skel}_n(\mathcal{I})) \to \mathcal{O}$$

that agrees with Δ.

In the above, $\mathsf{Bary}^{(t)}$ denotes t successive applications of the barycentric subdivision (see Fig. 2), and Skel_n denotes the $(n-1)$-dimensional skeleton operator, i.e., $\mathsf{Skel}_n(\mathcal{I})$ is the subcomplex of $\mathcal{I}$ (resp., of σ) including all simplices of $\mathcal{I}$ (resp., all faces of σ) with dimension at most $n-1$. When considering colored (general) tasks, a similar result holds when using chromatic simplicial maps and the *standard chromatic subdivision*; for colorless tasks, however, the barycentric subdivision suffices since if a colorless task is solvable then it is solvable by an algorithm ignoring multiplicities of identical views (or inputs) in the snapshots [28].

[1] Results similar to this one are known for several models of computation [23]. For instance, a corresponding result is known for Byzantine failures [23, Theorem 6.5.1] and dependent failures [26, Theorem 4.3] with an adversary of core size $c = 2$. For dependent failures in the case of wait-free computation, the theorem states that a colorless task is solvable if and only if there exists a continuous map between geometric realization $f : \mathsf{Skel}_{c-1}(\mathcal{I}) \to \mathcal{O}$ carried by Δ.

3 Round-Reduction Proofs

This section is essentially dedicated to the proof of Theorem A. The task transformation Cl in this theorem is based on a *closure* operator similar to the one in [20], where the main difference is that our closure operator is required to be colorless, and we stress that our result is not implied by the (colored) theorem of [20]. That is, given a class $\mathcal{A}$ of algorithms, requiring the closure operator to be in the class may not be sufficient for extending the speedup theorem in [20] to apply to algorithms in $\mathcal{A}$. We illustrate this in the full version of the paper for *comparison-based* algorithms, an important class of algorithms used for studying tasks such as renaming and weak symmetry-breaking. Using a comparison-based closure operator does not suffice for deriving a speedup theorem for comparison-based algorithms. On the other hand, in this section we show that for colorless algorithms, restricting the closure operator to be colorless suffices.

3.1 Colorless Closure

We first rephrase the notions of local tasks introduced in [20] in the context of colorless tasks.

▶ **Definition 3.** *Let $\Pi = (\mathcal{I}, \mathcal{O}, \Delta)$ be a colorless task, let $\sigma \in \mathcal{I}$, and let $\tau \subseteq V(\Delta(\sigma))$. Let us consider τ as a simplicial complex (with a unique facet). The* local task *with respect to σ and τ is the colorless task $\Pi_{\tau,\sigma} = (\tau, \Delta(\sigma), \Delta_{\tau,\sigma})$ where, for every face τ' of τ,*

$$\Delta_{\tau,\sigma}(\tau') = \begin{cases} v & \text{if } \tau' = v \text{ is a vertex,} \\ \mathsf{Skel}_{\dim(\tau')}(\Delta(\sigma)) & \text{otherwise.} \end{cases}$$

Note that $\Pi_{\tau,\sigma}$ is a well-defined colorless task, because $\mathsf{Skel}_{\dim(\tau')}$ guarantees that the output complex for τ' has dimension at most $\dim(\tau')$; this is true even if τ is not a simplex of $\Delta(\sigma)$, as seen in the second example in Section 3.1. Note also that the validity constraint of $\Pi_{\tau,\sigma}$ is just that if all processes start with the same input v, i.e., if they start from the same vertex $v \in V(\tau)$, then they must all output v. Without this constraint, the processes are only constrained to output values forming a legal set σ' of outputs w.r.t. σ, i.e., a set $\sigma' \in \Delta(\sigma)$ (and $\dim(\sigma') \leq \dim(\tau')$).

As opposed to the general (chromatic) tasks, which each has a fixed maximum number of participating processes, colorless tasks are defined for any number of processes. In particular, a colorless task may be solvable for a certain number of processes, but not for another number of processes. The definition below refers to solving local tasks with a prescribed number of processes.

▶ **Definition 4.** *The* colorless closure *of a colorless task $\Pi = (\mathcal{I}, \mathcal{O}, \Delta)$ is the colorless task $\mathsf{Cl}(\Pi) = (\mathcal{I}, \mathcal{O}', \Delta')$ where $V(\mathcal{O}') = V(\mathcal{O})$, and, for every $\sigma \in \mathcal{I}$, and every non-empty set $\tau \subseteq V(\mathcal{O})$, we set $\tau \in \Delta'(\sigma)$ if $\tau \subseteq V(\Delta(\sigma))$ and $\Pi_{\tau,\sigma}$ is solvable in one round by $\dim(\tau) + 1$ processes running a colorless algorithm. The simplices of $\mathcal{O}'$ are the images of Δ', and all their faces.*

Note that this definition is constructive, i.e., one can check the 1-round solvability of $\Pi_{\tau,\sigma}$, by Lemma 2. The closure operator also has the nice property that it does not change the allowed input and output values, and the only difference between Π and $\mathsf{Cl}(\Pi)$ is in the addition of some allowed combinations of output values.

For a simplex $\tau \in \Delta(\sigma)$, the local task $\Pi_{\tau,\sigma}$ is solvable in 0 rounds, by having each process starting with input $v \in V(\tau)$ output v. It follows that if $\tau \in \Delta(\sigma)$ then $\tau \in \Delta'(\sigma)$, and therefore, for every $\sigma \in \mathcal{I}$, $\Delta(\sigma) \subseteq \Delta'(\sigma)$. As a consequence, the colorless closure $\mathsf{Cl}(\Pi)$ of a task Π is not more difficult to solve than Π. Whether or not $\mathsf{Cl}(\Pi)$ is *simpler* to solve than Π is one of the foci of this paper. Before going further, we establish hereafter that the input-output specification Δ' of the colorless closure of a colorless task is a *carrier* map, which is a condition that is often required for a task [23].

The I/O-Specification of a Colorless Closure is a Carrier Map. Let $\Pi = (\mathcal{I}, \mathcal{O}, \Delta)$ be a colorless task. Recall that Δ is a *carrier* map if, for every σ and σ' in $\mathcal{I}$, $\sigma \subseteq \sigma' \implies \Delta(\sigma) \subseteq \Delta(\sigma')$, where the latter inclusion must be read as $\Delta(\sigma)$ is a subcomplex of $\Delta(\sigma')$. Being a carrier map is not necessary for Π to be solvable. However, if two simplices σ and σ' in $\mathcal{I}$ satisfy $\sigma \subseteq \sigma'$ and $\Delta(\sigma) \smallsetminus \Delta(\sigma') \neq \varnothing$, the output values outside $\Delta(\sigma')$ cannot be used for a set of processes starting with input σ' since these output values cannot be extended in case processes with inputs in $\sigma' \smallsetminus \sigma$ eventually participate later. Therefore, we may as well remove all simplices from $\Delta(\sigma)$ that are not in $\Delta(\sigma')$, and restrict ourselves to input-output specifications that are carrier maps.

▶ **Lemma 5.** *Let $\Pi = (\mathcal{I}, \mathcal{O}, \Delta)$ be a colorless task, and let $\mathsf{Cl}(\Pi) = (\mathcal{I}, \mathcal{O}', \Delta')$. If Δ is a carrier map then Δ' is a carrier map.*

Proof. Consider two simplices $\sigma, \sigma' \in \mathcal{I}$ satisfying $\sigma \subseteq \sigma'$, and let $\tau \in \Delta'(\sigma)$. By definition, the local task $\Pi_{\tau,\sigma} = (\tau, \Delta(\sigma), \Delta_{\tau,\sigma})$ is solvable in one round, by a simplicial map $g : \mathsf{Bary}^{(1)}(\tau) \to \Delta(\sigma)$ that agrees with $\Delta_{\tau,\sigma}$. To show that $\tau \in \Delta'(\sigma')$, we show that the local task $\Pi_{\tau,\sigma'} = (\tau, \Delta(\sigma'), \Delta_{\tau,\sigma'})$ is solvable in one round, using the same map g. Since $\Delta(\sigma) \subseteq \Delta(\sigma')$, g is a simplicial map from $\mathsf{Bary}^{(1)}(\tau)$ to $\Delta(\sigma')$. To see that g agrees with $\Delta_{\tau,\sigma'}$, let $\tau' \subseteq \tau$. If $\dim(\tau') > 0$, then $g(\tau') \in \Delta_{\tau,\sigma}(\tau') = \Delta(\sigma) \subseteq \Delta(\sigma') = \Delta_{\tau,\sigma'}(\tau')$. If $\tau' = \{v\}$ is a vertex, then $g(v) \in \Delta_{\tau,\sigma}(v) = v = \Delta_{\tau,\sigma'}(v)$. It follows that g agrees with $\Delta'_{\tau,\sigma'}$, and thus $\tau \in \Delta'(\sigma')$, from which we conclude that $\Delta'(\sigma) \subseteq \Delta'(\sigma')$, i.e., Δ' is a carrier map. ◀

Examples

- Let $k \geq 3$. For the set-agreement task SA_k, the closure is solvable in zero rounds by having each process outputting its input, since any combination of at most k input values forms a valid output configuration of $\mathsf{Cl}(\mathrm{SA}_k)$. To prove this, we show that for every $\sigma \in \mathcal{I}$ we have $\sigma \in \Delta'(\sigma)$. First, note that for $\sigma \in \mathcal{I}$ such that $\sigma \neq [k]$, we have $\sigma \in \Delta(\sigma)$ and $\sigma \in \Delta(\sigma) \Rightarrow \sigma \in \Delta'(\sigma)$, so we only have to show $[k] \in \Delta'([k])$. This is true since the local task $\Pi_{[k],[k]}$ is solvable in one round, by letting each process that sees more than one value output 1.
- On the other hand, for the Hexagon task HX we have $\mathsf{Cl}(\mathrm{HX}) = \mathrm{HX}$. To see this, consider an input simplex $\sigma = (\{u_i, u_{i+1 \bmod 3}\})$, for some $i \in \{0,1,2\}$, its image $\Delta(\{u_i, u_{i+1 \bmod 3}\}) = \{v_i, v_{i+1}\} \cup \{v_{i+3}, v_{i+4 \bmod 6}\}$, and two vertices in its image that do not already constitute a simplex, i.e. $w \in \{v_i, v_{i+1}\}$ and $w' \in \{v_{i+3}, v_{i+4 \bmod 6}\}$. Let $\tau = \{w, w'\}$.
  If the local task $\Pi_{\tau,\sigma} = (\tau, \Delta(\sigma), \Delta_{\tau,\sigma})$ would have been solvable in one round, then there would have been a map $f : V(\mathsf{Bary}^{(1)}(\tau)) \to \Delta(\sigma)$ satisfying $f(\{w\}) = w$ and $f(\{w'\}) = w'$. But $\mathsf{Bary}^{(1)}(\{w, w'\})$ is

  $$\{w\} \;\rule{2em}{1pt}\; \{w, w'\} \;\rule{2em}{1pt}\; \{w'\}$$

  and any such map cannot be simplicial by continuity: if $f(\{w, w'\}) \in \{v_i, v_{i+1}\}$ then $\{f(\{w'\}), f(\{w, w'\})\} \notin \Delta(\sigma)$, and similarly if $f(\{w, w'\}) \in \{v_{i+3}, v_{i+4 \bmod 6}\}$ then $\{f(\{w\}), f(\{w, w'\})\} \notin \Delta(\sigma)$.

3.2 Colorless Speedup Theorem

We now establish our speedup theorem for colorless algorithms, as stated next and proved in the full version of the paper.

▶ **Theorem 6.** *For every colorless task $\Pi = (\mathcal{I}, \mathcal{O}, \Delta)$, every $t > 0$, and every $n \geq 2$, if Π is solvable in t rounds by n processes running a wait-free colorless algorithm then the closure $\mathsf{Cl}(\Pi)$ is solvable in $t - 1$ rounds by n processes running a wait-free colorless algorithm.*

Since the colorless closure task $\mathsf{Cl}(\Pi) = (\mathcal{I}, \mathcal{O}', \Delta')$ of a colorless task $\Pi = (\mathcal{I}, \mathcal{O}, \Delta)$ only potentially adds valid output simplices to $\Delta(\sigma)$ for forming $\Delta'(\sigma)$, for every $\sigma \in \mathcal{I}$, it follows that, after applying the closure operator for some finite number of times t, we get a fixed point, i.e. a task $\mathsf{Cl}^{(t)}(\Pi)$ such that $\mathsf{Cl}^{(t+1)}(\Pi) = \mathsf{Cl}^{(t)}(\Pi)$. Naturally, this also implies $\mathsf{Cl}^{(t')}(\Pi) = \mathsf{Cl}^{(t)}(\Pi)$ for every $t' \geq t$.

▶ **Definition 7.** *The* fixed-point *of a colorless task Π is the task $\Pi^* = (\mathcal{I}, \mathcal{O}^*, \Delta^*)$ such that $\Pi^* = \mathsf{Cl}^{(t)}(\Pi)$ for some $t \geq 0$, and $\mathsf{Cl}^{(t+1)}(\Pi) = \mathsf{Cl}^{(t)}(\Pi)$.*

As a direct consequence of the speedup theorem (Theorem 6) the fixed-point task Π^* of a task Π is either 0-round solvable, or not solvable at all. Indeed, consider $\Pi^* = (\mathcal{I}, \mathcal{O}^*, \Delta^*)$ and assume it is solvable in $t > 0$ rounds. By Theorem 6, $\Pi^* = \mathsf{Cl}(\Pi^*)$ is solvable in $t - 1$ rounds. Repeating this argument for t times implies that Π^* is 0-round solvable.

▶ **Lemma 8.** *Let $n \geq 2$. Given a colorless task $\Pi = (\mathcal{I}, \mathcal{O}, \Delta)$, its fixed-point Π^* is either 0-round solvable by n processes running a wait-free colorless algorithm, or not solvable at all.*

The next corollary illustrates both the Theorem 6's interest and its simplicity.

▶ **Corollary 9.** *For every $n \geq 2$, the hexagon task cannot be solved by n processes running a wait-free colorless algorithm.*

Proof. If HX is solvable wait-free by $n \geq 2$ processes in the IIS model, then there exists $t \geq 0$ such that HX is solvable in t rounds. We have seen that $\mathsf{Cl}(\mathrm{HX}) = \mathrm{HX}$. It follows from Theorem 6 that if HX is solvable wait-free, then it is solvable wait-free in zero rounds.

Consider a possible zero-round algorithm for HX with $n \geq 2$ processes, and its decision map δ. As the algorithm must produce valid outputs for executions with a single input, we have $\delta(u_i) \in \{v_i, v_{i+3}\}$ for every $i \in \{0, 1, 2\}$. Let $v_j = \delta(u_0)$ (and hence $j \in \{0, 3\}$). The definition of Δ for executions with two different inputs guarantees $\delta(v_1) \in \{u_j, u_{j+1}\}$ and by the above we have $\delta(v_1) = u_{j+1}$. Similarly, $\delta(v_2) = u_{j+2}$, and hence $\delta\{v_0, v_2\} = \{u_j, u_{j+2}\} \notin \Delta\{v_0, v_2\}$, a contradiction. ◀

4 The Topology of the Closure

In this section, we study the topology of the colorless closure task, which, as opposed to the general closure, displays very simple properties. For stating these properties, let us recall that a complex $\mathcal{K}$ is *complete* if $V(\mathcal{K})$ is a simplex of $\mathcal{K}$. It is *complete up to dimension d* if every set $\tau \subseteq V(\mathcal{K})$ with $0 \leq \dim(\tau) \leq d$ satisfies $\tau \in \mathcal{K}$. A *connected component* of a complex $\mathcal{K}$ is the subcomplex of $\mathcal{K}$ induced by all the vertices in a connected component of the graph $\mathsf{Skel}_1(\mathcal{K})$. We mainly show that, for every $\sigma \in \mathcal{I}$, the closure of each connected component of $\Delta(\sigma)$ eventually becomes complete after a bounded number of closure operations.

For a colorless task $\Pi = (\mathcal{I}, \mathcal{O}, \Delta)$, we denote by $\mathsf{Cl}^{(k)}(\Pi) = (\mathcal{I}, \mathcal{O}^{(k)}, \Delta^{(k)})$ the k-th closure of Π, defined as $\mathsf{Cl}^{(k)}(\Pi) = \mathsf{Cl}(\mathsf{Cl}^{(k-1)}(\Pi))$, where k is a positive integer and $\mathsf{Cl}^{(0)}(\Pi) = \Pi$.

▶ **Theorem 10.** *Given a colorless task $\Pi = (\mathcal{I}, \mathcal{O}, \Delta)$ and a simplex $\sigma \in \mathcal{I}$, the following hold.*

- *Let D be the largest diameter of a connected component in the graph $\mathsf{Skel}_1(\Delta(\sigma))$, and let $\ell = \lceil \log_2 D \rceil + 1$. Then all the connected components of $\Delta^{(\ell)}(\sigma)$ are complete up to dimension $\dim(\sigma)$.*
- *For every $k \geq 0$, a set of vertices in $V(\Delta(\sigma))$ is a connected component of $\mathsf{Skel}_1(\Delta(\sigma))$ if and only if it is a connected component of $\mathsf{Skel}_1(\Delta^{(k)}(\sigma))$.*

See Appendix A for the proof of this theorem.

Examples

- In Section 3.1 we have proved that the closure of k-set agreement contains any combination of at most k input values. This fact can now be directly derived from Theorem 10, as $\Delta([k])$ is connected and contains all the values of $[k]$.
- In the same section, we have proved that $\mathsf{Cl}(\mathrm{HX}) = \mathrm{HX}$. This is also a direct consequence of Theorem 10, as for every $\sigma \in \mathcal{I}$, each of the connected components of $\Delta(\sigma)$ is full.

5 Round-Reduction is Complete for 1-Dimensional Tasks

1-dimensional tasks are tasks for which the input and output complexes are of dimension at most 1. In this section, we establish the completeness of the round-reduction proof technique for 1-dimensional colorless tasks, thus establishing part of Theorem C. FLP-style proofs are also complete in this case, but the proof of this fact is deferred to the next section, where we show that the techniques are equivalent. The next theorem asserts that for colorless tasks of dimension at most 1 the reciprocal of Theorem 6 holds as well, thus establishing the completeness of the round-reduction technique in this case; see Appendix A for the proof.

▶ **Theorem 11.** *For every 1-dimensional colorless task $\Pi = (\mathcal{I}, \mathcal{O}, \Delta)$, every $t > 0$, and every $n \geq 2$, if the colorless closure $\mathsf{Cl}(\Pi)$ is solvable in $t-1$ rounds by n processes running a wait-free colorless algorithm, then Π is solvable in t rounds by n processes running a wait-free colorless algorithm.*

By combining Theorems 6 and 11, we obtain the desired result.

▶ **Corollary 12.** *The round-reduction proof techniques is complete for 1-dimensional colorless tasks and wait-free colorless algorithms.*

6 Relations Between Round-Reduction and FLP-Style Proofs

In this section we establish tight connections between the FLP-style proof strategy and the round-reduction proof technique, in the context of wait-free solvability of colorless tasks. Specifically, we first establish Theorem B which asserts that the existence of a round-reduction proof implies the existence of an FLP-style proof, from which the remaining part of Theorem C (completeness of FLP-style proofs for 1-dimensional tasks) will follow. This implies that the round-reduction techniques and FLP-style proofs have the same power when considering impossibility proofs for 1-dimensional colorless tasks. In Theorem 16, we give a direct proof for this: the existence of an FLP-style proof implies the existence of a round-reduction proof for the impossibility of such tasks (complementing Theorem B, for 1-dimensional tasks). This direct proof may suggest that a more tight connection between the two proof techniques exists. We start by formally defining FLP-style proofs.

6.1 FLP-Style Proofs

Given a colorless task $\Pi = (\mathcal{I}, \mathcal{O}, \Delta)$, an FLP-style proof constructs an infinite sequence of simplices $\sigma_0, \sigma_1, \ldots$, where $\sigma_0 \in \mathcal{I} = \mathsf{Bary}^{(0)}(\mathcal{I})$, and, for every $t \geq 1$, $\sigma_t \in \mathsf{Bary}^{(1)}(\sigma_{t-1}) \subseteq \mathsf{Bary}^{(t)}(\mathcal{I})$, as follows.

The proof P assumes for contradiction the existence of an algorithm A solving Π. P starts by asking A to reveal, for every $\sigma \in \mathcal{I}$, the *valency* of σ, that is, the set of output values that are returned by A in executions starting with the input values forming the simplex σ. Then, P chooses a simplex $\sigma_0 \in \mathcal{I}$; in order to create an infinite sequence, P must choose σ_0 such that A is not able to claim it terminates in 0 rounds, i.e., such that any possible assignment of outputs to the processes in σ_0 is inconsistent with the valencies of $\mathcal{I}$.

Given a sequence $\sigma_0, \ldots, \sigma_t$ constructed by P so far, σ_{t+1} is obtained analogously, as follows. P asks A to reveal the valencies of all simplices in $\mathsf{Bary}^{(1)}(\sigma_t)$, that is, for each $\sigma \in \mathsf{Bary}^{(1)}(\sigma_t)$, the set of output values produced by A in all valid executions starting from σ. Based on these valencies, P chooses one simplex $\sigma_{t+1} \in \mathsf{Bary}^{(1)}(\sigma_t)$; as in the choice of σ_0, it must choose σ_{t+1} such that A is not able to assign outputs to the processes in σ_{t+1} consistent with the valencies of $\mathsf{Bary}^{(1)}(\sigma_t)$.

Let $\mathsf{val} : 2^{\sigma_0} \cup 2^{\sigma_1} \cup \cdots \to 2^{\mathcal{O}}$ be the function defined by the valencies returned by A. For any correct algorithm, the function val must satisfy some basic conditions of consistency with the task specification and with other valencies returned. We next formalize these conditions for σ_t.

- Consistent with itself: each $\sigma \in \mathsf{Bary}^{(1)}(\sigma_t)$ satisfies $\mathsf{val}(\sigma) \subseteq \mathsf{val}(\sigma_t)$; moreover, $\cup_{\sigma \in \mathsf{Bary}^{(1)}(\sigma_t)} \mathsf{val}(\sigma) = \mathsf{val}(\sigma_t)$.
- Consistent with Δ: For each $\sigma \in \mathsf{Bary}^{(1)}(\sigma_t)$, $\mathsf{val}(\sigma) \subseteq \Delta(\sigma_0)$.
- Monotone: for each $\sigma' \subseteq \sigma \in \mathsf{Bary}^{(1)}(\sigma_t)$, $\mathsf{val}(\sigma') \subseteq \mathsf{val}(\sigma)$.

If this strategy can proceed forever, constructing an infinite sequence $\sigma_0, \sigma_1, \ldots$ of simplices, then A does not terminate in this execution, disproving the existence of an algorithm A solving Π. At the core of FLP-style proofs stands a *choice mechanism* that picks the next simplex σ_{t+1}. We present such a mechanism for one-dimensional colorless tasks.

Our work on the FLP-style technique continues recent lines of work regarding the power of extension-based proofs [2, 5, 14]. We allow less diverse queries compared to these works, yet our results imply that if a one-dimensional colorless task is unsolvable, then the simple queries we allow are sufficient for proving this impossibility.

Example. The impossibility of solving the hexagon task can be proved using an FLP-style proof, by constructing a sequence $\sigma_0, \sigma_1, \ldots$ as follows. All the simplices $\sigma_0, \sigma_1, \ldots$ will be edges, and we will fix $i \in \{0, 1, 2\}$ such that each σ_t satisfies the invariants

(1) $\mathsf{val}(\sigma_t) \subseteq \{v_i, v_{i+1}\} \cup \{v_{i+3}, v_{i+4}\}$, and

(2) $\mathsf{val}(\sigma_t) \cap \{v_i, v_{i+1}\} \neq \emptyset$ and $\mathsf{val}(\sigma_t) \cap \{v_{i+3}, v_{i+4}\} \neq \emptyset$,

where the indices here and below are computed modulo 6, unless otherwise specified.

To choose σ_0 and i, inspect the valencies of the three edges $(u_i, u_{i+1 \bmod 3})$, for $i \in \{0, 1, 2\}$. Since $\mathsf{val}(u_i) \subseteq \mathsf{val}(u_{i-1 \bmod 3}, u_i) \cap \mathsf{val}(u_i, u_{i+1 \bmod 3})$ and valency is monotone, the valencies of every two edges must intersect, and thus for some $i \in \{0, 1, 2\}$ the edge $e = (u_i, u_{i+1 \bmod 3})$ must satisfy both $\mathsf{val}(e) \cap \{v_i, v_{i+1}\} \neq \emptyset$ and $\mathsf{val}(e) \cap \{v_{i+3}, v_{i+4}\} \neq \emptyset$; we fix this i, and set $\sigma_0 = e$, guaranteeing (2). Invariant (1) holds since the valency must be consistent with Δ.

Assume $\sigma_0, \ldots, \sigma_t$ are chosen and satisfy both invariants, and that $\mathsf{val}(\sigma)$ is known for each $\sigma \in \mathsf{Bary}_1(\sigma_t)$. Since $\mathsf{val}(\sigma) \subseteq \mathsf{val}(\sigma_t)$ for every $\sigma \in \mathsf{Bary}_1(\sigma_t)$, Invariant (1) will hold for any $\sigma_{t+1} \in \mathsf{Bary}_1(\sigma_t)$ we may choose. Let $\sigma_t = \{w_0, w_1\}$, then $\mathsf{Bary}_1(\sigma_t)$ is composed of the vertices $\{w_0\}, \{w_0, w_1\}, \{w_1\}$ and the edges $e_0 = (\{w_0\}, \{w_0, w_1\})$ and $e_1 = (\{w_1\}, \{w_0, w_1\})$.

We have $\mathsf{val}(e_0) \cap \mathsf{val}(e_1) \neq \emptyset$, and by Invariant (1) we also have $\mathsf{val}(e_0) \cap \mathsf{val}(e_1) \subseteq \mathsf{val}(\sigma_t) \subseteq \{v_i, v_{i+1}, v_{i+3}, v_{i+4}\}$, so at least one value $v \in \{v_i, v_{i+1}, v_{i+3}, v_{i+4}\}$ satisfies $v \in \mathsf{val}(e_0) \cap \mathsf{val}(e_1)$; assume without loss of generality that $v \in \{v_i, v_{i+1}\}$.

The valencies of the vertices are contained in the valencies of the edges, hence $\mathsf{val}(e_0) \cup \mathsf{val}(e_1) = \mathsf{val}(\sigma_t)$, so Invariant (2) implies that both $(\mathsf{val}(e_0) \cup \mathsf{val}(e_1)) \cap \{v_i, v_{i+1}\} \neq \emptyset$ and $(\mathsf{val}(e_0) \cup \mathsf{val}(e_1)) \cap \{v_{i+3}, v_{i+4}\} \neq \emptyset$ hold. Hence, at least one edge $e \in \{e_0, e_1\}$ has $\mathsf{val}(e) \cap \{v_{i+3}, v_{i+4}\} \neq \emptyset$, and since $v \in \mathsf{val}(e)$ it also have $\mathsf{val}(e) \cap \{v_i, v_{i+1}\} \neq \emptyset$. This edge is set as σ_{t+1}, and Invariant (2) is satisfied for σ_{t+1} as well. Since this process can continue for every $t \geq 0$, the proof is complete.

6.2 Connections Between the Proof Techniques

The next theorem, Theorem B, states that the existance of a round reduction impossibility proof implies the existence of an FLP-style proof. The theorem is proved in Appendix A.

▶ **Theorem 13.** *For every colorless task Π and $n \geq 2$, if there is a round-reduction proof establishing the impossibility of solving Π by n processes running a wait-free colorless algorithm, then there is an FLP-style proof establishing the same impossibility.*

We next prove Theorem C for FLP-style proofs, i.e. we show these are complete for 1-dimensional colorless tasks. For this, first observe that if there is an FLP-style proof for the impossibility of a 1-dimensional colorless task Π, then Π is unsolvable. By the completeness of the round-reduction proofs (cf. Corollary 12), there is a round-reduction impossibility proof for that task. On the other hand, Theorem 13 asserts that if a colorless task has a round-reduction impossibility proof then it also has an FLP-style impossibility proof. The establishes the desired equivalence between the two forms of proofs for colorless tasks.

▶ **Corollary 14.** *Let Π be a 1-dimensional colorless task. There is an FLP-style proof for the impossibility of solving Π using wait-free colorless algorithms if and only if there is a round-reduction proof of this impossibility for Π.*

This corollary can also be proved directly: one direction is Theorem 13 (for any dimension), and the other is given in Theorem 16 (below). We get that, for 1-dimensional tasks, the FLP-style proof style can be mechanized, i.e., for any 1-dimensional task Π, the FLP-style proof technique succeeds for Π if and only if Π is not wait-free solvable in the IIS model.

▶ **Corollary 15.** *The FLP-style proof technique is complete for 1-dimensional tasks.*

Proof. By Theorem 13, if the FLP-style proof technique fails for Π, then the round-reduction proof also fails for Π as well. By Lemma 8, this implies that Π^* is 0-round solvable. By Theorem 11, the original task Π is solvable. ◀

In the full version of this paper, we also give a direct proof for the converse of Theorem 13 for 1-dimensional tasks, stated next.

▶ **Theorem 16.** *For every 1-dimensional colorless task Π and $n \geq 2$, if there is an FLP-style proof for the impossibility of solving Π by n processes running a wait-free colorless algorithm, then there is a round-reduction proof for the same impossibility.*

7 Applications

Throughout this paper, we have shown how the theory we developed applies for set agreement and the Hexagon task. We complete the paper by presenting some further applications. Note that in terms of techniques, all these proofs are completely different from previous proofs of similar results: round-reduction works directly on the task specification, in an algorithmic way that does not depend on the specific task at hand. Hence, the arguments in round-reduction proofs are applied directly on the task specification, and not on executions of a protocol (which are encapsulated in the round-reduction theorem).

7.1 Time Lower Bound for Approximate Agreement

Let us show that the bound $\lceil \log_2 D \rceil$ in Theorem 10 is tight. For this purpose, consider the approximate agreement task. For an integer $N \geq 1$, let $\epsilon = 1/N$, and the ϵ-agreement task defined as follows. The input complex $\mathcal{I}$ of ϵ-agreement is merely the edge

$$0 \ \rule[0.5ex]{3em}{2pt}\ 1$$

The output complex $\mathcal{O}$ is the path

$$0 \ \rule[0.5ex]{3em}{2pt}\ \epsilon \ \rule[0.5ex]{3em}{2pt}\ 2\epsilon \ \rule[0.5ex]{3em}{2pt}\ \dots \ \rule[0.5ex]{3em}{2pt}\ (N-1)\epsilon \ \rule[0.5ex]{3em}{2pt}\ 1$$

Finally, the input-output specification Δ satisfies for each set $S \subseteq \mathcal{I}$

$$\Delta(S) = \{T \subseteq \mathcal{O} \mid \min S \leq \min T \text{ and } \max T \leq \max S\}$$

and specifically, for an element $x \in \mathcal{I}$ it specifies $\Delta(\{x\}) = \{x\}$.

▶ **Proposition 17.** *For every $\epsilon \in (0,1)$, ϵ-agreement cannot be solved by $n \geq 2$ processes in less than $\lceil \log_2 1/\epsilon \rceil$ rounds.*

The proof of this preposition appears in the appendix. Note that for $n > 2$ processes this bound is tight, and is the same for colored and colorless algorithm. Interestingly, for $n = 2$ processes there is a colored algorithm requiring only $\lceil \log_3 1/\epsilon \rceil$ rounds [4, 30]. Hence, while colorless and colored algorithms have the same computability power, colored algorithms are provably stronger in terms of time complexity.

7.2 Impossibility of Covering Tasks

Recall that, for two connected simplicial complexes $\mathcal{I}$ and $\mathcal{O}$, and for a simplicial map $f : \mathcal{O} \to \mathcal{I}$, the pair $(\mathcal{O}, f)$ is a *covering complex* of $\mathcal{I}$ if, for every $\sigma \in \mathcal{I}$, $f^{-1}(\sigma)$ is a union of pairwise disjoint simplexes. This condition can be rephrased as $f^{-1}(\sigma) = \cup_{i=1}^{k} \tau_i$ with $f_{|\tau_i} : \tau_i \to \sigma$ is one-one. The simplexes τ_i, $i = 1, \dots, k$, are called the *sheets* of σ. We often refer to f as a *covering map*. The following observations follow directly from the definition of covering complex (see, e.g., [39]).

- If $\sigma \in \mathcal{I}$ is a simplex of dimension d, each sheet τ_i of σ is also a simplex of dimension d.
- The two complexes $\mathcal{I}$ and $\mathcal{O}$ are *locally isomorphic*, in the sense that for each vertex $v \in \mathcal{O}$ the complex $\text{star}(v)$ is isomorphic to the complex $\text{star}(f(v))$. (The star of a vertex v in a complex $\mathcal{K}$ is the complex $\text{star}(v)$ consisting of all the simplexes of $\mathcal{K}$ that contain v.
- All the simplices in $\mathcal{O}$ have the same number of sheets.)

We define below a colorless variant of the chromatic covering tasks introduced in [21].

▶ **Definition 18.** *Given a covering complex $(\mathcal{O}, f)$ of a complex $\mathcal{I}$, the colorless* covering task *$(\mathcal{I}, \mathcal{O}, \Delta)$ is the task where Δ is defined, for every $\sigma \in \mathcal{I}$, by*

$$\Delta(\sigma) = \{\tau \in \mathcal{O} \mid f(\tau) \subseteq \sigma\},$$

where $f(\tau) \subseteq \sigma$ means that $f(\tau)$ is a sub-complex of the complex defined by σ and all its faces. A covering complex is non-trivial *if each simplex in $\mathcal{I}$ has more than one sheet.*

The Hexagone task discussed above is a basic example of a covering task, and another example of a covering task can be see in Figure 4 in the appendix.

We now turn to a general impossibility result for covering tasks, proved in Appendix A.

▶ **Theorem 19.** *No non-trivial colorless covering tasks can be solved by $n \geq 2$ processes running a wait-free colorless algorithm.*

A similar result was proved in the past using an ad-hoc argument [21] for colored covering tasks, and here we give a round-reduction based proof for colorless covering tasks. By Theorem 13, this also means that the claim has an FLP-style proof, proving Theorem D.

8 Conclusion

The purpose of this paper is to relate round-reduction proof techniques (formally stated in the Speedup Theorem) and FLP-style proof techniques, when applied to colorless tasks within the framework of wait-free computing in the IIS model.

The round-reduction technique offers many good features, including the fact that it is mechanical (it is sufficient to check whether the fixed-point closure is solvable in zero rounds), it enables to derive not only impossibility results but also complexity lower bounds (e.g., for approximate agreement), and it extends to wait-free computing in models stronger than IIS (e.g., IIS augmented with Test&Set objects). On the other hand, FLP-style proofs are very generic, and essentially apply to all models, including t-resilient models. Moreover, we have shown that FLP-style proofs are not weaker than round-reduction proofs, and it is possible that they are stronger. Nevertheless, we have also shown that for 1-dimensional colorless tasks the two techniques have exactly the same power, and are both complete in the sense that if a task is not solvable then any of the two techniques will enable to establish this fact.

It would be interesting to know whether the equivalence between the two techniques holds for arbitrary colorless tasks, and not only for the 1-dimensional ones, and we conjecture that this is indeed the case. Note however that if this conjecture is true, then these two proof techniques cannot be complete, simply because it is known that set-agreement impossibility has no extension-based proof [2, 5].

The round-reduction Cl we define and study in this work is not the only one possible (nor are F_{IIS} used in [20]); defining a more powerful operator for proving lower bounds on general algorithms is a central open question left in [20], and we leave a similar question open here with regard to colorless algorithm. Nevertheless, we have shown that for 1-dimensional colorless tasks, Cl is in fact the most powerful operator possible – this operator is shown to be complete for such tasks in Corollary 12. The more general question of whether there exists if-and-only-if operators for wait-free computing, as was shown for synchronous failure-free computing in networks, is another central open question.

Another research direction is to try to design an analog of round-reduction for other computational models, such as t-resilient models (the speedup theorem of [20], as well as ours, assume the ability of each process to run solo). The ultimate goal of the line of study initiated in this paper is to better understand the relation between backward and forward induction in the context of distributed computing.

References

1 Marcos Kawazoe Aguilera and Sam Toueg. A simple bivalency proof that t-resilient consensus requires $t + 1$ rounds. *Inf. Process. Lett.*, 71(3-4):155–158, 1999. doi:10.1016/S0020-0190(99)00100-3.

2 Dan Alistarh, James Aspnes, Faith Ellen, Rati Gelashvili, and Leqi Zhu. Why extension-based proofs fail. In Moses Charikar and Edith Cohen, editors, *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC*, pages 986–996. ACM, 2019. doi:10.1145/3313276.3316407.

3 Dan Alistarh, Faith Ellen, and Joel Rybicki. Wait-free approximate agreement on graphs. In Tomasz Jurdzinski and Stefan Schmid, editors, *Structural Information and Communication Complexity - 28th International Colloquium, SIROCCO*, volume 12810 of *Lecture Notes in Computer Science*, pages 87–105. Springer, 2021. doi:10.1007/978-3-030-79527-6_6.

4 J. Aspnes and M. Herlihy. Wait-free data structures in the asynchronous PRAM model. In *2nd ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 340–349, 1990. doi:10.1145/97444.97701.

5 Hagit Attiya, Armando Castañeda, and Sergio Rajsbaum. Locally solvable tasks and the limitations of valency arguments. *J. Parallel Distributed Comput.*, 176:28–40, 2023. doi:10.1016/j.jpdc.2023.02.002.

6 Hagit Attiya and Faith Ellen. *Impossibility Results for Distributed Computing*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2014. doi:10.2200/S00551ED1V01Y201311DCT012.

7 Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, Hoboken, NJ, USA, 2004.

8 Alkida Balliu, Sebastian Brandt, Juho Hirvonen, Dennis Olivetti, Mikaël Rabie, and Jukka Suomela. Lower bounds for maximal matchings and maximal independent sets. In *60th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 481–497, 2019. doi:10.1109/FOCS.2019.00037.

9 Ofer Biran, Shlomo Moran, and Shmuel Zaks. A combinatorial characterization of the distributed 1-solvable tasks. *J. Algorithms*, 11(3):420–440, 1990. doi:10.1016/0196-6774(90)90020-F.

10 Elizabeth Borowsky and Eli Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *25 ACM Symposium on Theory of Computing (STOC)*, pages 91–100, 1993. doi:10.1145/167088.167119.

11 Elizabeth Borowsky and Eli Gafni. A simple algorithmically reasoned characterization of wait-free computations. In *16th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 189–198, 1997. doi:10.1145/259380.259439.

12 Elizabeth Borowsky, Eli Gafni, Nancy A. Lynch, and Sergio Rajsbaum. The BG distributed simulation algorithm. *Distributed Comput.*, 14(3):127–146, 2001. doi:10.1007/PL00008933.

13 Sebastian Brandt. An automatic speedup theorem for distributed problems. In *38th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 379–388, 2019. doi:10.1145/3293611.3331611.

14 Kayman Brusse and Faith Ellen. Reductions and extension-based proofs. In *PODC '21: ACM Symposium on Principles of Distributed Computing*, pages 497–507. ACM, 2021. doi:10.1145/3465084.3467906.

15 Armando Castañeda, Damien Imbs, Sergio Rajsbaum, and Michel Raynal. Generalized symmetry breaking tasks and nondeterminism in concurrent objects. *SIAM J. Comput.*, 45(2):379–414, 2016. doi:10.1137/130936828.

16 Armando Castañeda, Sergio Rajsbaum, and Matthieu Roy. Two convergence problems for robots on graphs. In *2016 Seventh Latin-American Symposium on Dependable Computing, LADC*, pages 81–90. IEEE Computer Society, 2016. doi:10.1109/LADC.2016.21.

17 Soma Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Inf. Comput.*, 105(1):132–158, 1993.

18 Faith E. Fich and Eric Ruppert. Hundreds of impossibility results for distributed computing. *Distributed Comput.*, 16(2-3):121–163, 2003. doi:10.1007/s00446-003-0091-y.

19 Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985. doi:10.1145/3149.214121.

20 Pierre Fraigniaud, Ami Paz, and Sergio Rajsbaum. A speedup theorem for asynchronous computation with applications to consensus and approximate agreement. In *PODC*, pages 460–470. ACM, 2022.

21 Pierre Fraigniaud, Sergio Rajsbaum, and Corentin Travers. Locality and checkability in wait-free computing. *Distributed Comput.*, 26(4):223–242, 2013. doi:10.1007/s00446-013-0188-x.

22 Eli Gafni and Sergio Rajsbaum. Distributed programming with tasks. In *14th International Conference on Principles of Distributed Systems (OPODIS)*, LNCS 6490, pages 205–218. Springer, 2010. doi:10.1007/978-3-642-17653-1_17.

23 Maurice Herlihy, Dmitry N. Kozlov, and Sergio Rajsbaum. *Distributed Computing Through Combinatorial Topology.* Morgan Kaufmann, 2013.

24 Maurice Herlihy and Sergio Rajsbaum. The decidability of distributed decision tasks. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing*, pages 589–598. ACM, 1997. doi:10.1145/258533.258652.

25 Maurice Herlihy and Sergio Rajsbaum. A classification of wait-free loop agreement tasks. *Theor. Comput. Sci.*, 291(1):55–77, 2003. doi:10.1016/S0304-3975(01)00396-6.

26 Maurice Herlihy and Sergio Rajsbaum. The topology of shared-memory adversaries. In *PODC*, pages 105–113. ACM, 2010.

27 Maurice Herlihy and Sergio Rajsbaum. Simulations and reductions for colorless tasks. In *ACM Symposium on Principles of Distributed Computing, PODC*, pages 253–260. ACM, 2012. doi:10.1145/2332432.2332483.

28 Maurice Herlihy, Sergio Rajsbaum, Michel Raynal, and Julien Stainer. From wait-free to arbitrary concurrent solo executions in colorless distributed computing. *Theor. Comput. Sci.*, 683:1–21, 2017.

29 Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *J. ACM*, 46(6):858–923, 1999. doi:10.1145/331524.331529.

30 Gunnar Hoest and Nir Shavit. Toward a topological characterization of asynchronous complexity. *SIAM J. Comput.*, 36(2):457–497, 2006. doi:10.1137/S0097539701397412.

31 Idit Keidar and Sergio Rajsbaum. A simple proof of the uniform consensus synchronous lower bound. *Inf. Process. Lett.*, 85(1):47–52, 2003. doi:10.1016/S0020-0190(02)00333-2.

32 Petr Kuznetsov, Thibault Rieutord, and Yuan He. An asynchronous computability theorem for fair adversaries. In *PODC*, pages 387–396. ACM, 2018.

33 Nathan Linial. Locality in distributed graph algorithms. *SIAM J. Comput.*, 21(1):193–201, 1992.

34 Nancy A. Lynch. A hundred impossibility proofs for distributed computing. In *8th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 1–28, 1989. doi:10.1145/72981.72982.

35 Nancy A. Lynch and Sergio Rajsbaum. On the borowsky-gafni simulation algorithm. In *ISTCS*, pages 4–15. IEEE Computer Society, 1996.

36 Moni Naor and Larry J. Stockmeyer. What can be computed locally? *SIAM J. Comput.*, 24(6):1259–1277, 1995. doi:10.1137/S0097539793254571.

37 David Peleg. *Distributed Computing: A Locality-Sensitive Approach.* SIAM, 2000.

38 Sergio Rajsbaum. Iterated shared memory models. In *LATIN*, volume 6034 of *Lecture Notes in Computer Science*, pages 407–416. Springer, 2010.

39 Joseph Rotman. Covering complexes with applications to algebra. *Rocky Mountain Journal of Mathematics*, 3(4):641–674, 1973. doi:10.1216/RMJ-1973-3-4-641.

40 Michael E. Saks and Fotios Zaharoglou. Wait-free k-set agreement is impossible: the topology of public knowledge. In *25th ACM Symposium on Theory of Computing (STOC)*, pages 101–110, 1993. doi:10.1145/167088.167122.

41 Hans van Ditmarsch, Éric Goubault, Marijana Lazic, Jérémy Ledent, and Sergio Rajsbaum. A dynamic epistemic logic analysis of equality negation and other epistemic covering tasks. *J. Log. Algebraic Methods Program.*, 121:100662, 2021. `doi:10.1016/j.jlamp.2021.100662`.

A Omitted Proofs

Proof of Theorem 10. To establish the theorem, we first prove two auxiliary claims, with the same notations as in the statement of the theorem.

▷ Claim 20. Every two vertices $u \neq w$ of the same connected component of $\Delta(\sigma)$ satisfy $\{u,w\} \in \Delta^{(\ell-1)}(\sigma)$.

Proof of claim. Let u, v, w be three vertices of the same connected component $\mathcal{K}$, such that $\{u,v\} \in \mathcal{K}, \{v,w\} \in \mathcal{K}$, but $\{u,w\} \notin \mathcal{K}$, and let us show that $\{u,w\} \in \Delta^{(1)}(\sigma)$. (If there are no such three vertices then we are done.) For this, it is sufficient to define a simplicial map

$$f : \mathsf{Bary}(\{u,w\}) \to \Delta(\sigma)$$

which agrees with $\Delta_{\{u,w\},\sigma}$. We set

$$f(\{u\}) = u,\ f(\{w\}) = w, \text{ and } f(\{u,w\}) = v.$$

In this way, any edge of $\mathsf{Bary}(\{u,w\})$ is mapped to either $\{u,v\}$ or $\{v,w\}$, which both belong to $\Delta(\sigma)$. It follows that f is simplicial, and agrees with $\Delta_{\{u,w\},\sigma}$. Therefore, for any two vertices u, w of $\mathcal{K}$ at distance at most 2 in the graph $\mathsf{Skel}_1(\mathcal{K})$, $\{u,w\} \in \Delta^{(1)}(\sigma)$. By the same argument, any two vertices u, w of $\mathcal{K}$ at distance at most 4 in the graph $\mathsf{Skel}_1(\mathcal{K})$ satisfy $\{u,w\} \in \Delta^{(2)}(\sigma)$, and more generally, for any two vertices u, w of $\mathcal{K}$ at distance at most 2^r in the graph $\mathsf{Skel}_1(\mathcal{K})$, $\{u,w\} \in \Delta^{(r)}(\sigma)$. As a consequence, for every two vertices u, w of $\mathcal{K}$, $\{u,w\} \in \Delta^{(\ell-1)}(\sigma)$. ◁

We next show that a similar claim holds for any set of vertices in a connected component of $\Delta(\sigma)$, and not only for pairs.

▷ Claim 21. Every set $\tau \subseteq V(\Delta(\sigma))$ of vertices of the same connected component of $\Delta(\sigma)$ with $2 < |\tau| \leq |\sigma|$ satisfies $\tau \in \Delta^{(\ell)}(\sigma)$.

Proof of claim. It is sufficient to show that the local task $\Pi^{(\ell-1)}_{\tau,\sigma} = (\tau, \Delta^{(\ell-1)}(\sigma), \Delta^{(\ell-1)}_{\tau,\sigma})$ is solvable in one round, which we do by defining a simplicial map

$$f : \mathsf{Bary}(\tau) \to \Delta^{(\ell-1)}(\sigma)$$

that agrees with $\Delta^{(\ell-1)}_{\tau,\sigma}$, as follows. Let $\tau = \{v_1, \dots, v_d\}$ where the vertices of τ are indexed in an arbitrary order, where $d = |\tau|$. For every singleton set $\{v_i\} \subseteq \tau$ we set

$$f(\{v_i\}) = v_i$$

while for every set $S \subseteq \tau$ of cardinality $|S| > 1$, we set

$$f(S) = v_1.$$

Let τ' be a face of τ. The views encoded by different vertices in a simplex $\rho \in \mathsf{Bary}(\tau')$ of a barycentric subdivision are totally ordered by inclusion, so ρ may contain at most one vertex that corresponds to singleton view (i.e., a view composed of a single vertex of τ'). As a consequence, there are only three possible cases:

- $f(\rho) = v_i$ whenever $\rho = \{v_i\}$, or
- $f(\rho) = v_1$ whenever ρ contains no singleton sets but possibly $\{v_1\}$, or
- $f(\rho) = \{v_1, v_i\}$ for some $i \in \{2, \dots, d\}$ whenever ρ contains the singleton $\{v_i\}$ plus other non-singleton vertices.

In all three cases, the image of ρ is a simplex of $\Delta^{(\ell-1)}_{\tau,\sigma}(\tau')$, by Claim 20. Therefore f is simplicial, and agrees with $\Delta^{(\ell-1)}_{\tau,\sigma}$, which implies that $\tau \in \Delta^{(\ell)}(\sigma)$. $\lhd$

We are now ready to prove Theorem 10. For the first part of the theorem, let $\mathcal{K}$ be a connected component of $\Delta(\sigma)$, and our goal is to show that every $\tau \subseteq V(\mathcal{K})$ with $1 \leq |\tau| \leq \dim(\sigma) + 1$ satisfies $\tau \in \Delta^{(\ell)}(\sigma)$. The claim holds for $|\tau| = 1$ (i.e., for vertices) as every vertex of $\mathcal{K}$ is by definition a vertex of $\Delta(\sigma)$. It holds for $|\tau| = 2$ (i.e., for edges) by Claim 20, and for $|\tau| > 2$ by Claim 21.

For establishing the second item, first note that one direction of the if and only if statement is trivial, as the closure operator can only add simplices, so connected components of $\mathsf{Skel}_1(\Delta^{(k-1)}(\sigma))$ can only merge when moving to $\mathsf{Skel}_1(\Delta^{(k)}(\sigma))$ and not break. For the other direction, we proceed by induction on $k \geq 0$. The statement is trivial for $k = 0$.

Let us assume that the statement holds for k and show that it holds for $k + 1$. Let u and v be two vertices of $\Delta(\sigma)$ in two different connected components of $\Delta(\sigma)$, and by assumption in two different connected components of $\Delta^{(k)}(\sigma)$. Let us show that $\{u, v\} \notin \Delta^{(k+1)}(\sigma)$. For the purpose of contradiction, let us consider a map

$$f : \mathsf{Bary}(\{u, v\}) \to \Delta^{(k)}(\sigma)$$

that agrees with $\Delta^{(k)}_{\{u,v\},\sigma}$. We must have $f(\{u\}) = u$, $f(\{v\}) = v$, and $f(\{u, v\}) = w$ for some vertex $w \in V(\Delta^{(k)}(\sigma))$. However, u and v are in two different connected components of $\Delta^{(k)}(\sigma)$, which implies that $\{u, w\}$ or $\{v, w\}$ is not an edge of $\Delta^{(k)}(\sigma)$. As a consequence, f is not simplicial, and thus $\{u, v\} \notin \Delta^{(k+1)}(\sigma)$. In other words, no edges can be added between different connected components of $\Delta(\sigma)$ during successive closure operations. ◀

Proof of Theorem 11. Let $\Pi = (\mathcal{I}, \mathcal{O}, \Delta)$ such that $\mathsf{Cl}(\Pi)$ is solvable in $t - 1$ rounds by n processes. Let

$$\delta' : \mathsf{Bary}^{(t-1)}(\mathsf{Skel}_n(\mathcal{I})) \to \mathcal{O}$$

be a simplicial map agreeing with Δ. Roughly, an algorithm for solving Π consists of two phases: first solving $\mathsf{Cl}(\Pi)$ using δ', and, second, given the output of δ', reconciliating these outputs (valid for $\mathsf{Cl}(\Pi)$, but not necessarily for Π) using the algorithm solving the local task for these outputs (see Fig. 3). More formally, les us consider a process p with input x, i.e., $x \in \mathcal{I}$ is a vertex. After $t - 1$ rounds, this process gets a view w which is a vertex of $\mathsf{Bary}^{(t-1)}(\mathcal{I})$. Then, p proceeds with one more round of communication, and gets a view in $\mathsf{Bary}^{(t-1)}(\mathcal{I})$, which is either of the form $\{w\}$ or of the form $\{w, w'\}$ where $\{w, w'\}$ is an edge of $\mathsf{Bary}^{(t-1)}(\mathcal{I})$. Note that the property of the Barycentric subdivision guarantee that in the latter case w' must contain an input $x' \neq x$ of another process, where $e = \{x, x'\}$ was the actual input. Let $y = \delta'(w)$ and $y' = \delta'(w')$. Moreover, let $e' = \{y, y'\}$. Note that e' is an edge of $\Delta'(e)$ as δ' solves $\mathsf{Cl}(\Pi)$, but it is not necessarily an edge of $\Delta(e)$. The algorithm solving Π is as follows:

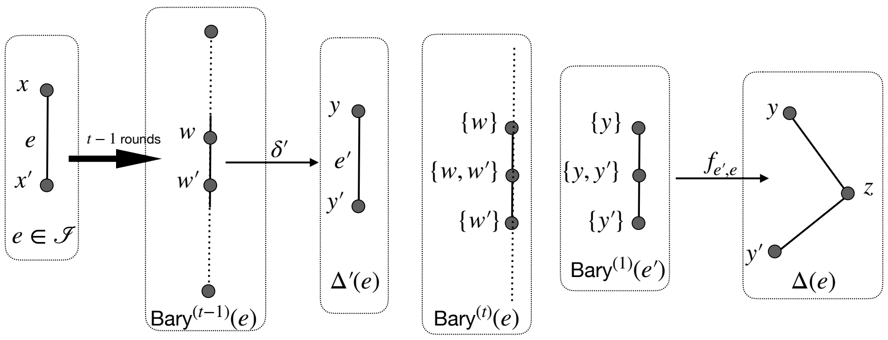


Figure 3 Proof of Theorem 11.

- If the view of p after t rounds is $\{w\}$, then p outputs y;
- If the view of p after t rounds is $\{w, w'\}$, then p outputs $z = f_{e',e}(\{y, y'\})$ where

$$f_{e',e} : \mathsf{Bary}^{(1)}(e') \to \Delta(e)$$

is a simplicial map[2] solving the local task $\Pi_{e',e}$.

This algorithm is well defined, as if p has view $\{w, w'\}$, then it knows x and x', and it can compute y and y' from the views w and w'. To show correctness, let $\sigma \in \mathcal{I}$, and let us show that our algorithm produces a simplex $\tau \in \Delta(\sigma)$. If σ is a vertex x, then all processes output $y \in \Delta'(x)$, which is a vertex of $\Delta(x)$, as desired. If σ is an edge $e = \{x, x'\}$, then let $\{w, w'\} \in \mathsf{Bary}^{(t-1)}(e)$ be the edge of the barycentric subdivision corresponding the the current configuration after $t-1$ rounds. Assume, w.l.o.g., that, during the t-th round, some processes (maybe none) get view $\{w\}$ while some other processes (at least one) get view $\{w, w'\}$ – the latter view is a vertex of $\mathsf{Bary}^{(t)}(e)$, whereas it was an edge of $\mathsf{Bary}^{(t-1)}(e)$. Note that starting from $\{w, w'\} \in \mathsf{Bary}^{(t-1)}(e)$, it is not possible that some processes reads a view $\{w\}$ in $\mathsf{Bary}^{(t)}(e)$ while some other processes reads $\{w'\}$ in $\mathsf{Bary}^{(t)}(e)$. It follows that a group of processes may output $y = \delta'(x)$ while another group of processes may output $z = f_{e',e}(\{y, y'\})$. The crucial property is that $f_{e',e}$ fixes vertices, that is, $f_{e',e}(\{y\}) = y$. Since $f_{e',e}$ is simplicial and agrees with $\Delta_{e',e}$, we get that

$$\{y, z\} = \{f_{e',e}(\{y\}), f_{e',e}(\{y, y'\})\} \in \Delta_{e',e}(e') = \Delta(e),$$

as desired. This completes the proof of the theorem. ◀

Proof of Theorem 13. Fix a colorless task Π that has a round-reduction impossibility proof. By Lemma 8, Π^* is not 0-round solvable. For each simplex $\sigma \in \mathcal{O}$, all the connected components of $\Delta^*(\sigma)$ are complete up to dimension $\dim(\sigma)$ by Claim 21: if some simplices of dimension at most $\dim(\sigma)$ are missing in $\Delta^*(\sigma)$ then at least one of them would have been added to it when applying the closure operator, contradicting the fact that Π^* is a fixed-point.

To construct an FLP-style proof, we consider an algorithm A that claims to solve Π, and define $\delta : V(\mathcal{I}) \to V(\mathcal{O})$ to map each input value $x \in \mathcal{I}$ to the output value $\delta(x)$ produced by A in the execution where only the value x appears; $\delta(x)$ is unique since the execution

[2] There might be more than one simplicial map from $\mathsf{Bary}^{(1)}(e')$ to $\Delta(e)$, in which case one selects one of them arbitrarily for defining the algorithm solving Π.

is unique and the algorithm is deterministic. The fact that $\mathsf{Cl}^*(\Pi)$ is not 0-round solvable means that there is a simplex $\sigma \in \mathcal{I}$ such that $\delta(\sigma) = \{\delta(x) \mid x \in \sigma\} \notin \Delta^*(\sigma)$. As the connected components are complete up to dimension $\dim(\sigma)$, the simplex σ is mapped to (at least) two different connected components, i.e. there are two input values $x, x' \in \sigma$ and two connected components C, C' of $\Delta(\sigma)$ such that $\delta(x) \in C$ and $\delta(x') \in C'$.

Let $\sigma_0 = \{x, x'\}$. As $\sigma_0 \subseteq \sigma$, the fact that Δ^* is a carrier map (Lemma 5) implies that the connected components of $\Delta(\sigma_0)$ are a refinement of the connected components of $\Delta(\sigma)$. Hence, there is a connected component $C_0 \subseteq C$ of $\Delta(\sigma_0)$ such that $\mathsf{val}(x) \cap C_0 \neq \emptyset$, and similarly a different connected component $C'_0 \subseteq C'$ of $\Delta(\sigma_0)$ such that $\mathsf{val}(x') \cap C'_0 \neq \emptyset$. Note that Theorem 10 asserts that the connected components of $\Delta(\sigma_0)$ and of $\Delta^*(\sigma_0)$ are the same.

Let us say that a configuration reachable from σ_0 is *bivalent* (w.r.t. σ_0) if a valency query on it returns output values in at least two different connected components of $\Delta(\sigma_0)$. Note that σ_0 is bivalent by construction, and that if the algorithm is in a bivalent configuration it cannot decide without taking further steps.

We construct an infinite sequence $\sigma_0, \sigma_1, \ldots$ of bivalent configurations. Each σ_t will consist only of two views

$$\sigma_t = \{w_t, w'_t\}.$$

Assume $\sigma_0, \ldots, \sigma_t$ are chosen and $\mathsf{val}(\sigma)$ is known for each $\sigma \in \mathsf{Bary}_1(\sigma_t)$. Recall that $\mathsf{Bary}_1(\sigma_t)$ is composed of the vertices $\{w_t\}, \{w_t, w'_t\}, \{w'_t\}$ and the edges $e = (\{w_t\}, \{w_t, w'_t\})$ and $e' = (\{w'_t\}, \{w_t, w'_t\})$. Let C_t be a connected component of $\Delta(\sigma_0)$ such that $\mathsf{val}(\{w_t, w'_t\}) \cap C_t \neq \emptyset$.

The fact that σ_t is bivalent implies that there is a connected component C'_t of $\Delta(\sigma_0)$, $C'_t \neq C_t$, such that $\mathsf{val}(\sigma_t) \cap C'_t \neq \emptyset$. As the valencies of the vertices are contained in the valencies of the edges, we have $\mathsf{val}(e) \cup \mathsf{val}(e') = \mathsf{val}(\sigma_t)$. Hence, at least one edge $f \in \{e, e'\}$ has $\mathsf{val}(f) \cap C'_t \neq \emptyset$. Since $\{w_t, w'_t\} \in f$, it also has $\mathsf{val}(f) \cap C_t \neq \emptyset$. The edge f is thus bivalent, and we set it as σ_{t+1}. This process can continue for every $t \geq 0$, and the proof is complete. ◀

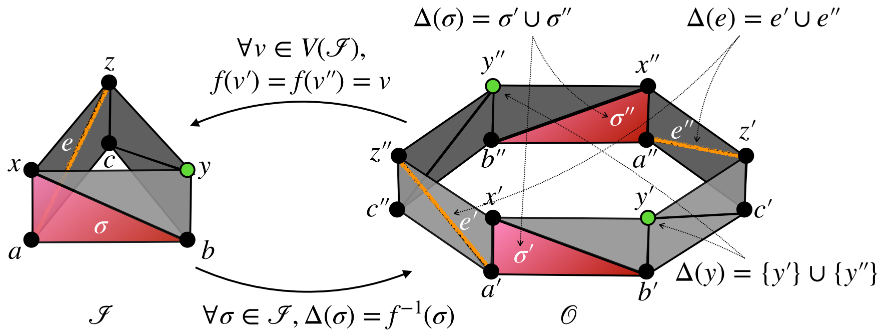
Proof of Proposition 17. The diameter D of $\mathcal{O}$ is $N = 1/\epsilon$. Let us first show that if $0 \leq k < \lceil \log_2 1/\epsilon \rceil$, we have $\{0,1\} \notin \Delta^{(k)}(\{0,1\})$. The proof is based on the following fact: for any two distinct vertices u and v of $\Delta^{(k)}(\{0,1\})$ at distance at least 3 in $\mathsf{Skel}_1(\Delta^{(k)}(\{0,1\}))$, we have $\{u,v\} \notin \Delta^{(k+1)}(\{0,1\})$. To establish this fact, let us consider any map

$$f : \mathsf{Bary}^{(1)}(\{u,v\}) \to \Delta^{(k)}(\{0,1\})$$

agreeing with $\Delta^{(k)}_{\{u,v\},\{0,1\}}$. Since f agrees with $\Delta^{(k)}_{\{u,v\},\{0,1\}}$, we must have $f(u) = u$ and $f(v) = v$. Therefore, if $w = f(\{u,v\})$ then either $\{u,w\}$ or $\{w,v\}$ is not an edge of $\Delta^{(k)}(\{0,1\})$ because u and v are at distance greater than 2. Therefore f is not simplicial, which shows that $\{u,v\} \notin \Delta^{(k+1)}(\{0,1\})$, as claimed. It follows that, for $k < \lceil \log_2 1/\epsilon \rceil$, we have $\{0,1\} \notin \Delta^{(k)}(\{0,1\})$.

This latter fact implies that the k-th closure of ϵ-agreement is not solvable in zero rounds: an algorithm f solving this k-th closure must satisfy $f(0) = 0$ and $f(1) = 1$. As a consequence, if some processes start with 0, and some other processes start with input 1, all these processes jointly output the set $\{0,1\}$, which is not a valid output as $\{0,1\} \notin \Delta^{(k)}(\{0,1\})$. ◀

Proof of Theorem 19. Consider a non-trivial covering complex $(\mathcal{O}, f)$ of a complex $\mathcal{I}$, and the corresponding covering task $\Pi = (\mathcal{I}, \mathcal{O}, \Delta)$. For the case of the Hexagon task we have seen that $\mathsf{Cl}(\mathrm{HX}) = \mathrm{HX}$, and here we start the same why.



$\Delta(\sigma) = \sigma' \cup \sigma''$
$\Delta(e) = e' \cup e''$
$\forall v \in V(\mathcal{I})$,
$f(v') = f(v'') = v$
z
x
y
c
e
σ
a
b
y''
x''
z''
b''
a''
σ''
e''
z'
c''
x'
y'
e'
σ'
c'
a'
b'
$\Delta(y) = \{y'\} \cup \{y''\}$
$\mathcal{I}$
$\forall \sigma \in \mathcal{I}, \Delta(\sigma) = f^{-1}(\sigma)$
$\mathcal{O}$

Figure 4 A 2-dimensional covering task extending the Hexagone task to a higher dimension. Here, $f(\sigma') = f(\sigma'') = \sigma$, and accordingly the image of σ under Δ is the union of the two complexes with unique facets σ' and σ''.

Consider an input simplex $\sigma \in \mathcal{I}$, and note that each of its sheets is a simplex, and that its sheets do not intersect. Hence, all the connected components of $\Delta(\sigma)$ are complete (each is of dimension $\dim(\sigma)$), and Theorem 10 implies that $\Delta^*(\sigma) = \Delta(\sigma)$, so $\mathsf{Cl}(\Pi) = \Pi$. Hence, if Π is solvable wait-free by $n \geq 2$ processes in the IIS model, then by Lemma 8 it is wait-free solvable in zero rounds.

Consider a possible zero-round algorithm for Π, and its decision map δ. Recall that δ must be simplicial, i.e. maps simplices to simplices, and hence also paths to paths.

Fix $x \in V(\mathcal{I})$, its image $y = \delta(x) \in V(\mathcal{O})$ under δ, and note that $f(y) = x$. Since $(\mathcal{O}, f)$ is non-trivial, there is another vertex $y' \in V(\mathcal{O})$, $y' \neq y$, such that $f(y') = x$. As $\mathcal{O}$ is connected, it contains a path $(y_0 = y, y_1, \ldots, y_k = y')$ connecting y and y', and $\mathcal{I}$ contains its image $C = (x_0 = f(y_0), \ldots, x_k = f(y_k))$ under f. Note that $x = x_0 = x_k$, so C is in fact a cycle in $\mathcal{I}$. Apply δ to C, and the fact that δ is simplicial gives a cycle $\delta(C)$ in $\mathcal{O}$.

As $\delta(x_0) = y_0$ but $\delta(x_k) \neq y_k$, there must exist a minimal index $0 \leq i < k$ such that $\delta(x_i) = y_i$ and $\delta(x_{i+1}) \neq y_{i+1}$. By the construction of the path, $f(y_i, y_{i+1}) = (x_i, x_{i+1})$, and by the assumption that δ solves the task and hence comply with Δ we have $f(y_i, \delta(x_{i+1})) = (x_i, x_{i+1})$. Hence $f^{-1}(x_i, x_{i+1})$ contains both (y_i, y_{i+1}) and $(y_i, \delta(x_{i+1}))$, i.e. (x_i, x_{i+1}) has two intersecting sheets, in contradiction to $(\mathcal{O}, f)$ being a covering complex. ◀

Topological Characterization of Task Solvability in General Models of Computation

Hagit Attiya
Department of Computer Science, Technion, Haifa, Israel

Armando Castañeda
Instituto de Matemáticas, Universidad Nacional Autónoma de México, Mexico

Thomas Nowak
Laboratoire Méthodes Formelles, Université Paris-Saclay, CNRS, ENS Paris-Saclay, France
Institut Universitaire de France, Paris, France

Abstract

The famous *asynchronous computability theorem* (ACT) relates the existence of an asynchronous wait-free shared memory protocol for solving a task with the existence of a simplicial map from a subdivision of the simplicial complex representing the inputs to the simplicial complex representing the allowable outputs. The original theorem relies on a correspondence between protocols and simplicial maps in round-structured models of computation that induce a compact topology. This correspondence, however, is far from obvious for computation models that induce a non-compact topology, and indeed previous attempts to extend the ACT have failed.

This paper shows that in *every* non-compact model, protocols solving tasks correspond to simplicial maps that need to be *continuous*. It first proves a *generalized* ACT for sub-IIS models, some of which are non-compact, and applies it to the *set agreement* task. Then it proves that in general models too, protocols are simplicial maps that need to be continuous, hence showing that the topological approach is *universal*. Finally, it shows that the approach used in ACT that equates protocols and simplicial complexes actually works for *every* compact model.

Our study combines, for the first time, combinatorial and point-set topological aspects of the executions admitted by the computation model.

2012 ACM Subject Classification Theory of computation → Distributed computing models

Keywords and phrases task solvability, combinatorial topology, point-set topology

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.5

Related Version *Full Version*: `https://arxiv.org/abs/2301.13837`

Funding *Hagit Attiya*: partially supported by the Israel Science Foundation (grants 380/18 and 22/1425).
Armando Castañeda: partially supported by the DGAPA PAPIIT project IN108723.
Thomas Nowak: partially supported by the ANR grant ANR-21-CE48-0003.

1 Introduction

The celebrated topological approach in distributed computing relates task solvability to the topology of inputs and outputs of the task and the topology of the protocols allowed in a particular model of computation. This approach rests on three pillars. First, *configurations*, whether of inputs, outputs or protocol states, can be modeled as *simplexes*, which are finite sets. Second, the inherent *indistinguishability* of configurations is crisply captured by intersections between simplexes. Third, a *carrier map* captures the notion of the set of configurations that are reachable from a given configuration.

More concretely, in this approach, *tasks* are triples $T = (\mathcal{I}, \mathcal{O}, \Delta)$, where $\mathcal{I}$ and $\mathcal{O}$ are *simplicial complexes* modeling the inputs and outputs of the task, and Δ is a *carrier map* specifying the possible valid outputs, $\Delta(\sigma)$, for each input simplex $\sigma \in \mathcal{I}$. Similarly, *protocols* are triples $P = (\mathcal{I}, \mathcal{P}, \Xi)$, where $\mathcal{P}$ is the complex modeling the final configurations of the protocol, and Ξ is a carrier map specifying the reachable final configurations, $\Xi(\sigma)$, from σ.

With this perspective in mind, it is natural to conclude that a protocol maps final states (*i.e.*, states in final configurations) to outputs, and for the protocol to be correct, the mapping must be *simplicial*; that is, all outputs of final states in the same simplex $\tau \in \Xi(\sigma)$ (*i.e.*, in the same final configuration) must be in the same output simplex of $\Delta(\sigma)$. Thus, a protocol induces a simplicial map from $\mathcal{P}$ to $\mathcal{O}$. Moreover, since decisions of processes are only based on local information, it is natural to conclude the converse, *i.e.*, *any* simplicial map implies a protocol. (In general, a protocol specifies also the communication during an execution. However, solvability can consider only existence of a decision function, by assuming that the protocol is *full-information.*) This leads to the following purely topological solvability characterization: a protocol $P = (\mathcal{I}, \mathcal{P}, \Xi)$ solves a task $T = (\mathcal{I}, \mathcal{O}, \Delta)$ *if and only if* there is a simplicial map $\delta : \mathcal{P} \to \mathcal{O}$ such that for every $\sigma \in \mathcal{I}$, we have $(\delta \circ \Xi)(\sigma) \subseteq \Delta(\sigma)$.

The discussion so far did not depend on a particular model of computation. Indeed, this approach seems universal and gives the impression that protocols and simplicial maps are the same, and that for all models, the solvability question can be reduced to the existence of a simplicial map. In fact, the correspondence between protocols and simplical maps seems so self-evident that frequently the characterization above seems to require no proof, and is introduced as a definition (*e.g.*, [13, Section 4.2.2; Definition 8.4.2]).

This approach works well for cases where the model of computation has a particular round structure[1] and it induces a *compact* topology so that the correspondence between protocols and simplicial maps holds. Roughly speaking, a compact topological space has no "punctures" or "missing endpoints", namely, it does not exclude any limit point. If a model of computation is specified as a set of infinite executions, then a compact model will contain all its "limit executions". For example, the *Iterated Immediate Snapshot* (IIS) model ensures that computation proceeds in sequence of (implicit) rounds; in each round, any of a *finite* set of possible schedules can happen. Thus, the model contains every infinite execution with this round structure. Models like IIS are sometimes called *oblivious* [7], and are known to induce *finite complexes with compact topology, where protocols and simplicial maps are the same.* Round-structured compact models have been extensively studied in the literature, and different techniques have been developed for them (*e.g.*, [4, 6, 10, 14]).

However, this approach is not true in all models, specifically, in non-compact ones. In a non-compact model, typically some "good" schedules only *eventually* happen, which then implies that the model is not limit-closed. Examples of a non-compact models are t-resilient *asynchronous* models where any process is guaranteed to eventually obtain information from at least $t-1$ other processes infinitely often, but the process can take an unbounded number of steps before that happens. This means, for example, that the model contains every infinite execution where a process runs solo for a *finite* number of steps and then obtains information from $t-1$ other processes, but it does not contain the infinite solo execution of the process, *i.e.*, the limit execution.

Challenging non-compact models have been mostly treated in the literature indirectly, through "compactification". Sometimes, compactification consists of considering only protocols with a concrete round structure, as is done in some chapters of [13]. In other cases, a

[1] Rounds may be explicit, like in the synchronous message-passing t-resilient model, or implicit, for example, modeled by *layers*, as in the Iterated Immediate Snapshot model.

computationally-equivalent round-structured compact model is analyzed instead (*e.g.*, [17,22]). This requires first to prove that the models are equivalent, through simulations in both directions, and then characterize solvability in the compact model. For impossibility results, it is also sometimes possible to identify a compact sub-model in which a problem of interest is still unsolvable [18].

Round-structured compact models have also served to analyze other compact models. In the famous *asynchronous computability theorem* (ACT) [15], fully characterizing solvability of the non-round-structured compact read/write wait-free shared memory model, a crucial step is showing equivalence with the IIS model. This restricts the solvability characterization to subdivisions, which are well-behaved topological spaces, making the ACT highly useful.

Some sub-IIS models, with subsets of IIS like t-resilient computations, are non-compact. For this reason, an attempt [11] to generalize the ACT to arbitrary sub-IIS models and tasks had to directly address non-compact models. The idea of this so-called *generalized* ACT is to somehow reuse the nice structure of IIS, modeling any sub-IIS model as a possibly infinite subdivision.

This raises two important questions that have not been explicitly investigated so far, which are addressed in this paper. (1) Can protocols in all models of distributed computation be captured as simplicial maps? (2) Can the topological approach be applied to all models of computation? The answers to these questions are not self-evident since there may be non-compact models or non-round-structured compact models, which cannot be compactified.

We note that there is already a recent negative answer to the first question: Godard and Perdereau [12] showed a non-compact model where consensus is unsolvable, but nevertheless, it has a simplicial map as described above. Roughly, it considers a sub-IIS model where a single infinite execution of IIS is removed. The resulting model is not compact. It turns out that the complex of a protocol is an *infinite* subdivision that is *disconnected*, hence, there is a simplicial map from it to the consensus output complex (which is disconnected too). But the map does not imply a consensus protocol, since intuitively, the decisions must be consistent as they approach the discontinuity of the removed execution. More specifically, the simplicial map does not imply a protocol because it is *not continuous.* Section 3 details the example based on their ideas. While continuity of simplicial maps is guaranteed in compact models, this is not the case in non-compact ones. This example demonstrates that the generalization in [11] is flawed as it misses the continuity property of simplicial maps. Godard and Perdereau also correct this problem *for the special case of two processes and the consensus task.*

Continuity of simplicial maps may seem trivial, but it was overlooked for long time, before [12]. Here, we further expose its importance.

We first study task solvability in the well-structured simplicial complexes induced by sub-IIS models. Our first contribution (Theorem 4.1) is to present a correct generalized ACT for any number of processes and arbitrary tasks. Our approach is motivated by the critical role of continuity. Our second contribution (Theorem 4.2) is to use our generalized ACT theorem in order to provide an impossibility condition for set agreement in sub-IIS models, where the continuity requirement of simplicial maps allows a natural generalization of the known impossibility conditions for round-structured compact models.

While this settles the questions for sub-IIS models, the questions for general models remain open. Our third contribution (Theorem 5.4) shows that the topological approach is applicable in *all* models of computation, if one requires simplicial maps to be continuous. Unlike the case of round-structured compact and sub-IIS models, proving the applicability of the topology approach to general non-compact models is not straightforward. It requires to combine *point-set topological* techniques [2] with *combinatorial topology* techniques [13].

We use this result in our fourth contribution: a proof that the approach described at the beginning of the introduction, equating protocols and simplicial maps that are not required to be continuous, is universal for compact models (Theorem 6.3). Namely, in every compact model, possibly non-round-structured, it is indeed the case that there is correspondence between protocols and simplicial maps, hence the approach works in all these cases. The proof of this result is far from trivial, and it uses *projective limits* from *category theory* [19].

As far as we know, non-compact models have been directly studied only in [8, 9, 11, 12, 20]. A full combinatorial solvability characterization for two-process consensus under synchronous general message-loss failures appears in [9]. For the case of two processes, these models are all sub-IIS, hence this work is the first that directly studies non-compact models. Then, [11] attempted to generalize ACT to general sub-IIS models and tasks, for any number of processes. The solvability of two-process consensus is studied again in [12], now from a combinatorial topology perspective, where it is shown that the attempt in [11] is flawed. That paper also provides an alternative full topological solvability characterization for two-process consensus. Recently, sub-IIS models were studied through *geometrization* [8], *i.e.*, using a mapping from IIS executions to points in the Euclidean space, which in turn induces a topology. The geometrization is used to derive a full solvability characterization for *set agreement* in sub-IIS models, and it generalizes the two-process consensus solvability characterization of [12]. A solvability characterization for consensus (only) in general models, for any number of processes, is presented in [20]. It is derived using point-set topology techniques from [2], without combining them with combinatorial topology. Recent formalizations [1, 3] for proofs based on valency arguments show that for some tasks, *e.g.*, set agreement and renaming, impossibility cannot be shown by inductively constructing infinite executions. This means that arguments regarding the final protocol states are necessary in order to prove impossibility. Our results indicate that such proofs can be carried within combinatorial topology, in general models of computation.

In summary, our contributions are:

1. A generalized ACT for arbitrary sub-IIS models (Theorem 4.1).
2. An application of the generalized ACT to set agreement (Theorem 4.2).
3. A proof that if simplicial maps from $\mathcal{P}$ to $\mathcal{O}$ are required to be continuous, the topological approach works for every model of computation (Theorem 5.4).
4. A proof that the usual topological approach where simplicial maps are not required to be continuous works for every compact model (Theorem 6.3).

2 Preliminaries

This section presents the elements of combinatorial topology and point set topology used in further sections, and defines tasks, system models and task solvability.

We start by fixing some basic notation. We denote by Π the set of processes and let $n = |\Pi|$. For any function $f : X \to Y$ and subsets $A \subseteq X$ and $B \subseteq Y$, we denote by $f[A]$ the image of the set A under f and by $f^{-1}[B]$ the inverse image of the set B under f.

2.1 Elements of Combinatorial Topology and Decision Tasks

To be the most general possible, we use the language of colored tasks [13, Definition 8.2.1], to study one-shot distributed decision tasks like consensus or set agreement. We use the standard concepts in [13] with the only difference that simplicial complexes might be infinite, *i.e.*, a possibly infinite sets of finite sets. Definitions of concepts like simplicial and carrier maps, geometric realization, standard chromatic subdivision and more appear in the Appendix. Here we just recall the definition of tasks.

A *decision task* is a triple $T = (\mathcal{I}, \mathcal{O}, \Delta)$ such that:

- $\mathcal{I}$, the *input complex*, is a *finite* pure chromatic simplical complex of dimension $n-1$, whose vertices are additionally labeled by a set of inputs V^{in}. Each simplex of $\mathcal{I}$ specifies private inputs for the processes that appear in the simplex.
- $\mathcal{O}$, the *output complex*, is a *finite* pure chromatic simplical complex of dimension $n-1$, whose vertices are additionally labeled by a set of inputs V^{out}. As above, each simplex of $\mathcal{O}$ specifies private outputs for the processes in the simplex.
- Δ is a chromatic carrier map from $\mathcal{I}$ to $\mathcal{O}$, $\Delta(\sigma)$, that *specifies* the valid outputs for every input simplex σ in $\mathcal{I}$. Namely, when the inputs are the ones specified in σ, the outputs in any simplex of $\Delta(\sigma)$ are allowed.

Whenever the complex is understood from the context, we will denote by $v(p, x)$ the unique vertex of the complex with color $p \in \Pi$ and label x.

2.2 Elements of Point-Set Topology

In addition to combinatorial topology, we employ point-set topology [5], *i.e.*, the general mathematical theory of closeness, convergence, and continuity. The topologies that we define here are described by *metrics*, which are distance functions $d : X \times X \to [0, \infty)$ that satisfy:

1. Positive definiteness: $d(x, y) = 0$ if and only if $x = y$
2. Symmetry: $d(x, y) = d(y, x)$
3. Triangle inequality: $d(x, z) \le d(x, y) + d(y, z)$

A set equipped with a metric is called a *metric space*. The most basic metric is the *discrete metric*, which is defined by:

$$d(x, y) = \begin{cases} 0 & \text{if } x = y \\ 1 & \text{if } x \neq y \end{cases}$$

That is, the discrete metric can only give the information whether two elements are equal, but implies no finer-grained notion of closeness.

A central notion in point-set topology are *open sets*, which are subsets $O \subseteq X$ such that

$$\forall x \in O \quad \exists \varepsilon > 0 \colon \quad B_\varepsilon(x) \subseteq O$$

where $B_\varepsilon(x) = \{y \in X \mid d(x, y) < \varepsilon\}$ is the open ball with radius ε around x. With respect to the discrete metric, every subset $O \subseteq X$ is open. This follows from the fact that the open ball with radius $1/2$ around x is equal to $B_{1/2}(x) = \{x\}$, *i.e.*, only contains x itself.

The general definition of a *topological space* is a nonempty set X together with a *topology*, *i.e.*, a set $\mathcal{O} \subseteq 2^X$ of subsets of X that is closed under arbitrary unions and finite intersections. The elements of $\mathcal{O}$ are called the open sets of the space. With the above definition, every metric induces a topology.

A particular class of metrics that we use in this paper is that of *ultrametrics*. They satisfy the stronger ultrametric triangle inequality: $d(x, z) \le \max\{d(x, y), d(y, z)\}$ for all $x, y, z \in X$. The discrete metric is an example of an ultrametric. In an ultrametric space, two open balls are either disjoint or one is a subset of the other, as is shown by the following folklore lemma:

▶ **Lemma 2.1.** *Let X be an ultrametric space. For all $x, y \in X$ and all $\delta, \varepsilon > 0$, one of the following is true: (1) $B_\delta(x) \cap B_\varepsilon(y) = \emptyset$, (2) $B_\delta(x) \subseteq B_\varepsilon(y)$, (3) $B_\varepsilon(y) \subseteq B_\delta(x)$.*

The *morphisms* of topological spaces X and Y are *continuous* functions, namely, those functions $f : X \to Y$ such that any inverse image of an open set is open. In metric terms, this means that for every $x \in X$ and every $\varepsilon > 0$ there exists a $\delta > 0$ such that $d_X(x, x') < \delta$

implies $d_Y(f(x), f(x')) < \varepsilon$ for all $x' \in X$. Here, we denoted by d_X the metric on X and by d_Y the metric on Y. All constant functions are continuous, as are all *locally constant* functions, *i.e.*, functions $f : X \to Y$ that are constant in some open ball $B_\varepsilon(x)$ with positive radius $\varepsilon > 0$ for every $x \in X$.

Topologies for standard set-theoretic constructions can be defined from their individual parts. For instance, the product topology of a countable collection of metric spaces X_i can be described by the metric $d : X \times X \to [0, \infty)$ with

$$d(x, y) = \sum_{i \in \mathbb{N}} 2^{-i} \frac{d_i(x_i, y_i)}{1 + d_i(x_i, y_i)} \quad .$$

We use the product metric to extend the notion of indistinguishability of local views of configurations (these concepts are formally defined in Section 2.3) to a metric on infinite executions. It has the following property:

▶ **Lemma 2.2** ([5, § 2.3, Proposition 4]). *Let $(X_i)_{i \in \mathbb{N}}$ be a countable collection of metric spaces and let $X = \prod_{i \in \mathbb{N}} X_i$ be their product equipped with the product metric. For all metric spaces Y and all functions $g : Y \to X$, the following are equivalent:*

1. *The function g is continuous.*
2. *The function $\pi_i \circ g$ is continuous for all $i \in \mathbb{N}$ where $\pi_i : X \to X_i$ is the projection on the component i.*

The disjoint-union topology of the disjoint union $X = \bigsqcup_{i \in I} X_i$ is described by the metric $d : X \times X \to [0, \infty)$ with $d(x, y) = d_i(x, y)$ if there is an index $i \in I$ such that both x and y are elements of X_i, and $d(x, y) = 2$ else. We use the disjoint-union metric to get a global metric from those defined for the local views of each process, with the following property:

▶ **Lemma 2.3** ([5, § 2.4, Proposition 6]). *Let $(X_i)_{i \in I}$ be a collection of metric spaces and let $X = \bigsqcup_{i \in I} X_i$ be their disjoint union equipped with the disjoint-union metric. For all metric spaces Y and all functions $g : X \to Y$, the following are equivalent:*

1. *The function g is continuous.*
2. *The function $g \circ \varphi_i$ is continuous for all $i \in I$ where $\varphi_i : X_i \to X$ is the embedding of X_i into X.*

2.3 System Model

Let $T = (\mathcal{I}, \mathcal{O}, \Delta)$ any task. Since our goal is to give a very general characterization of task solvability, we work with an abstract system model that hides most of the operational details, such as semantics of shared registers or guarantees of message delivery. We instead focus on the structure of the set of executions induced by the local *indistinguishability* relations, *i.e.*, by the processes' local views. We further assume that actions taken by processes do not influence the set of possible executions. That is, we assume the existence of *full-information* executions, on which we base our characterization. A full-information execution is a sequence of configurations. A configuration is a vector with the process states and the state of the environment (*e.g.*, shared memory, messages in transit) in its entries. In a full-information execution, every process relays all the information it gathered to all other processes whenever it can. This includes its input value, the order and contents of events it perceived, and the information relayed to it by others. In particular, we assume that there are no size constraints on messages or shared memory.

Formally, let Exec be the set of full-information executions of n processes in which initial configurations are chosen according to the input complex $\mathcal{I}$. We assume the existence of projection functions $\pi_p : \mathsf{Exec} \to \mathsf{View}_p$ from executions to sequences of local views of

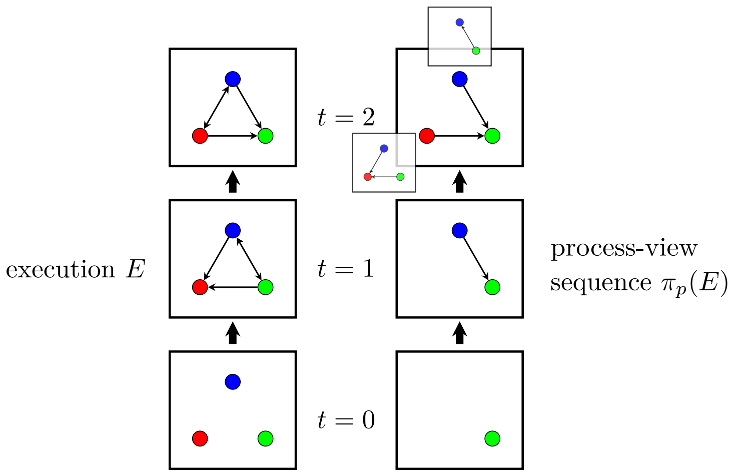


Figure 1 Prefix of a full-information execution E (left) and process-view projection $\pi_p(E)$ (right) of a synchronous message-passing system with dynamic communication graphs. The depicted prefix includes the initial configuration as well as the first two communication rounds. Process p is the green (lower right) process. Initially, after round 0, process p only knows its own initial value. After the first round, process p also knows the blue (upper) process's initial value as well as the fact that directed edge from the blue to the green process was present in the communication graph of the first round. After the second round, process p learned the initial value of the red (lower left) process, its own incoming edges of the second round's communication graph, as well as the views of the blue and the red processes after the first round.

process p. These sequences can be finite or infinite. Its element with index t contains the local view of process p right after its t^{th} step in the execution. The set of process views of executions in which process p is correct will be denoted by CView_p.

A step is defined as a possibility to irrevocably decide. That is, the t^{th} step of process p is process p's t^{th} possibility to decide a value (or not) in the execution. We allow processes to decide in their initial state, *i.e.*, in their step with index $t = 0$. Step counts are local to a process and need not be synchronized among processes. A process that only has finitely many steps is called *faulty* in the execution. For an execution $E \in \mathsf{Exec}$ we write $\mathsf{Correct}(E) \subseteq \Pi$ for the set of correct (non-faulty) processes in the execution. A *participating* process $p \in \mathsf{Part}(E) \subseteq \Pi$ is one that takes at least one step. We have that $\mathsf{Correct}(E) \subseteq \mathsf{Part}(E)$. We write $\mathsf{Init}_p(E) \in V^{\text{in}}$ for the initial value of process p in execution E. The concrete forms of executions and local views depend on the specifics of the computational model. Fig. 1 depicts an example execution and process-view sequence.

A (decision) *protocol* is a function from local views to $V^{\text{out}} \cup \{\bot\}$ with $\bot \notin V^{\text{out}}$ such that decisions are irrevocable: if some view is mapped to a decision value $v \in V^{\text{out}}$, then all its successor views are also mapped to v. A process p thus has at most one decision value in every execution E, which we denote by $\mathsf{Decision}_p(E) \in V^{\text{out}}$. A protocol *solves* a task $T = (\mathcal{I}, \mathcal{O}, \Delta)$ if it satisfies the following two conditions in every execution $E \in \mathsf{Exec}$:

- Every correct process $p \in \mathsf{Correct}(E)$ has a decision value in E.
- We have $\{v(p, \mathsf{Decision}_p(E)) \mid p \in \mathsf{Correct}(E)\} \in \Delta(\sigma)$ where $\sigma = \{v(p, \mathsf{Init}_p(E)) \mid p \in \mathsf{Part}(E)\}$.

Example: Lossy-Link Model. The lossy-link model [21] is a synchronous computation model with $n = 2$ processes, p and q, that communicate via message passing. The communication graph can change from round to round. In each round, the adversary chooses one of three

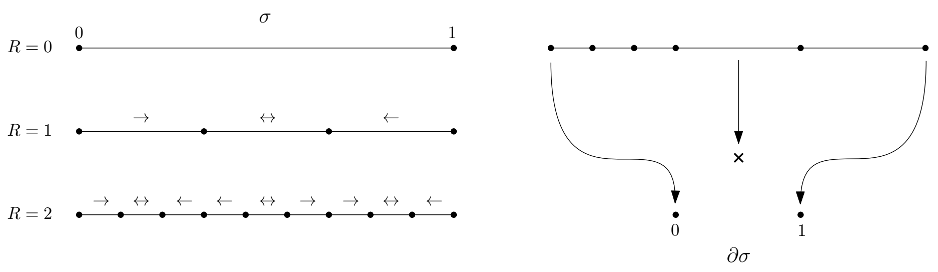


Figure 2 Subdivisions in the IIS model for two processes.

communication graphs: $\leftarrow$, $\rightarrow$, or $\leftrightarrow$. In a round with communication graph $\leftarrow$, only the message from the right to the left processes arrives, the other message is lost. In a round with communication graph $\rightarrow$, only the message from the left to the right processes arrives, the other message is lost. In a round with communication graph $\leftrightarrow$, both messages arrive and no message is lost. In a full-information execution, each process starts out by sending its initial value and then records all received messages in subsequent rounds, relaying this information to the other process. In this model, there is no notion of faulty processes; the only source of uncertainty is the communication. We thus have $\mathsf{Part}(E) = \mathsf{Correct}(E) = \{p, q\} = \Pi$ for every full-information execution E, and thus $\mathsf{CView}_p = \mathsf{View}_p$ and $\mathsf{CView}_q = \mathsf{View}_q$.

Since both processes are correct in every execution, both processes are participating, *i.e.*, $\mathsf{Part}(E) = \Pi$.

3 The Need of Continuity

This section explains the need of continuity of simplicial maps to model protocols in non-compact models. This is done using in part the example in [12] showing a flaw in the attempt to generalize the ACT [11].

For a system with two processes, left and right, the compact IIS model can be equivalently defined as the lossy-link model described in Section 2.3. Thus, IIS for two processes consists of all infinite sequences of communication graphs $\leftarrow$, $\rightarrow$, or $\leftrightarrow$, each graph specifying the communication that occurs in a round. A sub-IIS model is any subset of IIS.

Let us consider an *inputless* version of the consensus task where the left process has fixed input 0 and the right process has fixed input 1. Then, the input complex of the task $\mathcal{I}$ is the complex made of the edge $\sigma = \{0, 1\}$ and its faces (processes are identified with their inputs), the output complex $\mathcal{O}$ has simplexes $\{0\}$ and $\{1\}$, and Δ maps σ to $\mathcal{O}$, and each $\{i\}$ to itself. Complexes $\mathcal{I}$ and $\mathcal{O}$ will be denoted σ and $\partial\sigma$, respectively.

The topology of the IIS executions is well understood: the complex modeling all configurations at the end of round R is a *finite* subdivision of the input complex σ (basically a subdivision of the real interval $[0, 1]$), and as R increases, the subdivision gets finer. Concretely, it is the R-th standard chromatic subdivision. Figure 2(left) shows the subdivisions for the first two rounds, where, for example, the left-most and right-most edges of the second subdivision correspond to the configurations at the end of the finite solo executions $\rightarrow, \rightarrow$ and $\leftarrow, \leftarrow$, respectively, where a process does not hear from the other, and the central edge corresponds to $\leftrightarrow, \leftrightarrow$, where processes hear from each other.

A key property of round-structured compact models like IIS is that, for any protocol solving a task, there is a *finite* round R such that all correct processes make a decision at round R, at the latest (assuming $\mathcal{I}$ is finite). With this property, it is simple to see that consensus is impossible in IIS (see the right side of Figure 2):

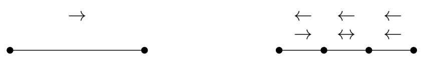


Figure 3 A possible subdivision for the sub-IIS model M_1.

1. For any round R, the complex corresponding to the *decided states* of a hypothetical protocol P, is a finite subdivision $\mathcal{K}$ of σ, *i.e.*, $|\mathcal{K}| = |\sigma|$. (Recall that $|\mathcal{K}|$ is the geometric realization of $\mathcal{K}$.) The subdivision might be irregular because processes might make decisions at different rounds; processes keep running after decision, hence an edge models infinitely many infinite executions, all of them sharing the finite prefix where the decisions are made.
2. P must map each vertex (state) of $\mathcal{K}$ to an output in $\partial\sigma$, with the restriction that the left-most vertex must be mapped to 0 and the right-most vertex must be mapped to 1, as they correspond to solo executions, hence, by validity of the consensus task (*i.e.*, $\Delta(\{i\}) = \{i\}$), the process that only sees its input is forced to decide it.
3. Since P solves consensus, it induces a simplicial map $\delta : \mathcal{K} \to \partial\sigma$, which, as $\mathcal{K}$ is finite, *necessarily* induces a continuous map $|\delta| : |\mathcal{K}| \to |\partial\sigma|$. The map $|\delta|$ is ultimately a continuous map $|\sigma| \to |\partial\sigma|$ that maps the boundary of σ to itself.
4. Finally, this continuous map does not exist because $|\sigma|$ is solid whereas $|\partial\sigma|$ is disconnected.

The argument above goes from protocols to simplicial maps. In models like IIS, the other direction is also true. Namely, for any given task $T = (\mathcal{I}, \mathcal{O}, \Delta)$, for any complex $\mathcal{K}$ related to $\mathcal{I}$ that satisfies some model-dependent properties, any simplicial map from $\mathcal{K}$ to $\mathcal{O}$ that agrees with Δ, induces a protocol for T. Thus, to show that a task is solvable in two-process IIS, it suffices to exhibit a finite, possibly irregular, subdivision of the input complex, in the style of the one in Figure 2(left), and a simplicial map that is valid for the task.

The main aim of [11] is to generalize the approach above that equates simplicial maps and protocols to arbitrary sub-IIS models, in order to exploit the already known topology of IIS. The high-level idea is that the complexes that model a sub-IIS model are still subdivisions but not necessarily of the input complex, and not necessarily finite.

Let us consider first the sub-IIS model M_1 with all infinite executions of the form $\leftarrow$ followed by any infinite sequence with $\leftarrow$, $\rightarrow$, or $\leftrightarrow$ (intuitively right goes first), or $\rightarrow$ followed by any infinite sequence with $\leftarrow$, $\rightarrow$, or $\leftrightarrow$ (intuitively left goes first). It can be seen that consensus is solvable in this model: since $\leftrightarrow$ cannot happen in the first round, the process that receives no message in the first round is the "winner". Figure 3 shows an irregular subdivision that models all executions of M_1; for example, the right-most edge corresponds to all executions of M_1 with prefix $\leftarrow, \leftarrow$. Intuitively, in the subdivision, in some executions processes decide in round one (represented by the edge at the left), and in the remaining executions processes decide in round two (represented by the three edges at the right). Clearly, there is a simplicial map from such a disconnected subdivision to $\partial\sigma$ that agrees with consensus. This simplicial map induces a consensus protocol for M_1.

The argument above works well because the model is compact, hence finite subdivisions are able to capture all its executions. However, in non-compact models, some executions can only be modeled through infinite subdivisions, which implies that simplicial maps are not necessarily protocols.

Consider now the sub-IIS model M_2 obtained by removing from IIS the infinite execution E described by the sequence $\leftrightarrow, \leftarrow, \leftarrow, \ldots$ This model is not compact because it contains any infinite execution with a *finite* prefix (of any length) of E, but it does not contain E itself, the limit execution. As said, a crucial property of non-compactness is that the executions of

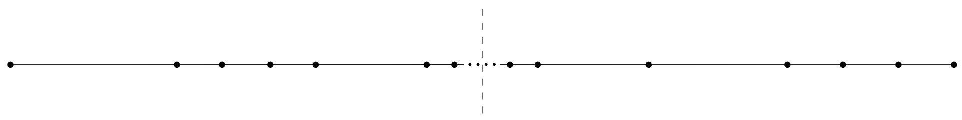


Figure 4 An schematic representation of an infinite subdivision for the sub-IIS model M_2.

the model cannot be captured by a finite subdivision. Intuitively, an edge can only model executions that have a common finite prefix of E of length x, but in M_2 there are executions with a prefix larger than x, hence these executions are not captured by the edge; if the subdivision is finite, there are necessarily executions that are not modeled by any edge.

Figure 4 contains a schematized infinite subdivision $\mathcal{K}$ that indeed captures all executions of M_2. Intuitively, there are infinitely many edges that get closer and closer to the point that represents the removed execution E (depicted as a vertical dashed line at the center), but no edge actually "crosses" it (as E is not in M_2). Thus the simplicial complex $\mathcal{K}$ is disconnected, and there is a simplical map from $\mathcal{K}$ to $\partial\sigma$ that agrees with consensus. Although all executions are captured in the infinite subdivision, such a simplicial map *does not* imply a protocol. The intuition is that there is a sudden jump in the decisions around E, which ultimately implies that the decision in executions that are similar enough to the removed limit execution E are not consistent, namely, they cannot be produced by a protocol.

It turns out that the topological space $|\mathcal{K}|$ is actually a subdivision of $|\sigma|$: in the limit, $|\mathcal{K}| = |\sigma|$. Thus, the infinite subdivision $\mathcal{K}$ describes a space that is not disconnected! Moreover, for any infinite subdivision that models M_2, the space associated with it is connected, *i.e.*, this is an invariant of the model of computation. Any simplicial map that intends to capture a protocol should consider that $|\mathcal{K}| = |\sigma|$. This is precisely captured by demanding that the induced map $|\mathcal{K}| \to |\mathcal{O}|$ must be continuous (hence smooth around E). Therefore, there is no continuous map $|\sigma| \to |\partial\sigma|$ that maps the boundary of σ to itself, and indeed consensus is not solvable in this model [9, Theorem III.8].

Formalizing this seemingly simple observation in arbitrary models of computation is not obvious, and it requires a combination of combinatorial topology techniques and point-set topology techniques, as is done in the following sections. Intuitively, distance functions in point-set topology are used to equip protocol complexes with a topology that in turn yields a correspondence between continuous simplicial maps and protocols.

4 Proof of the Generalized Asynchronous Computability Theorem with an Application to Set Agreement

In this section we use the definitions and notation of Gafni, Kuznetsov, and Manolescu [11] for sub-IIS models. They introduced the notion of *terminating subdivisions* of the input complex $\mathcal{I}$ of a task. The idea is to repeatedly subdivide all simplexes via the standard chromatic subdivision, except those that are already marked as terminated. The terminated simplexes model configurations where processes have decided.

Formally, a terminating subdivision $\mathscr{T}$ is specified by a sequence of chromatic complexes $\mathcal{I}_0, \mathcal{I}_1, \ldots$ and a sequence of subcomplexes $\Sigma_0 \subseteq \Sigma_1 \subseteq \ldots$ such that for all $k \geq 0$: (1) Σ_k is a subcomplex of $\mathcal{I}_k$ (each $\mathcal{I}_k$ is a *non-uniform* subdivision [16]) and (2) $\mathcal{I}_0 = \mathcal{I}$ and $\mathcal{I}_{k+1}$ is obtained from $\mathcal{I}_k$ by the *partial* chromatic subdivision in which the simplexes in Σ_k are not further subdivided (the terminated simplexes), and each simplex $\tau \notin \Sigma_k$ is replaced with its standard chromatic subdivision $\mathrm{Chr}\,\tau$. Precisely, we replace a simplex σ in $\mathcal{I}_k$ by a

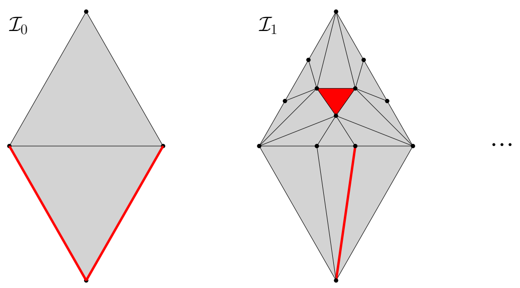


Figure 5 First two complexes of a three-process terminating subdivision.

coarser subdivision than $\operatorname{Chr}\sigma$. Whereas the vertices of $\operatorname{Chr}\sigma$ are pairs (p, σ') with $p \in \Pi$ and $\sigma' \subseteq \sigma$, in $\mathcal{I}_{k+1}$ we consider the pairs (p, σ') of that form such that either $\sigma' \notin \Sigma_k$, or σ' consists of a single vertex in Σ_k.

Figure 5 schematizes a terminating subdivision where $\mathcal{I}$ is made of two triangles and terminated simplexes are marked in red.

A simplex of Σ_k, for some k, is called *stable*. The simplicial complex $K(\mathscr{T})$ is the union of all Σ_k; $K(\mathscr{T})$ might be infinite.

The vertices of $K(\mathscr{T})$ are naturally embedded in the geometric realization of $\mathcal{I}$ by their definition as a vertex of the repeated chromatic subdivision $\operatorname{Chr}^k \mathcal{I}$ (recall that $|\operatorname{Chr}^k \mathcal{I}| = |\mathcal{I}|$, for every $k \geq 0$). In particular, we identify the geometric realization $|K(\mathscr{T})|$ with a subset of $|\mathcal{I}|$. Every IIS execution can be described as an infinite sequence of simplexes $\sigma_0, \sigma_1, \ldots$ such that $\sigma_k \in \operatorname{Chr}^k \mathcal{I}$, for every $k \geq 0$.

A terminating subdivision is *admissible* for a sub-IIS model M if $K(\mathscr{T})$ covers all executions of M, namely, for each execution $\sigma_0, \sigma_1, \ldots$ of M, there is a k such that $|\sigma_k| \subseteq |\tau|$, for some terminated simplex $\tau \in \Sigma_k$.

▶ **Theorem 4.1.** *A sub-IIS model M solves a task $T = (\mathcal{I}, \mathcal{O}, \Delta)$ if and only if there exists a terminating subdivision $\mathscr{T}$ of $\mathcal{I}$ and a chromatic simplicial map $\delta\colon K(\mathscr{T}) \to \mathcal{O}$ such that:*
(a) *$\mathscr{T}$ is admissible for the model M.*
(b) *For any simplex σ of $\mathcal{I}$, if τ is a stable simplex of $\mathscr{T}$ such that $|\tau| \subseteq |\sigma|$, then $\delta(\tau) \in \Delta(\sigma)$.*
(c) *$|\delta|$ is continuous.*

Proof sketch. ($\Rightarrow$): This direction consisting in showing that the geometric realization of the map δ as constructed in the proof of Gafni, Kuznetsov, and Manolescu [11, Theorem 6.1] is continuous by generalizing the proof given by Godard and Perdereau [12, Theorem 33] for the consensus task with two processes.

($\Leftarrow$): We modify the protocol that is constructed in the proof of Gafni, Kuznetsov, and Manolescu [11, Theorem 6.1] for process p to decide in round k if the set

$$B_k(v) = \{w \in V(K(\mathscr{T})) \mid d(v, w) \leq D_k \wedge \chi(w) = p\}$$

only contains vertices that are mapped to the same output vertex by δ, where v is the view of the process in round k. This condition eventually holds since the subset topology on the geometric realization of the output vertices $V(\mathcal{O})$ is discrete; thus, δ is locally constant. ◀

We now use Theorem 4.1 to derive a condition for the impossibility of $(n-1)$-set agreement task in IIS-sub models. Recall that in this task each process is required to eventually decide an input value (termination) of a process participating in the execution (validity) such that no more than $n - 1$ distinct values are decided (agreement).

Let $\Pi = \{p_0, p_1, \ldots, p_{n-1}\}$. For simplicity, we focus on the *inputless* version of the set agreement task, where each process p_i, $0 \leq i \leq n-1$, has fixed input i in every execution, and thus the task is the triple $T = (\mathcal{I}, \mathcal{O}, \Delta)$, where the input complex $\mathcal{I}$ is made of all faces of simplex $\sigma = \{0, 1, \ldots, n-1\}$, and for simplicity it is denoted σ, the output complex $\mathcal{O}$, denoted $\partial\sigma$, is the complex with all *proper* faces of σ, and Δ maps every proper face $\sigma' \subset \sigma$ to the complex with all faces of σ', and maps σ to $\partial\sigma$.

▶ **Theorem 4.2.** *Let M be an IIS-sub model such that for any termination subdivision $\mathscr{T}$ of σ that is admissible for M, $|\sigma| = |K(\mathscr{T})|$. Then, $(n-1)$-set agreement is impossible in M.*

5 Characterization of Task Solvability in General Models

In this section we present a topological solvability characterization in general models, hence showing that the topology approach is applicable in all models of computation. As anticipated, the characterization demands simplicial maps to be continuous, which is particularly relevant if complexes are infinite. Differently from sub-IIS models in the previous section, where continuity naturally arises in protocols complexes as they are subdivisions (hence embedded in a Euclidean space), the general case requires to equip protocol complexes with a topology, which is used to capture continuity.

Recall that the set of process views of executions in which process p is correct is denoted CView_p. These are the executions in which we demand process p to decide on an output value. We always have $\mathsf{CView}_p \subseteq \mathsf{View}_p$. For every process p we define a topology on the set CView_p of correct process-p local-view sequences induced by the distance function

$$d_p(\alpha, \beta) = 2^{-T_p(\alpha,\beta)}$$

where $T_p(\alpha, \beta)$ is defined as the smallest index at which the local views in the process-view sequences α and β differ. If no such index exists, then $T_p(\alpha, \beta) = \infty$. This means that the distance between two process-view sequences is smaller the later the process can detect a difference between the two. If α and β do not differ in any index, then $\alpha = \beta$ and $d_p(\alpha, \beta) = 2^{-\infty} = 0$. A variant of this distance function, which considers complete executions instead of local views, was introduced by Alpern and Schneider [2].

We first establish that the distance function d_p is an ultrametric.

▶ **Lemma 5.1.** *The distance function d_p is an ultrametric on CView_p.*

In the next lemma, we establish the fundamental fact that the decision functions for process p are exactly the continuous functions $\mathsf{CView}_p \to V(\mathcal{O})$ when using d_p on CView_p and the discrete metric on $V(\mathcal{O})$.

▶ **Lemma 5.2.** *Let $\delta_p : \mathsf{CView}_p \to V(\mathcal{O})$ be a function. The following are equivalent:*
1. *There is a protocol such that process p decides the value $\delta_p(\alpha)$ in every execution $E \in \mathsf{Exec}$ with local view $\pi_p(E) = \alpha \in \mathsf{CView}_p$.*
2. *The function δ_p is continuous when equipping CView_p with the topology induced by d_p and $V(\mathcal{O})$ with the discrete topology.*

To formulate and prove our characterization for the solvability of tasks in general models, we define a structure that combines the notions of chromatic simplicial complexes and the notion of point-set topology of sequences of local views. Formally, a *topological chromatic simplicial complex* is a chromatic simplicial complex whose set of vertices is equipped with a topology. A vertex map between two topological chromatic simplicial complexes is a *morphism* if it is continuous, chromatic, and simplicial.

The protocol complex $\mathcal{P}$ is a (possibly infinite) topological chromatic simplicial complex defined as follows. The set of vertices of $\mathcal{P}$ is the disjoint union $V(\mathcal{P}) = \bigsqcup_{p\in\Pi} \mathsf{CView}_p$ of the correct local-view spaces. The vertices from CView_p are colored with the process name p. We equip the set of vertices with the disjoint-union topology, *i.e.*, the finest topology that makes all embedding maps $\iota_p : \mathsf{CView}_p \to V(\mathcal{P})$ continuous.

A set σ of vertices of $\mathcal{P}$ is a simplex of $\mathcal{P}$ if and only if the local views are consistent with the views of correct processes in an execution, *i.e.*, if it is of the form

$$\sigma = \{\pi_p(E) \mid p \in P\}$$

for some execution $E \in \mathsf{Exec}$ and some set $P \subseteq \mathsf{Correct}(E)$.

The execution map $\Xi : \mathcal{I} \to 2^{\mathcal{P}}$ is defined by mapping every input simplex in $\mathcal{I}$ to the local views of correct processes of executions in which the initial values of participating processes are as in the input simplex. Formally,

$$\Xi(\sigma) = \big\{\{\pi_p(E) \mid p \in P\} \mid E \in \mathsf{Compatible}(\sigma) \wedge P \subseteq \mathsf{Correct}(E)\big\}$$

where $\mathsf{Compatible}(E) \subseteq \mathsf{Exec}$ denotes the set of executions that are compatible with the initial values described by the input simplex σ. That is,

$$\mathsf{Compatible}(\sigma) = \{E \in \mathsf{Exec} \mid \mathsf{Part}(E) \subseteq \chi[\sigma] \wedge \forall p \in \mathsf{Part}(E)\colon \mathsf{Init}_p(E) = \ell(\pi_p(\sigma))\}$$

where $\pi_p(\sigma)$ denotes the unique vertex of the simplex σ with the color p, if it exists, and $\ell(\pi_p(\sigma))$ is the input (label) of vertex $\pi_p(\sigma)$. The execution map Ξ assigns a subcomplex of $\mathcal{P}$ to every input simplex σ in $\mathcal{I}$. As in the classical finite-time setting [13, Definition 8.4.1], the next lemma shows that it is a carrier map:

▶ **Lemma 5.3.** *The execution map Ξ is a carrier map such that $\mathcal{P} = \bigcup_{\sigma\in\mathcal{I}} \Xi(\sigma)$.*

In contrast to the classical finite-time setting, however, the execution map is not necessarily rigid. Whether it is depends on whether any finite execution prefix can be extended to a fault-free execution. This is not the case, *e.g.*, in many synchronous models. If Ξ is not rigid, then, by definition, it is *a fortiori* not chromatic. It does, however, satisfy the inclusion

$$\big\{\chi(v) \mid v \in V(\Xi(\sigma))\big\} \ \subseteq\ \chi[\sigma]$$

for all input simplices $\sigma \in \mathcal{I}$. In other words, the colors of $\Xi(\sigma)$ are included in the colors of σ; no new process names appear. It turns out that the stronger assumptions of rigidity or chromaticity are not necessary to show our solvability characterization.

▶ **Theorem 5.4.** *The task $T = (\mathcal{I}, \mathcal{O}, \Delta)$ is solvable if and only if there exists a continuous chromatic simplicial map $\delta : \mathcal{P} \to \mathcal{O}$ such that $\delta \circ \Xi$ is carried by Δ.*

Proof. ($\Rightarrow$): Assume that there is a protocol that solves task T. Define the vertex map $\delta : \mathcal{P} \to \mathcal{O}$ by setting $\delta(\alpha)$ to be the vertex of $\mathcal{O}$ with color p and label v where p is the unique process such that $\alpha \in \mathsf{CView}_p$ and v is the decision value of process p in an execution with local-view sequence α when executing the protocol.

The map δ is continuous on each individual CView_p by Lemma 5.2. By Lemma 2.3, it is thus continuous on their disjoint union $\mathcal{P}$. The map δ is chromatic since the color of the vertex $\alpha \in \mathsf{CView}_p$ is p, as is the color of $\delta(\alpha)$.

To prove that δ is simplicial, let φ be a simplex of $\mathcal{P}$. Then, by definition, there exists an execution $E \in \mathsf{Exec}$ and a set $P \subseteq \mathsf{Correct}(E)$ such that $\varphi = \{\pi_p(E) \mid p \in P\}$. Set $\sigma = \{v(p, \mathsf{Init}_p(E)) \mid p \in \Pi\}$ and $\tau = \{v(p, \mathsf{Decision}_p(E) \mid p \in \mathsf{Correct}(E)\}$. Then, since the protocol solves task T, we have $\tau \in \Delta(\sigma)$. By definition of δ, we then have $\delta[\varphi] = \{v(p, \mathsf{Decision}_p(E)) \mid p \in P\} \subseteq \tau \in \Delta(\sigma) \subseteq \mathcal{O}$, which means that $\delta[\varphi] \in \mathcal{O}$ and hence that δ is simplicial.

It remains to prove that $\delta \circ \Xi$ is carried by Δ. So let $\sigma \in \mathcal{I}$ and $\tau \in (\delta \circ \Xi)(\sigma)$. We need to show that $\tau \in \Delta(\sigma)$. By the definitions of Ξ and δ, there exists an execution $E \in \mathsf{Compatible}(\sigma)$ and a set $P \subseteq \mathsf{Correct}(E)$ such that $\tau = \{v(p, \mathsf{Decision}_p(E)) \mid p \in P\}$. Since the protocol solves task T, we have $\tau' = \{v(p, \mathsf{Decision}_p(E)) \mid p \in \mathsf{Correct}(E)\} \in \Delta(\sigma')$ where $\sigma' = \{v(p, \mathsf{Init}_p(E)) \mid p \in \mathsf{Part}(E)\}$. Since $\tau \subseteq \tau'$ and $\Delta(\sigma')$ is a simplicial complex, we deduce that $\tau \in \Delta(\sigma')$. Now, because $E \in \mathsf{Compatible}(\sigma)$, we have $\mathsf{Part}(E) \subseteq \chi[\sigma]$ and $\sigma' \subseteq \sigma$. It thus follows that $\tau \in \Delta(\sigma') \subseteq \Delta(\sigma)$ because Δ is a carrier map.

($\Leftarrow$): The restriction δ_p of δ to the set CView_p is continuous because δ is. By Lemma 5.2 there hence exists a protocol such that every process p decides the value $\ell(\delta(\pi_p(E))) \in V^{\text{out}}$ for every execution E in which p is correct.

Let $E \in \mathsf{Exec}$ be any execution and define the sets $\sigma = \{v(p, \mathsf{Init}_p(E)) \mid p \in \mathsf{Part}(E)\}$ and $\tau = \{v(p, \mathsf{Decision}_p(E)) \mid p \in \mathsf{Correct}(E)\}$. To show that the protocol solves task T, it remains to show that $\tau \in \Delta(\sigma)$. Since $\delta \circ \Xi$ is carried by Δ, it suffices to prove $\tau \in (\delta \circ \Xi)(\sigma)$. Setting $\varphi = \{\pi_p(E) \mid p \in \mathsf{Correct}(E)\}$, we have $\tau = \delta[\varphi]$. We are thus done if we show $\varphi \in \Xi(\sigma)$. But this follows from $E \in \mathsf{Compatible}(\sigma)$, which is true by construction of σ. ◀

6 Relationship to the Classical Finite-Time Approach

In this section, we formalize the relationship between our infinite protocol complex used for the general solvability characterization in Theorem 5.4 and the classically studied finite-time protocol complexes. Besides demonstrating that the classical formalism is a special case of ours, we show the finite-time approach is sufficient for all compact models. More specifically, we show that it is possible to restrict the study to finite-time protocols if the computational model is compact. Formally, a topological space is *compact* if every open cover has a finite subcover. Many computational models that are defined by safety predicates are compact. We use the concept of projective limit from category theory [19] to formalize the relationship between finite-time and infinite-time complexes. In particular, we show that the infinite-time complex is the projective limit of the finite-time complexes if the model is compact.

Finite-Time Complexes. For every nonnegative integer T, we define the time-T protocol complex $\mathcal{P}|_T$ as follows:

- The set of vertices of $\mathcal{P}|_T$ is the disjoint union of the sets $\mathsf{CView}_p|_T$ where p varies in the set Π of processes.
- The set $\mathsf{CView}_p|_T$ is defined as the set of open balls of radius $\varepsilon = 2^{-T}$ in CView_p. These balls are either identical or disjoint by Lemma 2.1.
- All vertices of $\mathsf{CView}_p|_T$ are colored with p.
- A set of vertices of $\mathcal{P}|_T$ is a simplex of $\mathcal{P}|_T$ if and only if there is a simplex of $\mathcal{P}$ that is formed by choosing one element in each vertex of the set.
- The topology on $V(\mathcal{P}|_T)$ is the discrete topology.

This definition makes $\mathcal{P}|_T$ a topological chromatic simplicial complex. As a chromatic simplicial complex, it is isomorphic to the classical finite-time construction of protocol complexes [13]. The finite-time execution map $\Xi_T : \mathcal{I} \to 2^{\mathcal{P}|_T}$ is defined by

$$\Xi_T(\sigma) = \{B_T[\tau] \mid \tau \in \Xi(\sigma)\}$$

where $B_T(\alpha) = \{\beta \in V(\mathcal{P}) \mid d(\alpha, \beta) < 2^{-T}\}$ is the function that takes each vertex α of $\mathcal{P}$ to the open 2^{-T}-ball in which it is included.

Projective Limits. We will show that, if the model is compact, then $\mathcal{P}$ is the limit of the $\mathcal{P}|_T$ in a precise sense. For this, we use the notion of projective limits from category theory [19], which we introduce in this subsection.

A *category* is a class of *objects* and a class of *morphisms* between objects. Every morphism $f : X \to Y$ is assigned a domain object X and a codomain object Y. For compatible morphisms $f : X \to Y$ and $g : Y \to Z$, the composition $g \circ f$ is a morphism $X \to Z$. The composition operator is required to be associative. For every object X, the existence of an identity morphism $\mathrm{id}_X : X \to X$ is required. The identity morphism satisfies $f \circ \mathrm{id}_x = f$ for all morphism $f : X \to Y$ with domain X and $\mathrm{id}_X \circ g = g$ for all morphisms $g : Z \to X$ with codomain X.

A sequence $(X_T)_{T\geq 0}$ of objects of a category can be transformed into an *inverse system* by specifying a family $(f_{S,T})_{0\leq S\leq T}$ of morphisms $f_{S,T} : X_T \to X_S$ such that $f_{T,T} = \mathrm{id}_{X_T}$ and $f_{R,T} = f_{R,S} \circ f_{S,T}$ for all $0 \leq R \leq S \leq T$. The *projective limit* of the sequence is then an object X together with morphisms $\pi_T : X \to X_T$ such that $\pi_S = f_{S,T} \circ \pi_T$ for all $0 \leq S \leq T$ and with the universal property that for any other such object Y and morphisms $\psi_T : Y \to X_T$, there exists a unique morphism $u : Y \to X$ such that the following diagram commutes for all $0 \leq S \leq T$:

$$\begin{tikzcd}
 & Y \arrow[d, "u"] \arrow[ddl, "\psi_T"'] \arrow[ddr, "\psi_S"] & \\
 & X \arrow[dl, "\pi_T"] \arrow[dr, "\pi_S"'] & \\
X_T \arrow[rr, "f_{S,T}"'] & & X_S
\end{tikzcd}$$

For every pair of integers S and T, $0 \leq S \leq T$, define the vertex maps $f_{S,T} : \mathcal{P}|_T \to \mathcal{P}|_S$ by setting $f_{S,T}(B)$ to be the unique open 2^{-S}-ball of $\mathcal{P}|_S$ in which the open 2^{-T}-ball B of $\mathcal{P}|_T$ is included. These are morphisms between topological chromatic simplicial complexes and they satisfy $f_{R,T} = f_{R,S} \circ f_{S,T}$ for all $0 \leq R \leq S \leq T$. This makes the sequence of the $\mathcal{P}|_T$ an inverse system.

▶ **Lemma 6.1.** *The projective limit of the sequence of complexes $\mathcal{P}|_T$ exists.*

We can equip the set of executions with the metric $d(E, E') = 2^{-K}$ where $K = \inf\{k \geq 0 \mid E_k \neq E'_k\}$, which measures how many configurations are identical in two execution prefixes [2]. With this topology on Exec, the projection maps $\pi_p : \mathsf{Exec} \to \mathsf{View}_p$ are continuous. In fact, continuity of the map means that each local view needs to be determined by some finite prefix of the execution. We have the following lemma:

▶ **Lemma 6.2.** *If* Exec *is compact, then $V(\mathcal{P})$ is compact as well, and $\mathcal{P}$ is the projective limit of the $\mathcal{P}|_T$.*

Sufficiency of Finite-Time Complexes for Compact Models. We can now state the fact that finite-time protocol complexes are sufficient to study compact models.

▶ **Theorem 6.3.** *If $V(\mathcal{P})$ is compact, then the following are equivalent:*
1. *The task $T = (\mathcal{I}, \mathcal{O}, \Delta)$ is solvable.*
2. *There is a continuous chromatic simplicial map $\delta : \mathcal{P} \to \mathcal{O}$ such that $\delta \circ \Xi$ is carried by Δ.*
3. *There is a time T such that there exists a chromatic simplicial map $\delta_T : \mathcal{P}|_T \to \mathcal{O}$ such that $\delta_T \circ \Xi_T$ is carried by Δ.*
4. *The task $T = (\mathcal{I}, \mathcal{O}, \Delta)$ is solvable in a bounded number of local steps per process.*

On the other hand, if $V(\mathcal{P})$ is not compact, then the equivalence in Theorem 6.3 need not hold, as is shown by the example in Section 3.

7 Conclusion

We put together combinatorial and point-set topological arguments to prove a generalized asynchronous computability theorem, which applies also to non-compact computation models. This relies on showing that in non-compact models, protocols solving tasks correspond to simplicial maps that need to be *continuous*. We show an application to the *set agreement* task. We also show that the usual finite-time protocol complex, where protocols and simplicial maps are the same, suffices for *all* compact models.

It would be interesting to find other computation models and tasks where our techniques, and the generalized ACT, in particular, can be applied. Another intriguing direction for future research is to characterize which computation models lead to non-compact topological objects.

References

1 Dan Alistarh, James Aspnes, Faith Ellen, Rati Gelashvili, and Leqi Zhu. Why extension-based proofs fail. In Moses Charikar and Edith Cohen, editors, *Proceedings of the 51st Annual ACM Symposium on Theory of Computing (STOC 2019)*, pages 986–996. ACM, New York, 2019.

2 Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 1985.

3 Hagit Attiya, Armando Castañeda, and Sergio Rajsbaum. Locally solvable tasks and the limitations of valency arguments. *Journal of Parallel and Distributed Computing*, 176:28–40, 2023. `doi:10.1016/j.jpdc.2023.02.002`.

4 Hagit Attiya and Sergio Rajsbaum. The combinatorial structure of wait-free solvable tasks. *SIAM Journal on Computing*, 31(4):1286–1313, 2002. `doi:10.1137/S0097539797330689`.

5 Nicolas Bourbaki. *General Topology. Chapters 1–4.* Springer, Heidelberg, 1989.

6 Armando Castañeda, Pierre Fraigniaud, Ami Paz, Sergio Rajsbaum, Matthieu Roy, and Corentin Travers. A topological perspective on distributed network algorithms. *Theoretical Computer Science*, 849:121–137, 2021.

7 Étienne Coulouma, Emmanuel Godard, and Joseph Peters. A characterization of oblivious message adversaries for which consensus is solvable. *Theoretical Computer Science*, 584:80–90, June 2015. `doi:10.1016/j.tcs.2015.01.024`.

8 Yannis Coutouly and Emmanuel Godard. A topology by geometrization for sub-iterated immediate snapshot message adversaries and applications to set-agreement. In Rotem Oshman, editor, *Proceedings of the 37th International Symposium on Distributed Computing (DISC 2023)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, 2023. To appear.

9 Tristan Fevat and Emmanuel Godard. Minimal obstructions for the coordinated attack problem and beyond. In *Proceedings of the 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2011)*, pages 1001–1011. IEEE, New York, 2011.

10 Pierre Fraigniaud, Ran Gelles, and Zvi Lotker. The topology of randomized symmetry-breaking distributed computing. In Avery Miller, Keren Censor-Hillel, and Janne H. Korhonen, editors, *Proceedings of the 40th ACM Symposium on Principles of Distributed Computing (PODC 2021)*, pages 415–425. ACM, New York, 2021. `doi:10.1145/3465084.3467936`.

11 Eli Gafni, Petr Kuznetsov, and Ciprian Manolescu. A generalized asynchronous computability theorem. In Shlomi Dolev, editor, *Proceedings of the 33rd ACM Symposium on Principles of Distributed Computing (PODC 2014)*, pages 222–231. ACM, New York, 2014.

12 Emmanuel Godard and Eloi Perdereau. Back to the coordinated attack problem. *Mathematical Structures in Computer Science*, 30(10):1089–1113, 2020. `doi:10.1017/S0960129521000037`.

13 Maurice Herlihy, Dmitry N. Kozlov, and Sergio Rajsbaum. *Distributed Computing Through Combinatorial Topology*. Morgan Kaufmann, Waltham, 2014. URL: `https://store.elsevier.com/product.jsp?isbn=9780124045781`.

14 Maurice Herlihy and Sergio Rajsbaum. Simulations and reductions for colorless tasks. In Darek Kowalski and Alessandro Panconesi, editors, *Proceedings of the 31st ACM Symposium on Principles of Distributed Computing (PODC 2012)*, pages 253–260. ACM, New York, 2012. doi:10.1145/2332432.2332483.

15 Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858–923, 1999.

16 Gunnar Hoest and Nir Shavit. Toward a topological characterization of asynchronous complexity. *SIAM Journal on Computing*, 36(2):457–497, 2006. doi:10.1137/S0097539701397412.

17 Petr Kuznetsov, Thibault Rieutord, and Yuan He. An asynchronous computability theorem for fair adversaries. In Calvin Newport and Idit Keidar, editors, *Proceedings of the 37th ACM Symposium on Principles of Distributed Computing (PODC 2018)*, pages 387–396. ACM, New York, 2018. URL: https://dl.acm.org/citation.cfm?id=3212765.

18 Ronit Lubitch and Shlomo Moran. Closed schedulers: a novel technique for analyzing asynchronous protocols. *Distributed Computing*, 8:203–210, 1995.

19 Saunders Mac Lane. *Categories for the Working Mathematician*. Springer, Heidelberg, 2nd edition, 1987.

20 Thomas Nowak, Ulrich Schmid, and Kyrill Winkler. Topological characterization of consensus under general message adversaries. In *Proceedings of the 38th ACM Symposium on Principles of Distributed Computing (PODC 2019)*, pages 218–227. ACM, New York, 2019. doi:10.1145/3293611.3331624.

21 Nicola Santoro and Peter Widmayer. Time is not a healer. In B. Monien and R. Cori, editors, *Proceedings of the 6th Annual Symposium on Theoretical Aspects of Computer Science (STACS 1989)*, pages 304–313. Springer, Heidelberg, 1989. doi:10.1007/bfb0028994.

22 Vikram Saraph, Maurice Herlihy, and Eli Gafni. Asynchronous computability theorems for t-resilient systems. In Cyril Gavoille and David Ilcinkas, editors, *Proceedings of the 30th International Symposium on Distributed Computing (DISC 2016)*, pages 428–441. Springer, Heidelberg, 2016. doi:10.1007/978-3-662-53426-7_31.

A Additional Details for Section 2 (Preliminaries)

A *simplicial* complex is a (possibly infinite) set V along with a (possibly infinite) collection $\mathcal{K}$ of finite subsets of V closed under containment *i.e.*, if $\sigma \in \mathcal{K}$ then $\sigma' \in \mathcal{K}$, for any $\sigma' \subseteq \sigma$. An element of V is called a *vertex* of $\mathcal{K}$, and the vertex set of $\mathcal{K}$ is denoted by $V(\mathcal{K})$. Each set in $\mathcal{K}$ is called a *simplex*. A subset of a simplex is called a *face* of that simplex. The *dimension of a simplex* σ, denoted $\dim \sigma$, is one less than the number of elements of σ, *i.e.*, $|\sigma| - 1$. The *dimension of a complex* is the smallest integer that upper bounds the dimension of any of its simplexes, or ∞ if there is no such bound. A simplex σ in $\mathcal{K}$ is called a *facet* of $\mathcal{K}$ if σ is not properly contained in any other simplex. A complex is *pure* if all its facets have the same dimension. We will focus on pure complexes, either finite or infinite.

Let $\mathcal{K}$ be a complex and σ be a simplex of it. The *star* of σ in $\mathcal{K}$ is the complex $\operatorname{st} \sigma = \{\tau \in \mathcal{K} \mid \sigma \subseteq \tau\}$.

Let $\mathcal{K}$ and $\mathcal{L}$ be complexes. A *vertex map* from $\mathcal{K}$ to $\mathcal{L}$ is a function $h : V(\mathcal{K}) \to V(\mathcal{L})$. If h also carries simplexes of $\mathcal{K}$ to simplexes of $\mathcal{L}$, it is called a *simplicial map*.

For two complexes $\mathcal{K}$ and $\mathcal{L}$, if $\mathcal{K} \subseteq \mathcal{L}$, we say $\mathcal{K}$ is a *subcomplex* of $\mathcal{L}$. Given two complexes $\mathcal{K}$ and $\mathcal{L}$, a *carrier map* $\Phi : \mathcal{K} \to 2^{\mathcal{L}}$ maps each simplex $\sigma \in \mathcal{K}$ to a subcomplex $\Phi(\sigma)$ of $\mathcal{L}$, such that for every two simplexes τ and τ' in $\mathcal{K}$ that satisfy $\tau \subseteq \tau'$, we have $\Phi(\tau) \subseteq \Phi(\tau')$. We say that Φ is *rigid* if for every $\sigma \in \mathcal{K}$, $\Phi(\sigma)$ is pure of dimension $\dim \sigma$.

A *geometric realization* of a complex $\mathcal{K}$ is an embedding of the simplexes of $\mathcal{K}$ into a real vector space such that, roughly speaking, intersections of simplexes are respected. All geometric realizations of a complex are topologically equivalent, *i.e.*, homeomorphic. Thus,

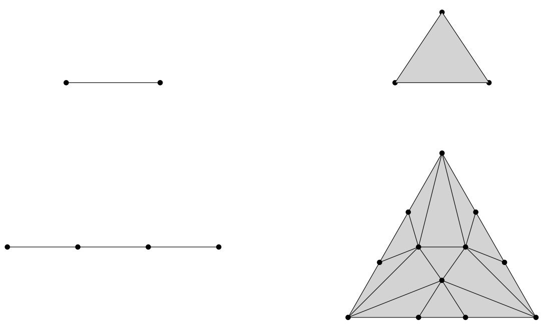


Figure 6 The standard chromatic subdivision of an edge and of a triangle.

we speak of *the* geometric realization of $\mathcal{K}$, which is denoted $|\mathcal{K}|$. The standard construction sets $|\mathcal{K}|$ equal to the set of functions $\alpha : V(\mathcal{K}) \to [0,1]$ such that $\{v \in V(\mathcal{K}) \mid \alpha(v) > 0\}$ is a simplex of $\mathcal{K}$ and $\|\alpha\|_1 = \sum_{v \in V(\mathcal{K})} \alpha(v) = 1$. The 1-norm induces a metric on $|\mathcal{K}|$ that makes its diameter equal to 1 if $\mathcal{K}$ has more than one vertex. Any simplicial map $h : \mathcal{K} \to \mathcal{L}$ induces a function $|h| : |\mathcal{K}| \to |\mathcal{L}|$. If the complexes are finite, then $|h|$ is necessarily continuous, and there is no guarantee of that otherwise.

A *coloring* of a complex $\mathcal{K}$ is a function $\chi : V(\mathcal{K}) \to \Pi$. The coloring is *chromatic* if any two distinct vertices of the same facet of $\mathcal{K}$ have distinct colors. A *chromatic* complex is a simplicial complex equipped with a chromatic coloring. A *labeling* of a complex $\mathcal{K}$ is a function $\ell : V(\mathcal{K}) \to L$, where L is a set. The set L will be a set of inputs, outputs or process states. Below, we will consider chromatic and labeled complexes such that each vertex is uniquely identified by its color together with its label, namely, for any two distinct vertices u and v, $(\chi(u), \ell(u)) \neq (\chi(v), \ell(v))$. For any vertex v of any such complex, we let denote by $v(p, x)$ the unique vertex of the complex with color $p \in \Pi$ and label $x \in L$.

Let $\mathcal{K}$ be a chromatic complex. The *standard chromatic subdivision* of $\mathcal{K}$, denoted $\operatorname{Chr} \mathcal{K}$, is the chromatic complex whose vertices have the form (p, σ), where $p \in \Pi$, σ is a face of a facet of $\mathcal{K}$ and $p \in \chi(\sigma)$. A set $\{(p_0, \sigma_0), (p_1, \sigma_1), \dots, (p_s, \sigma_s)\}$ is a simplex of $\operatorname{Chr} \mathcal{K}$ if and only if $\sigma_0 \subseteq \sigma_1 \subseteq \dots \subseteq \sigma_s$ and for all $0 \leq q, r \leq s$, if $q \in \chi(\sigma_r)$ then $\sigma_q \subseteq \sigma_r$. The chromatic coloring χ for $\operatorname{Chr} \mathcal{K}$ is defined as $\chi(p, \sigma) = p$. Figure 6 contains the standard chromatic subdivision of an edge, a 1-dimensional simplex, and a triangle, a 2-dimensional simplex. The k-th standard chromatic subdivision, $\operatorname{Chr}^k \mathcal{K}$, is obtained by iterating k times the standard chromatic subdivision. The standard chromatic subdivision is indeed a subdivision: $|\operatorname{Chr}^k \mathcal{K}| \cong |\mathcal{K}|$, for every $k \geq 0$.

A simplicial map $h : V(\mathcal{K}) \to V(\mathcal{L})$ is *chromatic* if it carries colors, *i.e.*, $\chi(v) = \chi(h(v))$, for every vertex v of $\mathcal{K}$. A carrier map $\Phi : \mathcal{K} \to 2^{\mathcal{L}}$ is *chromatic* if $\Phi(\sigma)$ is pure and chromatic of dimension $\dim \sigma$, and each facet of it has colors $\chi(\sigma)$.

▶ **Lemma 2.1.** *Let X be an ultrametric space. For all $x, y \in X$ and all $\delta, \varepsilon > 0$, one of the following is true: (1) $B_\delta(x) \cap B_\varepsilon(y) = \emptyset$, (2) $B_\delta(x) \subseteq B_\varepsilon(y)$, (3) $B_\varepsilon(y) \subseteq B_\delta(x)$.*

Proof. Assume that both (1) and (2) are false. We will prove that then (3) is true.

Let $v \in B_\varepsilon(y)$. We need to show that $v \in B_\delta(x)$. Since (1) is false, there exists a $z \in B_\delta(x) \cap B_\varepsilon(y)$. Applying the ultrametric triangle inequality twice, we have:

$$\begin{aligned} d(v,x) &\leq \max\{d(v,z), d(z,x)\} \leq \max\{d(v,y), d(y,z), d(z,x)\} \\ &< \max\{\varepsilon, \varepsilon, \delta\} = \max\{\varepsilon, \delta\} \end{aligned}$$

It remains to prove that $\varepsilon \leq \delta$ so that $\max\{\varepsilon, \delta\} = \delta$ and $v \in B_\delta(x)$.

Suppose by contradiction that $\varepsilon > \delta$. Since (2) is false, there exists a $u \in B_\delta(x) \setminus B_\varepsilon(y)$. But then we have

$$\begin{aligned} d(u,y) &\leq \max\{d(u,z), d(z,y)\} \leq \max\{d(u,x), d(x,z), d(z,y)\} \\ &< \max\{\delta, \delta, \varepsilon\} = \varepsilon \ , \end{aligned}$$

which means that $u \in B_\varepsilon(y)$, a contradiction to the choice of u. ◀

B Additional Details for Section 4 (Proof of the Generalized Asynchronous Computability Theorem with an Application to Set Agreement)

▶ **Theorem 4.1.** *A sub-IIS model M solves a task $T = (\mathcal{I}, \mathcal{O}, \Delta)$ if and only if there exists a terminating subdivision $\mathscr{T}$ of $\mathcal{I}$ and a chromatic simplicial map $\delta\colon K(\mathscr{T}) \to \mathcal{O}$ such that:*

(a) *$\mathscr{T}$ is admissible for the model M.*

(b) *For any simplex σ of $\mathcal{I}$, if τ is a stable simplex of $\mathscr{T}$ such that $|\tau| \subseteq |\sigma|$, then $\delta(\tau) \in \Delta(\sigma)$.*

(c) *$|\delta|$ is continuous.*

Proof. ($\Rightarrow$): We prove that the geometric realization of the map δ as constructed in the proof of Gafni, Kuznetsov, and Manolescu [11, Theorem 6.1] is continuous by generalizing the proof given by Godard and Perdereau [12, Theorem 33] for the consensus task with two processes.

Let $x \in |K(\mathscr{T})|$ and $\varepsilon > 0$. We show the existence of an $\eta > 0$ such that:

$$\forall y \in |K(\mathscr{T})|: \quad d(x,y) < \eta \implies d\big(|\delta|(x), |\delta|(y)\big) < \varepsilon \tag{1}$$

Let σ be the minimal stable simplex in $K(\mathscr{T})$ such that $x \in |\sigma|$. Since $K(\mathscr{T})$ is locally finite, the star $\operatorname{st}\sigma = \{\tau \in K(\mathscr{T}) \mid \sigma \subseteq \tau\}$ is finite. Let k be the smallest round number such that $\operatorname{st}\sigma \subseteq \Sigma_k$. Denote by D_k the diameter of the geometric realization of simplices in $\operatorname{Chr}^k \mathcal{I}$ and choose $\eta = \varepsilon D_k$.

We show (1) in the geometric realization of every simplex $\tau \in \operatorname{st}\sigma$. By the choice of k, we have $\tau \in \operatorname{Chr}^r \mathcal{I}$ for some $0 \leq r \leq k$. Let $y \in |\tau|$ and denote by α the barycentric coordinates of x with respect to τ and by β the barycentric coordinates of y with respect to τ, *i.e.*, $x = \sum_{v\in\tau} \alpha(v) \cdot v$ and $y = \sum_{v\in\tau} \beta(v) \cdot v$ with $\alpha, \beta \geq 0$ and $\|\alpha\|_1 = \|\beta\|_1 = 1$. Here, we identified each vertex $v \in \tau$ with its position in the geometric realization $|K(\mathscr{T})|$. We then have:

$$d(x,y) = \|x-y\|_1 = \operatorname{diam}|\tau| \cdot \sum_{v\in\tau} |\alpha(v) - \beta(v)| \geq D_k \cdot \sum_{v\in\tau} |\alpha(v) - \beta(v)|$$

By definition of the geometric realization $|\delta|$, we have

$$|\delta|(x) = \sum_{v\in\tau} \alpha(v) \cdot \delta(v)$$

where, again, we identify the vertex $\delta(v)$ with its position in geometric realization $|\mathcal{O}|$. Since $\delta(v)$ is a vertex of $\mathcal{O}$ for every vertex $v \in \tau$, we have

$$d\big(|\delta|(x), |\delta|(y)\big) \leq \sum_{v\in\tau} |\alpha(v) - \beta(v)| \leq \frac{d(x,y)}{D_k} < \frac{\eta}{D_k} = \varepsilon \ ,$$

which shows (1) and concludes the proof of continuity of $|\delta|$.

($\Leftarrow$): We modify the protocol that is constructed in the proof of Gafni, Kuznetsov, and Manolescu [11, Theorem 6.1] for process p to decide in round k if the set

$$B_k(v) = \{w \in V(K(\mathscr{T})) \mid d(v,w) \leq D_k \wedge \chi(w) = p\}$$

only contains vertices that are mapped to the same output vertex by δ, where v is the view of the process in round k. This condition eventually becomes true since the subset topology on the geometric realization of the output vertices $V(\mathcal{O})$ is discrete, and thus δ is locally constant. ◀

▶ **Theorem 4.2.** *Let M be an IIS-sub model such that for any termination subdivision $\mathscr{T}$ of σ that is admissible for M, $|\sigma| = |K(\mathscr{T})|$. Then, $(n-1)$-set agreement is impossible in M.*

Proof. Let M be a sub-IIS model. By Theorem 4.1, if $(n-1)$-set agreement is solvable in model M, there is a (possibly infinite) terminating subdivision $\mathscr{T}$ of σ and a chromatic simplicial map $\delta : K(\mathscr{T}) \to \partial\sigma$ such that (1) $\mathscr{T}$ is admissible for M, (2) for every input simplex $\sigma' \subseteq \sigma$, if τ is a stable simplex of $\mathscr{T}$ such that $|\tau| \subseteq |\sigma'|$, then $\delta(\tau) \in \Delta(\sigma')$, and (3) $|\delta|$ is continuous.

Let us suppose that $|\sigma| = |K(\mathscr{T})|$, namely, $K(\mathscr{T})$ subdivides σ. Thus, for each face $\sigma' \subseteq \sigma$, $|\sigma'| = |K(\sigma')|$, where $K(\sigma')$ denotes the terminating subdivision of σ'. Consider the identity map $g : |\sigma| \to |K(\mathscr{T})|$. Clearly, g is continuous, with $g(|\sigma'|) = |K(\sigma')|$. Consider the function $f = |\delta| \circ g : |\sigma| \to |\partial\sigma|$. Since $|\delta|$ and g are continuous, the function f is continuous too. We argue that $f(|\sigma'|) \subseteq |\sigma'|$, for every proper face $\sigma' \subset \sigma$. Consider any proper face $\sigma' \subset \sigma$. We have that (a) $g(|\sigma'|) = |K(\sigma')|$, by definition of g, (b) for any stable simple $\tau \in \mathscr{T}$ with $|\tau| \subseteq |\sigma'| = |K(\sigma')|$, $\delta(\tau) \in \Delta(\sigma')$, by the properties of δ, and (c) $\Delta(\sigma') = \sigma'$, by definition of Δ. We thus conclude that $f(|\sigma'|) \subseteq |\sigma'|$.

The following lemma is direct consequence of Lemma 4.3.5 in [13], and proves below the impossibility of $(n-1)$-set agreement whenever $|K(\mathscr{T})| = |\sigma|$.

▶ **Lemma B.1.** *There is no continuous map $f : |\sigma| \to |\partial\sigma|$ such that for every proper face $\sigma' \subset \sigma$, $f(|\sigma'|) \subseteq |\sigma'|$.*

One can understand Lemma B.1 as a continuous version of the discrete Sperner's lemma. Intuitively, it states that if a continuous map $f : |\sigma| \to |\partial\sigma|$ maps the boundary of σ to itself (*i.e.*, $f(|\sigma'|) \subseteq |\sigma'|$, for each $\sigma' \subset \sigma$), similar to Sperner's lemma hypothesis, then f cannot exist because the mapping cannot be extended to the interior of σ, since $|\sigma|$ is solid whereas $|\partial\sigma|$ has a hole.

As explained above, Theorem 4.1 and assumption $|K(\mathscr{T})| = |\sigma|$ imply that if $(n-1)$-set agreement is solvable in M, then there exists a continuous map $f : |\sigma| \to |\partial\sigma|$ such that for every proper face $\sigma' \subset \sigma$, $f(|\sigma'|) \subseteq |\sigma'|$. Such continuous map f contradicts Lemma B.1. Therefore, $(n-1)$-set agreement is impossible in M. ◀

C Additional Details for Section 5 (Characterization of Task Solvability in General Models)

▶ **Lemma 5.1.** *The distance function d_p is an ultrametric on* CView_p.

Proof. If $\alpha = \beta$, then $d_p(\alpha,\beta) = 0$. If $d_p(\alpha,\beta) = 0$, then $T_p(\alpha,\beta) = \infty$, which means that there is no index at which they differ by definition, *i.e.*, $\alpha = \beta$. This shows that d_p is positive definite.

The symmetry condition $d_p(\alpha,\beta) = d_p(\beta,\alpha)$ holds since the definition of $T_p(\alpha,\beta)$ is symmetric in α and β.

We now prove the ultrametric triangle inequality by showing:

$$T_p(\alpha,\gamma) \geq \min\big\{T_p(\alpha,\beta), T_p(\beta,\gamma)\big\}$$

Assume by contradiction that $T_p(\alpha,\gamma) < \min\big\{T_p(\alpha,\beta), T_p(\beta,\gamma)\big\}$. Set $t = T_p(\alpha,\gamma)$. Since $t < \infty$ and $t < T_p(\alpha,\beta)$, all local views up to index t coincide in both sequences α and β. Likewise, all local views up to index t coincide in both sequences β and γ. But then, by transitivity of the equality relation on local views, all local views up to index t coincide also in the two sequences α and γ, which means $T_p(\alpha,\gamma) > t = T_p(\alpha,\gamma)$; a contradiction. ◀

▶ **Lemma 5.2.** *Let $\delta_p : \mathsf{CView}_p \to V(\mathcal{O})$ be a function. The following are equivalent:*

1. *There is a protocol such that process p decides the value $\delta_p(\alpha)$ in every execution $E \in \mathsf{Exec}$ with local view $\pi_p(E) = \alpha \in \mathsf{CView}_p$.*
2. *The function δ_p is continuous when equipping CView_p with the topology induced by d_p and $V(\mathcal{O})$ with the discrete topology.*

Proof. ($\Rightarrow$): To show that δ_p is continuous, we will show that the inverse image of any singleton $\{o\} \subseteq V(\mathcal{O})$ is open with respect to d_p. This then implies that the inverse image of any subset $O \subseteq V(\mathcal{O})$, *i.e.*, of any subset of $V(\mathcal{O})$ that is open with respect to the discrete topology, is open with respect to d_p.

Let $\alpha \in \delta_p^{-1}[\{o\}]$. Because process p decides the value o in the local view α, there exists an index T at which this decision has already happened in α. Choose $\varepsilon = 2^{-T}$. Now let $\alpha' \in \mathsf{CView}_p$ with $d_p(\alpha,\alpha') < \varepsilon$. Then, by definition of d_p, the local views of α and of α' are indistinguishable for process p up to and including index T. But then, process p needs to have decided value o at index T in local view α' as well. We thus have $\delta_p(\alpha') = o$, which means that $\alpha' \in \delta_p^{-1}[\{o\}]$. Therefore, the inverse image $\delta_p^{-1}[\{o\}]$ is open with respect to d_p.

($\Leftarrow$): We define the protocol for process p in the following way. Decide value $o \in V(\mathcal{O})$ in the t^{th} step if the set of all local views in CView_p that are indistinguishable from the current execution in the first t steps of process p is included in the inverse image $\delta_p^{-1}[\{o\}]$.

Let $E \in \mathsf{Exec}$ be an execution with $p \in \mathsf{Correct}(E)$. We will show that process p decides value $o = \delta_p(\alpha)$ where $\alpha = \pi_p(E)$. By definition of o, we have $\alpha \in \delta_p^{-1}[\{o\}]$. By continuity of δ_p, the inverse image $\delta_p^{-1}[\{o\}]$ is an open set in CView_p. There hence exists an $\varepsilon > 0$ such that $\alpha' \in \delta_p^{-1}[\{o\}]$ for all $\alpha' \in \mathsf{CView}_p$ with $d_p(\alpha,\alpha') < \varepsilon$. It remains to show that process p eventually decides the value o and that it does not decide any other value in execution E. Setting $T = \lceil \log_2 \varepsilon \rceil$, we see that, by design of the protocol's decision rule, process p has decided value o at the latest in step number T. To show that process p does not decide any other value than o, it suffices to observe that $d(\alpha,\alpha) = 0 < 2^{-t}$ for every $t \geq 0$ and $\alpha \in \delta_p^{-1}[\{o\}]$. ◀

▶ **Lemma 5.3.** *The execution map Ξ is a carrier map such that $\mathcal{P} = \displaystyle\bigcup_{\sigma\in\mathcal{I}} \Xi(\sigma)$.*

Proof. We first prove that Ξ is a carrier map. Let $\sigma \subseteq \tau$. We need to prove that $\Xi(\sigma) \subseteq \Xi(\tau)$. We first show that $\mathsf{Compatible}(\sigma) \subseteq \mathsf{Compatible}(\tau)$. Let $E \in \mathsf{Compatible}(\sigma)$. Then $\mathsf{Part}(E) \subseteq \chi[\sigma] \subseteq \chi[\tau]$ since $\sigma \subseteq \tau$, which means that $E \in \mathsf{Compatible}(\tau)$ because the second condition in the definition is fulfilled since $\pi_p(\sigma) = \pi_p(\tau)$ for all $p \in \chi[\sigma]$. But then, for every $\varphi \in \Xi(\sigma)$, we also have $\varphi \in \Xi(\tau)$ since every E in the definition of $\Xi(\sigma)$ is also valid for $\Xi(\tau)$.

To prove $\mathcal{P} = \bigcup_{\sigma\in\mathcal{I}} \Xi(\sigma)$, it suffices to show $\mathsf{Exec} = \bigcup_{\sigma\in\mathcal{I}} \mathsf{Compatible}(\sigma)$. The inclusion of $\mathsf{Compatible}(\sigma)$ in Exec is immediate by its definition. So let $E \in \mathsf{Exec}$ be any execution. We need to show the existence of a simplex $\sigma \in \mathcal{I}$ such that $E \in \mathsf{Compatible}(\sigma)$. For this, it suffices to choose $\sigma = \{v(p, \mathsf{Init}_p(E)) \mid p \in \mathsf{Part}(E)\}$. This set is an input simplex since the initial values in Exec are chosen according to $\mathcal{I}$ by definition. ◀

Base Fee Manipulation in Ethereum's EIP-1559 Transaction Fee Mechanism

Sarah Azouvi[1]
Unaffiliated, Edinburgh, UK

Guy Goren
Protocol Labs, Haifa, Israel

Lioba Heimbach
ETH Zurich, Switzerland

Alexander Hicks
University College London, UK

Abstract

In 2021 Ethereum adjusted the transaction pricing mechanism by implementing EIP-1559, which introduces the *base fee* – a network fee that is burned and dynamically adjusts to the network demand. The authors of the Ethereum Improvement Proposal (EIP) noted that a miner with more than 50% of the mining power could be incentivized to deviate from the honest mining strategy. Instead, such a miner could propose a series of empty blocks to artificially lower demand and increase her future rewards. In this paper, we generalize this attack and show that under rational player behavior, deviating from the honest strategy can be profitable for a miner with less than 50% of the mining power. We show that even when miners do not collaborate, it is at times rational for smaller miners to join the attack. Finally, we propose a mitigation to address the identified vulnerability.

2012 ACM Subject Classification Theory of computation → Algorithmic game theory and mechanism design; Applied computing → Economics

Keywords and phrases blockchain, Ethereum, transaction fee mechanism, EIP-1559

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.6

Supplementary Material *Software (Source Code)*: `https://github.com/liobaheimbach/Base-Fee-Manipulation-In-Ethereum-s-EIP-1559-Transaction-Fee-Mechanism`
archived at `swh:1:dir:fcf6c5989632f539038fc7c1f9c82f8ef2b17154`

Acknowledgements We thank Andrei Constantinescu for the helpful ideas and discussions.

1 Introduction

Ethereum occupies a central role in the blockchain and decentralized application landscape. Not only does Ethereum's market capitalization currently exceed $225 billion [4], but it is also the leading platform for smart contracts and decentralized finance. Thus, Ethereum has revolutionized the way people envision and interact with blockchain technology.

The proposal of Ethereum Improvement Proposal #1559 (EIP-1559) [3] in April 2019 and its later deployment on Ethereum's mainnet in August 2021, mark a significant milestone for the Ethereum network. EIP-1559 reshaped Ethereum's transaction fee mechanism and remains in place to this day. One of the main goals of EIP-1559 is to simplify the bidding process by reducing the need for complex fee estimation algorithms while ensuring that the mechanism is *incentive compatible* – the best strategy for all players is to follow the protocol as intended. In particular, the mechanism should be both truthful and not incentivize miner

[1] The authors of this work are listed alphabetically.

or user bribes, as articulated by Roughgarden [19]. EIP-1559 has been shown to be incentive compatible [19, 20] when miners are assumed to be *myopic*, i.e., short-sighted in the sense that they are maximizing only their immediate profits. Furthermore, an empirical study [16] concludes that EIP-1559 succeeded in making fees easier to estimate and in reducing delays.

EIP-1559 represents a departure from the previous *first-price auction* system, wherein users submit bids and pay the exact amount of their bid if their transaction was included. Importantly, the entirety of the fees paid by users is awarded to miners. This is no longer the case under EIP-1559 which introduced a base fee, a portion of the transaction fee that is burned and not awarded to miners. The base fee varies according to the fill rate of blocks – a proxy for network demand. Blocks exceeding a predefined target size increase the base fee and blocks below this target size decrease the base fee. Miners are then compensated for creating blocks through a block reward and user tips, i.e., user fees exceeding the base fee.

Considering the substantial financial value being traded on the Ethereum network, it is highly probable that profitable and rational deviations will occur when possible. Thus, it is essential for EIP-1559 to be incentive compatible as the protocol guarantees rely on participants following the intended behavior. Deviations from the intended behavior are considered attacks on the systems. As we will see, the dynamic nature of the base fee opens the door for possible rational attacks by miners. Miners have control over the fill rate of blocks and may thus choose to mine emptier blocks in order to decrease the base fee and increase their future profits, i.e., tips paid on top of the base fee by users. This genre of attack strategies has been acknowledged in Ethereum's EIP-1559 proposal and has been explored under various assumptions in previous research works [19, 11].

This paper presents an analysis of the potential for minority attacks on Ethereum's EIP-1559 transaction fee mechanism under the conservative assumption of a steady demand curve. Our results show that the mechanism is vulnerable to such a 20% minority attack.[2] Additionally, we show that smaller miners may be incentivized to join in on the attack. We provide general insights into when deviating from the prescribed strategy is rational and note that, due to the nature of our model and assumptions, the results are applicable in a wide range of scenarios. We also explain how the attack can be initiated by an Ethereum user (rather than a miner), i.e., show the incentives of users to enact bribes. Finally, to address the identified vulnerability, we propose a mitigation and evaluate its effectiveness through simulations.

2 Basic block reward mechanism in Ethereum

Block proposals. Whether Ethereum's blockchain relies on proof-of-work (PoW) or proof-of-stake (PoS) as sybil resistance, it relies on a leader election to determine the proposer of the next block. To be precise, in a PoW blockchain, miners compete to solve a computational puzzle, and the likelihood of a miner being chosen is based on their share of the network's computational power. In a PoS blockchain, on the other hand, miners stake amounts of the blockchain's native currency to participate and are randomly selected to create a new block with probability proportional to the amount they stake. In both cases, the ideal process is memoryless so each leader election is independent of the previous one. Our model covers both PoW and PoS, hence, the results of this paper apply to both types of blockchains.

When a miner is chosen to propose a block, they select a set of pending transactions to include in the next block and broadcast it to the network. Upon successful inclusion of their

[2] An individual entity with more than 30% staking power currently exists in the Ethereum network [9].

block in the blockchain, the miner receives two rewards: a fixed reward of newly minted currency (Ether in Ethereum's case), and a variable reward from the transaction fees of the included transactions. The block reward is fixed, regardless of the block's content or the miner that proposes it, so our analysis focuses on the potential for miners to increase their revenue via transaction fees. Therefore, we do not consider the block reward.

Transaction fees under EIP-1559. Ethereum transactions involve a set of instructions that are carried out by miners when the transaction is added to the blockchain. To prevent users from overwhelming the network with bogus transactions, users must pay *transaction fees*. These fees should reflect the amount of computational resources needed to execute the instructions, measured in units of *gas* and priced in *Gwei*. Note that 1 Gwei $\triangleq 10^{-9}$ Ether.

The EIP-1559 transaction fee includes a *base fee*, which is paid per unit of gas and varies to balance the supply of gas with the demand for gas. To be exact, the base fee $b[i]$ for block i is determined from the base fee $b[i-1]$ and size $s[i-1]$ of the previous block as follows:

$$b[i] \triangleq b[i-1] \cdot \left(1 + \phi \cdot \frac{s[i-1] - s^*}{s^*}\right). \tag{2.1}$$

Thus, the base fee is determined by comparing the size (measured in consumed gas units) of the previous block to a target block size s^*. If the block is larger than the target size, it indicates high demand for gas, and the base fee is increased to decrease demand. Conversely, if the block size is smaller than the target, it indicates low demand for gas, and the base fee is decreased to increase demand. The sensitivity of the base fee to the size of the previous block is determined by the adjustment parameter ϕ that is currently set to $\frac{1}{8}$ on Ethereum. We note that $b[0]$ was set to 1 Gwei initially and that the maximum valid block size is $2s^*$.

When creating a transaction under the EIP-1559 mechanism, in addition to the gas limit (g), the user must specify the fee cap (c) which is the maximum fee per gas unit they are willing to pay, and a maximum tip per gas unit (ε) which is the priority fee. The transaction will be included in a block only if the fee cap is greater than or equal to the base fee (b). The total fee paid by the user is $\tilde{g} \cdot \min\{b + \varepsilon, c\}$, where $\tilde{g} < g$ is the actual gas consumed by the transaction.[3] A portion of the fee $\tilde{g} \cdot b$ is burnt and the remaining $\tilde{g} \cdot \min\{\varepsilon, c - b\}$ goes to the miner as a tip. The EIP-1559 mechanism aims for users to bid small tips that only cover a miner's costs [3, 19]. Miners are intended to include all transactions that have a fee cap greater or equal to the base fee and prioritize transactions with higher fees only if the maximal block size (s_{max}) is exceeded. For simplicity, we assume that all transactions are of the same size and have a sufficient gas limit (i.e., $g = \tilde{g} = 1$) to eliminate considerations including knapsack and gas estimation and keep the focus on the core matter. E.g., a large transaction is modeled by multiple smaller transactions.

3 Model and Assumptions

In the following, we present our model and outline our assumptions. We highlight that our model follows Roughgarden's [19] very closely with the exception that we do not restrict miners to be myopic. Specifically, while Roughgarden [19] considers only a miner's immediate profits (myopici), we will also consider her future profits.

[3] The gas limit (g) specifies the amount of gas units available for the execution of the transaction. The amount of gas needed to execute a transaction is not always predictable in advance. Moreover, if the needed amount of gas exceeds g, then g gas units are consumed and paid for but the transaction fails.

Users. We assume that users are rational agents who want their submitted transactions to be processed and included in the blockchain. The cost of having one's transaction (tx) included in the blockchain under EIP-1559 depends on the base fee (b), fee cap (c_{tx}), and the maximum tip (ε_{tx}), which we described in Section 2. Each transaction tx has a value v_{tx} that is private to the user who proposes it, which can be thought of as the maximum price that the user is willing to pay for the transaction tx to be included in the blockchain. We, therefore, take the utility of a user proposing tx at each block to be $u(tx) = v_{tx} - \min\{b + \varepsilon_{tx}, c_{tx}\}$ if tx is included and 0 otherwise. The user will then adjust c_{tx} and ε_{tx} according to her bidding strategy, while b is determined by the size of the past blocks according to Equation 2.1.

Miners. Miners produce blocks that include transactions to be executed. In our model, the miner to propose the next block is chosen at random. Drawings constitute independent experiments. In each drawing, a miner X is chosen with probability p_x, where p_x is the miner's share of the total network power. We assume that the power distribution in the network changes slowly, hence, to simplify the analysis we model the system as having a fixed network power distribution, i.e., miners' network power does not change with time.

A miner has dictatorial power over which transactions to include in the block she produces. We assume that miners are rational agents who wish to maximize their profit and therefore choose which transactions to include in their block according to a strategy that maximizes their profit. Aside from their transaction picking strategy, we assume that miners behave "honestly" – that is, as specified by the protocol. We highlight that by not considering other forms of deviating from the protocol, we strengthen our result, showing that even "practically honest" miners would deviate from EIP-1559.

Collaboration. We adopt a standard buyers-sellers perspective. We assume that miners do not collaborate with each other, as they can be viewed as one miner with more power. Similarly, we assume that users do not collaborate with one another since they are competing for the same scarce resource. However, a user (buyer) and a miner (seller) can communicate and adjust their strategies for mutual benefit. In particular, a (sophisticated[4]) user and a miner will collaborate if it benefits both of them. That is, the user pays fewer fees for her transaction while the miner receives more fees. The collaboration is thus rational behavior, leading to improved outcomes for both.

Steady State. The distribution of transaction values is represented by a demand curve, and the standard demand curve is a decreasing function; the higher the fee is, the fewer transactions are willing to pay it. In our work, we make no assumptions about the shape of the demand curve other than that it is a decreasing function. We consider a system in *steady state*, which we define as a system in which the demand curve does not change over time, i.e., it is the same whenever a miner creates a block. This steady state assumption is good for several reasons: (1) it keeps us on par with Roughgarden's work [19], (2) it simplifies the analysis by removing noisy components that can obscure the core principles, and (3) it is a conservative assumption that strengthen our results in comparison to others. For example, the "steady influx" model – where there is a fixed influx of transactions to the network – assumes that miners are able to manipulate the demand curve in their favor, i.e.,

[4] It is expected that sophisticated participants will emerge, actively seeking opportunities to generate excess profits (reduce costs). This expectation is supported by the history of traditional exchange systems, where a multitude of players specialize in high-frequency trading.

artificially increase demand by delaying the inclusion of transactions. Thus, when the attack strategy we will present in Section 4 is beneficial under the *steady state* it will be at least as beneficial under the “steady influx” model, but not vice versa. Hence, our steady demand curve assumption makes our results more general and robust, as it applies to a wider range of adversarial assumptions.

The desired dynamics of EIP-1559 in the steady state are that produced blocks are of target size s^* and that the base fee remains constant at what we refer to as the target base fee b^*, i.e., $s[i] = s^*$ and $b[i] = b^*$ for all i. Further, EIP-1559's desired bidding dynamics are for the user's optimal strategy to be honest in reporting the value of a transaction via c_{tx} and to offer a minimal tip. In other words, in the steady state, EIP-1559 should lead to a Nash-equilibrium with the following strategies: (users) honest value-reporting, and (miners) including all txs with $c_{tx} > b^*$. Then a block produced during steady state would include s^* transactions that are each paying $b^* + \varepsilon$ in fees. Thus, $b^* \cdot s^*$ is burned (red area in Figure 1a), and $\varepsilon \cdot s^*$ is received by the miner (blue area in Figure 1a).

4 A Miner's Deviation from the Honest Strategy

In the following, we consider a miner X who controls a proportion p_x of the network's mining power, i.e., the probability that X proposes the next block is p_x.

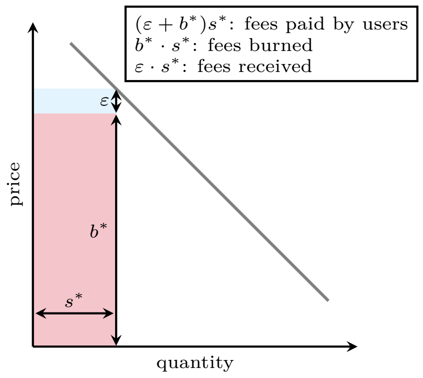
Honest strategy. The honest strategy for X is always to include the maximum possible number of transactions whose gas fee covers at least the base fee. As we consider a system in steady state, the demand curve does not change over time. Further, the honest user bids $c_{tx} = b^* + \varepsilon$ and $\varepsilon_{tx} = \varepsilon$. Thus, miner X will always propose a block that is exactly the target size s^*. The payout received by miner X for every proposed block is, therefore, $s^* \cdot \varepsilon$, where ε is the tip the miner receives. The costs for the users are $(b^* + \varepsilon)s^*$.

Deviation from honest strategy. We continue by outlining a strategy miner X and sophisticated users can employ to manipulate the base fee which results in increasing X's profit and reducing users' costs. When X proposes a block, for which the preceding block was not created by X, she proposes an empty block to reduce the base fee b for subsequent blocks. The miner will receive no payout for this block. Any other consecutive blocks X is chosen to propose, she will propose at target size s^*, profiting from the difference between the targeted and the reduced base fee. To highlight the miner's profit, we assume in the following that users are motivated by slightly reducing their costs. Specifically, users continue to submit transactions with $c_{tx} = b^* + (1-\alpha)\varepsilon$ and $\varepsilon_{tx} = \phi \cdot b^* + (1-\alpha)\varepsilon$, where $0 < \alpha \leq 1$. Thus, the miner will receive at least the difference in base fees, i.e., $\phi \cdot b^*$, and the user will pay $\alpha \cdot \varepsilon$ less for her transaction. By making this assumption, we let the miner extract most of the excess profit. In Section 7 we describe the complementary attack where most of the excess value is gained by the users.

Whenever X proposes a s^* sized block with an artificially reduced base fee, X receives

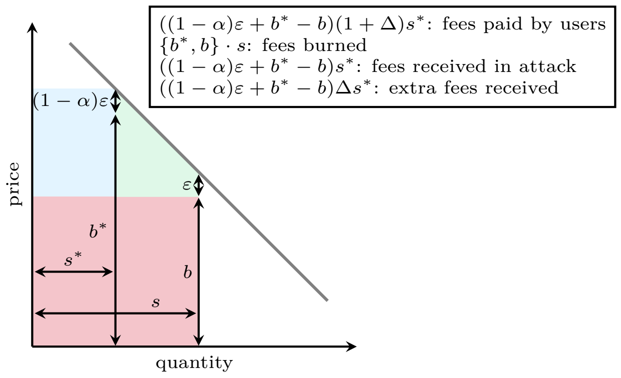
$$s^* \left(\phi \cdot b^* + (1-\alpha)\varepsilon\right), \tag{4.1}$$

where $\phi \cdot b^*$ is the base fee reduction and α represents the proportional reduction of the tip paid by the users. For simplicity, we assume the attack finishes whenever miner X's consecutive turns as proposer finishes. The deviation can only become more profitable if X can continue the attack after an honest proposer, thus, our results apply without the simplification and their generality is not reduced.



$(\varepsilon + b^*)s^*$: fees paid by users
$b^* \cdot s^*$: fees burned
$\varepsilon \cdot s^*$: fees received
price
quantity
ε
b^*
s^*

(a) Sample steady state demand curve, s^* transactions are willing to pay $b^*+\varepsilon$, where b^* is the base fee that corresponds to the target size. The red area is the amount of fees being burned, whereas the blue area are the fees received by the miner.



$((1-\alpha)\varepsilon + b^* - b)(1+\Delta)s^*$: fees paid by users
$\{b^*, b\} \cdot s$: fees burned
$((1-\alpha)\varepsilon + b^* - b)s^*$: fees received in attack
$((1-\alpha)\varepsilon + b^* - b)\Delta s^*$: extra fees received
price
quantity
$(1-\alpha)\varepsilon$
ε
b^*
s^*
b
s

(b) Sample steady state demand curve with lowered base fee b. If the base fee was lowered to $b < b^*$, the demand is increased to $s \geq s^*$. An honest miner will fill up the block with all transactions, paying at least the base fee. The red area indicates the amount of fees being burned, the blue area the fees the attacking miner X receives, and the green area is the additional fees received by the honest miner Y for including all transactions. The scaling factor Δ dictates the size of the green area.

Figure 1 Example demand curves for Ethereum transactions. The quantity (x-axis) indicates the number of transactions willing to pay the transaction fee (price shown y-axis).

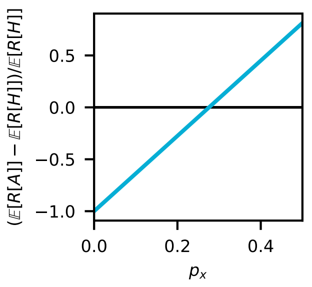
In Theorem 1 we compute the expected reward of X following the honest strategy, as well as the aforementioned described deviation from the honest strategy. By comparing the payout of consecutive turns of X in both strategies, we find that it is rational also for a miner with less than 50% of the power to deviate from the honest strategy.

▶ **Theorem 1.** *In expectation, it is rational for miner X to deviate from the honest strategy, if*

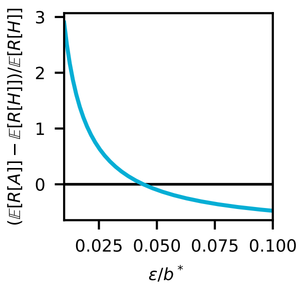
$$p_x > \frac{\varepsilon}{\phi \cdot b^* + (1-\alpha)\varepsilon}.$$

Proof. See Appendix A. ◀

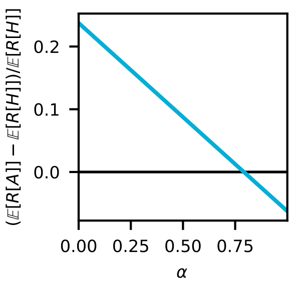
To better illustrate, when it is rational behavior for miner X to deviate from the honest strategy, we plot the relative difference between the expected reward of the attack and the honest strategy in Figure 2. For many realistic parameter configurations, it is rational behavior for miner X to perform the attack and thereby manipulate the base fee. In Figure 2a, we plot the profitability of the attack in comparison to the honest strategy dependent on X's mining power p_x. We set $\phi = 1/8, \varepsilon/b^* = 1/25$, and $\alpha = 0.5$. Notice that even miners with a mining power of less than 0.3 are expected to profit from executing the attack. To underline the expected profitability of the attack, even for small miners, we vary the ratio between the tips and the base fee (ε/b^*) in Figure 2b and set $p_x = 0.3$. Additionally, we vary α, the share of the tips the users keep for themselves, in Figure 2c, and again set $p_x = 0.3$. We conclude that there are multiple realistic parameter configurations for which the outlined attack is profitable, even for a miner with less than 50% of the mining power. Thus, as individual Ethereum pools control in excess of 30% of the staking power [9] such an attack is realistic.



(a) Relative difference shown as a function of p_x. We set $\alpha = 0.5$, $\varepsilon/b^* = 1/25$.



(b) Relative difference shown as a function of ε/b^*. We set $p_x = 0.3$, $\alpha = 0.5$.



(c) Relative difference shown as a function of α. We set $p_x = 0.3$, $\varepsilon/b^* = 25$.

Figure 2 Relative difference between the expected reward of attack and honest strategy as a function of p_x (cf. Figure 2a), ε/b^* (cf. Figure 2b) and α (cf. Figure 2c). We set $\phi = 1/8$, as set in Ethereum. It is rational for X to perform the attack whenever the relative difference is positive. The equations to produce these graphs can be found in the proof of Theorem 1 in Appendix A.

5 The Attack's Effect on Other Miners

In the following, we consider a scenario where a miner X with staking power p_x $(0 < p_x < 1)$ exists for which it is rational behavior to perform the base fee manipulation studied in Section 4. We then analyze the effect of a miner Y with staking power p_y, where $0 < p_y < p_x$, observing that X performs the base fee manipulation attack. More precisely, we study whether it is rational behavior for Y to join the attack partially, i.e., Y will always propose blocks at target size and thereby help keep the base fee artificially low in Section 5.1. Additionally, we will also study when Y would rationally join the attack entirely, i.e., Y also proposes empty blocks when the base fee is not already artificially lowered in Section 5.2. We remark that, throughout, we always assume that miners X and Y do not collaborate.

5.1 Joining the Attack

Consider a miner Y that observes X performing the attack outlined from Section 4. Miner Y is selected as the proposer for a block that follows X's turn as proposer, i.e., the base fee is currently artificially lowered to $(1 - \phi)b^*$. We analyze whether it is rational for Y to follow the honest strategy, i.e., propose the largest block possible and thereby increase the base fee again, or to join the attack and continue keeping the base fee artificially low.

Honest strategy. We first describe the honest strategy. When it is miner Y's turn to propose a block at an artificially lower base fee, miner Y proposes a block with the most transactions possible. Note that the number of transactions is restricted both by the demand at the current base fee (b), as well as the maximum block size, which is twice the target size ($2s^*$). We now consider the demand curve that is drawn in Figure 1b. The demand at price $b^* + \varepsilon$, where b^* is the target base fee price and ε the tip, corresponds to a block of target size s^*. Miner Y will propose a block with the artificially lowered base fee $b = (1 - \phi)b^*$. The demand at this new price $b + \varepsilon$ is represented as s in Figure 1b. Recall, that we make no assumptions about the shape of the demand curve. To make our results stronger, we consider the best case for the honest strategy. Namely, due to the increased demand, Y can propose a block of maximum size (and reap the resulting extra rewards). That is, $s = 2s^*$, after X's turn. Thus, the payout for miner Y proposing a block of size $2s^*$ is given by

$$s^* \left(\phi \cdot b^* + (1-\alpha)\varepsilon\right)(1+\Delta), \tag{5.1}$$

where $s^* \left(\phi \cdot b^* + (1-\alpha)\varepsilon\right)$ is the payout the attacking miner X would receive (cf. Equation 4.1) and $\Delta \in [0,1]$ is a scaling factor that dictates how much additional rewards miner Y received for mining a maximum size block. While $\Delta = 0$ would indicate that Y earns exactly as much as X, i.e., all extra transactions are capped at exactly the base fee, $\Delta = 1$ would indicate that miner Y earns twice the rewards of X, i.e., all extra transactions are capped at the highest possible price of $b^* + \varepsilon$.

After miner Y proposes the block, she will be chosen to propose the next block with a probability of p_y. We continue with the approximation from before, i.e., the effect of a full block after an empty one leads approximately to the same point on the demand curve – $(s^*, b^* + \varepsilon)$. We note that this approximation is accurate up to a term in $O(\phi^2 b^*)$ where $b \in \Theta(\phi b^*)$. Thus, miner Y proposes a block of size s^* and is awarded $s^* \cdot \varepsilon$. From thereon out, she will continue doing so until her consecutive turns as a proposer stops.

With probability p_x, miner X will interrupt Y's turn as proposer. Miner X will start the attack again and propose an empty block to lower the base fee. Note that our approximation of the base fee returning to steady-state levels is the best case for Y's honest strategy and, thus, makes our results stronger. For all consecutive blocks proposed by X, miner X will propose target size blocks to keep the base fee b (i.e., $(1-\phi)b^*$). If miner Y is again selected as a proposer after X's turn, miner Y will proceed with her previously outlined strategy.

At any point, with a probability of $1 - p_y - p_x$, the consecutive turn of the two miners finishes. Note that for Y's honest strategy analysis, we are only interested in these consecutive turns of the two miners as proposers, as we analyze the expected payout of the same sequences for the deviation from the honest strategy which we outline in the following.

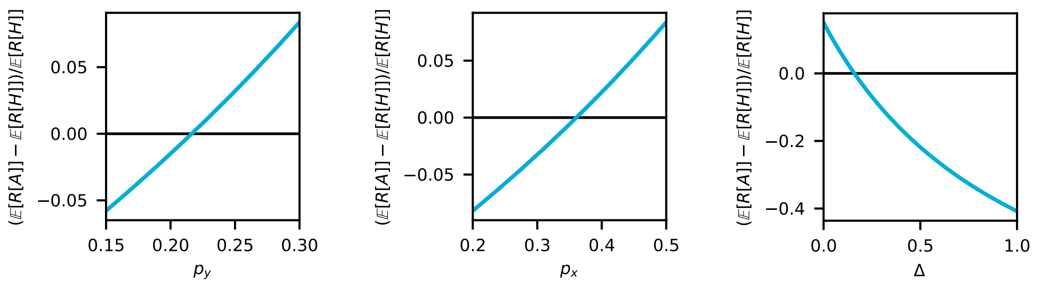
Deviation from honest strategy. We now describe a strategy Y can employ to join the attack she observes. When it is Y's turn to propose a block and the base fee is artificially lowered by X, miner Y will propose a block at the target size. Equivalently to the payout received by miner X for such a block (cf. Section 4), miner Y's reward for proposing the block is given by

$$s^* \left(\phi \cdot b^* + (1-\alpha)\varepsilon\right). \tag{5.2}$$

Following the block proposed by Y, miner Y is again selected to propose a block with probability p_y and miner X with probability p_x. As long as the two miners have an uninterrupted sequence of block proposals, they both keep the base fee artificially low by always proposing target size blocks. Once their consecutive turn as proposers ends, we consider the attack finished. In Theorem 2, we show that it can be profitable for such a miner Y with a mining power p_y to join the attack, i.e., keep the base fee artificially low if she sees a miner X with a mining power $p_x > p_y$ perform the attack. Importantly, this is without assuming collaboration between the two miners.

▶ **Theorem 2.** *In expectation, it is rational for a miner Y to deviate from the honest strategy and join X in keeping the base fee low, if*

$$p_y > \frac{\Delta((1-\alpha)\varepsilon + \phi \cdot b^*)}{(1-\alpha)\Delta \cdot \varepsilon + (1+\Delta)\phi \cdot b^* - \alpha \cdot \varepsilon}.$$



(a) Relative difference shown as a function of p_y. We set $p_x = 0.3$, $\Delta = 0.2$.

(b) Relative difference shown as a function of p_x. We set $p_y = 0.6p_x$, $\Delta = 0.2$.

(c) Relative difference shown as a function of Δ. We set $p_x = 0.3$, $p_y = 0.18$.

Figure 3 Relative difference between the expected reward of attack and honest strategy as a function of p_y (cf. Figure 3a), p_x (cf. Figure 3b) and Δ (cf. Figure 3c). We set $\phi = 1/8$, as implemented in Ethereum, $\varepsilon/b^* = 1/25$, $\alpha = 0.5$ in all plots. It is rational for Y to join the attack whenever the relative difference is positive. The equations to produce these graphs can be found in the proof of Theorem 2 in Appendix A.

Proof. See Appendix A. ◀

To better gauge when it is profitable for miner Y to join the attack, we plot the relative difference between the expected return from the outlined attack and the expected return from the honest strategy in Figure 3. We vary the mining powers of miner Y (cf. Figure 3a) and miner X (cf. Figure 3b), as well as Δ (cf. Figure 3c). In all three plots we set $\phi = 1/8, \varepsilon/b^* = 1/25$, and $\alpha = 0.5$. In Figure 3a, we set $\Delta = 0.2$ and notice that even a miner Y with a mining power slightly larger than 0.2 would be inclined to join the attacking miner X with a mining power of 0.3 in manipulating the base fee. We observe in Figure 3b that a miner Y that is only six-tenths of the size of miner X would also join the attack even if miner X only controls less than 40% of the mining power. Finally, in Figure 3c, we show the dependency of the attack's profitability for miner Y as a function of Δ, i.e., the additional payout received by miner Y following the honest strategy when proposing a full block after the attack by X. Notice that while it is rational for a miner Y to join the attack for small Δ's, this is not the case for larger Δ's. We remark that this is due to the rewards from the full block mined if Y follows the honest strategy being very significant for a large Δ. Nevertheless, our results show that for realistic parameter configurations, it is rational for a miner Y to join the attack she sees a larger miner X perform – even without assuming collaboration between the two.

5.2 Join and Initiate the Attack

In addition to only joining the attack, it is also possible for miner Y, observing X continuously performing the base fee manipulation, to also propose an empty block whenever she proposes a block with the target base fee (b^*), knowing that X will aid her in keeping the base fee low subsequently. We analyze, in the following, when it is more profitable for miner Y to join X's attack in her entirety, as opposed to remaining honest.

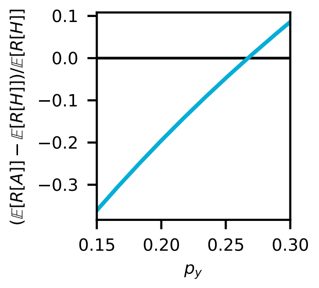
Honest strategy. The honest strategy for miner Y is identical to that described in Section 5.1. However, now it is also important to mention that anytime miner Y proposes a block with base fee b^*, i.e., whenever Y proposes a block that does not follow X's attack, Y will propose a target size (s^*) block. For proposing such a block, miner Y will receive $s^* \cdot \varepsilon$.

Deviation from honest strategy. In the deviation from the honest strategy, Y will propose an empty block whenever she mines a block where the base fee corresponds to the target base fee b^*. Then with probability p_y, miner Y will also propose the next block and will profit from the difference between the base fee and the target base fee by mining a target size block. On the other hand, with probability p_x, miner X will propose the next block and will also mine a target size block – keeping the base fee low. With Theorem 3, we show that it can even be profitable for a miner Y to commence the attack if she knows that a larger miner X will help her in keeping the base fee artificially low.

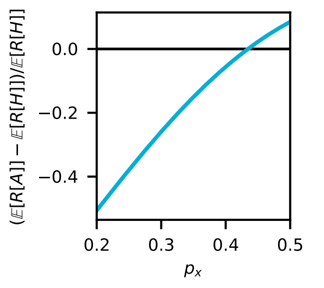
▶ **Theorem 3.** *In expectation, it is rational for a miner Y to deviate from the honest strategy and lower the base fee, if*

$$p_y > \frac{\varepsilon(1-p_x)}{(1-\alpha)\varepsilon + \phi \cdot b^*(1-p_x) + (1+\Delta)\varepsilon\alpha p_x - \Delta p_x(\varepsilon + \phi \cdot b^*)}.$$

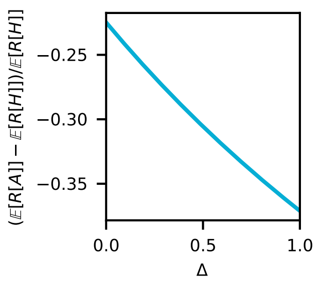
Proof. See Appendix A. ◀



(a) Relative difference shown as a function of p_y. We set $p_x = 0.3$, $\Delta = 0.2$.



(b) Relative difference shown as a function of p_x. We set $p_y = 0.6p_x$, $\Delta = 0.2$.



(c) Relative difference shown as a function of Δ. We set $p_x = 0.3$, , $p_y = 0.18$.

Figure 4 Relative difference between the expected reward of attack and honest strategy as a function of p_y (cf. Figure 4a), p_x (cf. Figure 4b) and Δ (cf. Figure 4c). We set $\phi = 1/8$, as implemented in Ethereum, $\varepsilon/b^* = 1/25$, $\alpha = 0.5$ in all plots. It is rational for Y to initiate new attacks in addition to X whenever the relative difference is positive. The equations to produce these graphs can be found in the proof of Theorem 3 in Appendix A.

To better understand the parameter configuration under which it would be rational for Y to also initiate the attack, we plot the relative difference of the reward of the attack in comparison to the reward of the honest strategy in Figure 4. Note that we again set $\phi = 1/8, \varepsilon/b^* = 1/25$, and $\alpha = 0.5$. Figure 4a, where we also set $p_x = 0.3$ and $\Delta = 0.2$, shows that it can be profitable for a miner Y to initiate the attack, i.e., mine an empty block to lower the base fee, knowing that miner X will support her in keeping the base fee artificially low. Notice though that the threshold where it is rational for Y to also start the attack is reached later in comparison to the threshold where it is rational for miner Y to only join the attack (cf. Figure 3a). A similar picture paints itself when we look at the profitability of initiating the attack for Y as a function of X's mining power in Figure 4a. Again we see that for a miner Y, it can be profitable to start the attack only with the knowledge that X will aid her in keeping the base fee low. Finally, in Figure 4c, we show the relative difference between the expected profit of the outlined attack for Y and the honest strategy. For the chosen parameter configuration, $p_x = 0.3$ and $p_y = 0.18$, the attack would actually never be profitable regardless of Δ. We, thus, summarize that while it is only rational for Y to initiate the attack for a reduced set of parameters, it is remarkable that this is even the case. By starting the attack, Y is taking a loss for a larger miner X just based on the knowledge that she will be supported by X in keeping the base fee low. Astonishingly, the collaboration of the two miners is not required.

6 Possible Mitigations

To mitigate the attacks described in Sections 4 and 5, we focus on addressing the deviation of player X, as if X does not deviate from the honest strategy, and neither does Y. We start by examining the trivial mitigation of reducing ϕ. Theorem 1 shows that decreasing ϕ by a factor of β will require that p_x is approximately β times larger for a deviation to be profitable. For example, if Ethereum were to decrease its current ϕ of 1/8 to 1/16, it would require p_x to be approximately twice as large for a deviation to be profitable. This approach can be effective during stable periods, but it might not be able to adjust quickly enough to changes in demand. Further, Leonardos et al. [14] show how the value of ϕ determines the trade-off between a base fee that adjusts too quickly (which can lead to chaotic behavior) and one that adjusts too slowly to fulfill its purpose.

The following question appears in EIP1559 FAQ [2]: *“Won’t miners have the incentive to collude to push down the base fee by making all their blocks less than half full?”* In response to the question Buterin proposes this mitigation: *“Divert half of the collected base fees, that would otherwise be burned, into a special pool. Whenever a miner mines a new block add to her block reward a* 1/8192 *portion of the amount in that pool. This will incentivize miners to maintain a higher base fee.”* One might falsely presume that this solution, to the above-posed question, might also be used to solve the deviation we outline in Section 4. However, the two attacks are inherently different as the one we outline does not require miners to collude. Thus, this proposal does not solve our deviation. The added cost of the attack, i.e., the lost revenue from half of the base fee reduction, is distributed among many while the attacker’s expected revenue remains unchanged. In the long run, this proposal only minimally reduces the attacker’s expected revenue by $b^* \cdot 2^{-13}$, which is typically significantly less than the expected rewards and, therefore, ineffective.

Another straw man to consider is to use the average of the previous W block sizes instead of just the size of the last block to measure demand in the base fee update rule. This method may appear to be promising because it reduces the effect of an empty block on the following block by a factor of W, and its effect on the rate of adjustment (embodied in ϕ) is easily accounted for, adding an adjustment delay within $O(W)$. However, this mitigation fails to mitigate the attack, as it increases the opportunity for X to profit from later blocks that are within a W distance from the empty block, thereby increasing the expected profitability of the deviation. For example, if we use a two-block window (i.e., $W = 2$) and $\phi = 1/8$, the base fee is reduced by a smaller factor of only $\phi/W = -1/16$ to $b_1 = 15b^*/16$ as desired. Nevertheless, even if the ensuing block is completely full (i.e., $s_1 = 2s^*$), the base fee for the block following it does not increase and remains $b_2 = b_1 = 15b^*/16$ which means an additional profit opportunity for the deviation that compensates X for the reduced factor. As a result, the deviation is not mitigated and is actually exacerbated.

We propose the following mitigation, to use a geometric sequence as weights to average the history of block sizes. Formally, for $q \in (0, 1)$ denote

$$s_{avg}[i] \triangleq \frac{1-q}{q} \sum_{k=1}^{\infty} q^k \cdot s[i-k+1] = (1-q) \cdot s[i] + q \cdot s_{avg}[i-1], \tag{6.1}$$

and replace $s[i]$ in Equation 2.1 by $s_{avg}[i]$ to get the following base-fee update rule

$$b[i+1] = b[i] \cdot \left(1 + \phi \cdot \frac{s_{avg}[i] - s^*}{s^*}\right). \tag{6.2}$$

We note that $b[0]$ would be initialized to 1 Gwei initially and that s_{avg} would be initialized to be the size of the first block after the transition. By applying the update rule in Equation 6.2, we reduce the effect of the empty block on the ensuing block by a factor of $(1 - q)$ and discount its effects on later blocks exponentially fast (with base q). For example, using the same parameters as before with $\phi = 1/8$, if we set $q = 1/2$, after X's empty block (at slot τ), the base fee will be reduced by a factor of $(1 - \phi(1 - q)) = (1 - 1/16)$ to $b[\tau + 1] = 15b^*/16$. Now, however, making the same assumption as before (i.e., $s[\tau + 1] = 2s^*$), the base fee for the next block, $b[\tau+2]$, will be $495b^*/512$. This decreases the potential profit margin of block $\tau + 2$ from $b^*/16$ to $b^*/32 + b^*/512$, which is almost a factor of 2 reduction. Therefore, the added profit opportunity in future blocks is not enough to compensate for the lost potential in the immediately ensuing block. As a consequence, the attack is considerably mitigated.

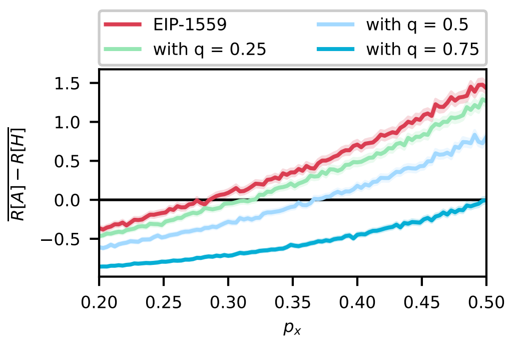
The above mitigation method has two additional properties: (i) its computation and space complexity are both in $O(1)$, and (ii) it gradually phases out the impact of a single empty block without causing significant fluctuations. To reason about the effect our proposal has on response times we use the following methodology. Suppose that the demand suddenly changes and the new (desired) steady state should be reached at a new point with a base fee that is β times higher than the current base fee (b^*). Denote by T the number of consecutive full blocks required to reach the new base fee. We use T as a function of β to characterize the response time of a fee-setting mechanism to sudden changes in demand.

EIP-1559 as it is currently implemented will take $T(\beta) = \lceil \log_{(1+\phi)} \beta \rceil$ blocks to reach the new base fee βb^*. With our mitigation, it takes slightly longer. To be precise, in Appendix B we show the exact delay $T(\beta)$ of our mitigation proposal is the smallest integer T that satisfies $\prod_{k=1}^{T}(1 + \phi(1 - q^k)) \geq \beta$. We further plot the $T(\beta)$ for EIP-1559 and our mitigation in Appendix B (cf. Figure 10).

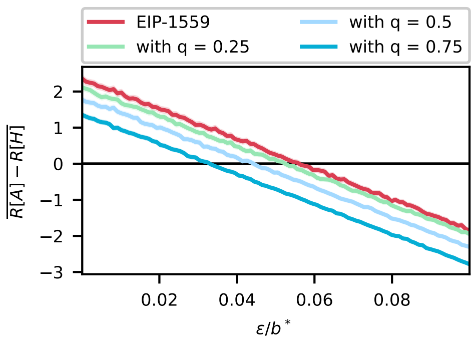
Simulations

In order to gauge the effectiveness of the proposed mitigation, we conducted simulations that compare the excess profit of an attacker (profit gained through deviation minus profit gained through honest mining) under EIP-1559 with and without the mitigation. To account for the probabilistic nature of the attack (profits in expectation only), we calculated the average of the results for each data point over 10,000 runs, each using a different random seed. Each simulation begins with X mining an empty block, the next blocks are mined by X with probability p_x per block and by an honest miner with probability $1 - p_x$. We assume that X will then always mine target-size blocks, while the honest miners will mine blocks at twice the target size. We keep track of the payout received by X and declare the attack finished if the base fee has recovered to 99% of its target size. We perform 10,000 runs for both the base fee evolution according to EIP-1559, as well as our mitigation. Simultaneously, we keep track of the payout X would have received for the same random seed if she had followed the honest strategy. The results of the simulation are depicted in Figures 5 and 6.

Figures 5a and 5b show X's profit from attacking as a function of p_x and the ratio ε/b^* for EIP-1559 with and without the mitigation for $q \in \{1/4, 1/2, 3/4\}$. The results decisively demonstrate the benefit of our mitigation; it becomes much harder for X to profit from attacking. Finally, Figure 6 illustrates the effect of the mitigation (with $q \in \{1/4, 1/2, 3/4\}$)



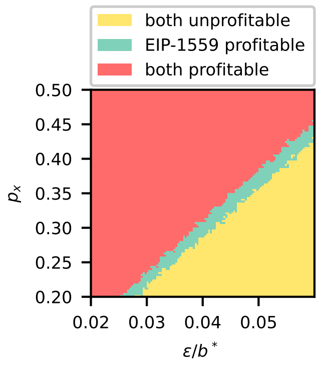
(a) Difference of mean attack and mean honest return as a function of p_x for EIP-1559 and the proposed mitigation for $q \in \{1/4, 1/2, 3/4\}$. We set $\varepsilon/b^* = 1/25$.



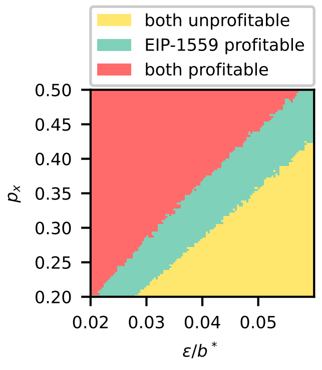
(b) Difference of mean attack and mean honest return as a function of ε/b^* for EIP-1559 and the proposed mitigation for $q \in \{1/4, 1/2, 3/4\}$. We set $p_x = 0.4$.

Figure 5 Profitability of the attack under EIP-1559 and our proposed mitigation. We plot the mean attack profitability (cf. Figures 5a and 5b) along with the 95% confidence interval. We set $\phi = 1/8$, $\alpha = 0.5$, $s^* = 1$.

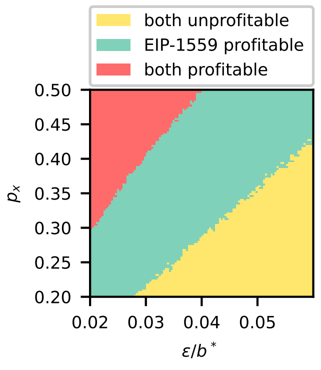
on the configurations at which X will attack. Most notably, the green area represents a set of configurations in which X had attacked without the mitigation and will be attacking no longer. The value of the proposed approach is evident from the results.



(a) $q = 1/4$.



(b) $q = 1/2$.



(c) $q = 3/4$.

Figure 6 Attack profitability under EIP-1559 and the proposed mitigation for $q \in \{1/4, 1/2, 3/4\}$ as a function of p_x and ε/b^*. Importantly, the green area shows where the mitigation can prevent the attack but EIP-1559 cannot. We set $\phi = 1/8$, $\alpha = 0.5$, $s^* = 1$.

7 User perspective

Until now, we have approached the topic from the miners' point of view. Considering the users as first-class citizens, and observing the attack through their eyes, contributes a new perspective on the results.

Instead of the miners taking it upon themselves to initiate the attack, we can imagine users who wish to pay lower costs coordinating the attack. Let u be a user (or group of users) that has transactions with a g amount of gas. Assume the other users naively follow the strategy of the desired equilibrium. That is, they bid an honest valuation with an ε tip. The attacker's strategy is as follows: u bribes the miner of the current block (no matter the miner's power) to propose an empty one instead. Any bribe larger than $s^*\varepsilon$ will suffice. Consequently, the base fee reduces in the next block. If the other users naively continue to bid with an ε tip (or they are simply slow to react), u can guarantee the inclusion of her transactions with any tip larger than ε – making the attack profitable whenever $g\phi b^* > s^*\varepsilon$.

8 Discussion

Myopic vs. non-myopic. Roughgarden's analysis [19] of the EIP-1559 protocol suggests that, in the steady state, the honest strategies of the users and miners are incentive compatible. Our opposing conclusion stems from a single different assumption, namely, [19] considers miners that only care for immediate profits (referred to as myopic), while we consider non-myopic miners that do not disregard future profits. There are valid reasons to model miners as myopic. For example, the proposing turns of a very small miner are sporadic, and accounting for rare future profits is negligible in comparison to the prize at hand. However, there are strong reasons for modeling miners as non-myopic. Measurement studies have shown that the distribution of mining power follows a power law rather than a uniform distribution [8, 9], i.e., the biggest miners control significant portions of the mining power. Moreover, the "shorter vision" of smaller miners is accounted for under our model. Our quantitative results show that as a miner's size decreases, the lesser the profitability of the deviations. Finally, from a conceptual perspective, miners are typically players of high stakes (the minimum stake in Ethereum is 32 ETH which is currently over $50,000), and participation also requires some expertise, planning, and locking of assets. Therefore, it does not seem appropriate to consider these players as ones that neglect future considerations.

Additional observations. The deviations described in this paper have an interesting property; they appear to have a win-win-win outcome. The attacker profits, the non-attacking miners profit, and the users profit by paying a cheaper total gas fee. But not all is rosy; there is a hidden cost involved. In order for users to benefit, they must diligently follow the miners' actions and compute the appropriate response. This increase in complexity for the users is in opposition to one of the main goals of EIP-1559 – simplifying the bidding mechanism and eliminating the need for complex fee estimation algorithms. As a result, sophisticated users take precedence and push naïve users to the back of the line.

Although it is desirable that the leader election process be unpredictable, in practice, this is not the case. Currently, in Ethereum, implementation considerations led to miners knowing their own proposing slots 32–62 blocks in advance [6]. This predictability clearly favors the attacker, who no longer needs to lose tips for the probability of winning more. Instead, the miner only attacks when it is guaranteed to mine at least two blocks in a row. The predictability issue has risen with the move to PoS and was not present in PoW Ethereum.[5]

[5] Since the randomness in PoW does not depend on a peer-to-peer communication source, it does not have a predictability concern. The predictability issue is a result of implementation constraints for the cryptographic protocols of Verifiable Random Functions (VRFs).

9 Related work

Blockchain transaction fee mechanism. Huberman et al. [12] provide an early analysis of Bitcoin's first-price auction. An in-depth exploration of miner manipulation in transaction fee mechanisms was first explored by Lavi et al. [13]. While these works study the first-price auction used in Bitcoin and originally in Ethereum as well, our work studies the incentives for miners to deviate from the EIP-1559 protocol currently used in Ethereum.

Stability of the base fee. As Leonardos et al. [14] show in their theoretical analysis, a key parameter in the base fee mechanism is the adjustment parameter ϕ. In particular, they show that stability is not guaranteed depending on system conditions (e.g., a congested network), especially if ϕ is set to too high a value. However, there are also conditions under which the base fee may have bounded oscillations or even converge, providing stability. In another work, Leonardos et al. [15] show that despite the short-term chaotic behavior on the base fee, the long-term average block size is close to the target size.

Reijsbergen et al. [18] empirically show that a stable base fee may not tell the whole story, however, as even in cases where the base fee remains relatively stable (e.g., between 25 and 35 Gwei) and the block size is on average the target block size, block sizes can fluctuate wildly (as explained by Leonardos et al. [14]), impacting miner revenue. One reason for this is that the currently used value for the adjustment parameter ($\phi = \frac{1}{8}$) is too low during periods where the demand rises sharply (i.e., the base fee does not increase quickly enough) but also too high when demand is stable (inducing fluctuations in block size). Therefore, [18] suggests making ϕ variable based on the demand. Their work does not consider bribes and is complementary to ours. Their suggested mitigation to the stability issue is to have ϕ adaptive according to an Additive Increase Multiplicative Decrease (AIMD). Since we do not vary ϕ (but instead update the base fee update rule), an interesting experiment would be to combine our mitigation technique (averaging $s[i]$ geometrically) with their AIMD ϕ setting and examine the results on data from the real world.

Ferreira et al. [7] show that although the first-price auction utilized under EIP-1559 is incentive-aligned for miners, it provides a bad user experience. In particular, they observe bounded oscillation of the base fee in experiments when bidders all associate the same value with their transaction. While their work studies the stability of the base fee with myopic miners, we study the manipulability of the base fee in the presence of non-myopic miners.

Manipulability of the base fee. Manipulation of the base fee has been a concern since EIP-1559 was proposed [1] as it is straightforward to notice that the base fee could be manipulated downwards by a miner that intentionally mines empty blocks.

The EIP as listed on Ethereum's Github repository acknowledges the possibility of miners mining empty blocks but determines that such a deviation from an honest mining strategy would not lead to a stable equilibrium as other miners would benefit from this (i.e., benefiting from the reduced base fee without the opportunity cost of mining an empty block) [3]. Their belief was therefore that executing such a strategy would require a miner to control more than half the hashing power (the document precedes Ethereum's switch to PoS).

In his exposition of EIP-1559 [19], Roughgarden considered this in the case of a 100% miner (or any miner with greater than 50% of the mining power) that would drive the base fee down to 0 by mining empty blocks then, in order to maximize their revenue, switching to either mining target size blocks in perpetuity, maintaining the base fee at 0 and essentially reverting back to a first price auction, or mining sequences of under full and overfull blocks (relative to the target size) according to the demand. Roughgarden restricts himself to this case and does not otherwise consider non-myopic miners, assuming that with a high enough level of decentralization, the probability of any miner being elected to propose a block two times in a row was too low for any miner to consider strategies over multiple blocks.

Similar to us, a concurrent work by Hougaard and Pourpouneh [11] also relaxes the myopic assumption and proves that non-myopic miners would be incentivized to deviate even if they are a minority. However, their analysis relies on several assumptions, that while legitimate, provide the attacker with advantageous conditions in comparison to the more conservative assumptions made in this paper. In particular, [11] relies on miners knowing the exact parameters of the demand distribution of the users (which itself is restricted to each user drawing from a uniform distribution), as well as on miners being able to manipulate the demand curve in favor of the future blocks; inducing artificial future congestion by not including transactions in the current block and letting them accumulate. We do not assume any specific demand curve, only a steady demand curve, and do not rely on miners inducing congestion, which makes our attack more profitable. Although we do assume that users act rationally and will adapt their strategy if benefits them, while [11] assumes users are passive and do not adapt their strategies in a rational manner.

MEV. Miner/Maximal Extractable Value (MEV) has gained significant attention in the blockchain research community in recent years [5, 21, 17]. Similarly to this work, MEV strategies enable a miner to accrue excess profits in comparison to naïve mining. However, in MEV the value comes from analyzing the actual data in the transactions, whereas, in this work, the miner's excess profit comes from manipulating the EIP-1559 mechanism. Therefore, many of the suggested mitigations against MEV [10] (such as obscuring transaction data until inclusion) will not work against our attack.

10 Conclusion and Future Work

In this paper, we demonstrated that even under very conservative assumptions (steady state, miners cannot induce congestion, unknown demand functions), there are strong incentives for minority miners to deviate from the EIP-1559 protocol. Furthermore, we showed that once an attack begins, previously honest miners' rational response may be to join the deviation and even sometimes initiate new attacks – worsening the problem rather than improving it.

To mitigate the problem, we suggested using a weighted average with the weights being a geometric series. This direction seems promising as it has several desirable properties and trade-offs (e.g., balancing attack mitigation with low additional response delays). However, further research rigorously analyzing it in a broader context is warranted.

References

1 Tim Beiko. EIP-1559 Community Outreach Report. `https://medium.com/ethereum-cat-herders/eip-1559-community-outreach-report-aa18be0666b5`, 2020. Accessed: 2023-01-25.

2 Vitalik Buterin. EIP 1559 FAQ. `https://notes.ethereum.org/@vbuterin/eip-1559-faq#Won%E2%80%99t-miners-have-the-incentive-to-collude-to-push-down-the-BASEFEE-by-making-all-their-blocks-less-than-half-full`, 2021. Accessed: 2023-01-26.

3 Vitalik Buterin, Eric Conner, Rick Dudley, Matthew Slipper, Ian Norden, and Abdelhamid Bakhta. Fee market change for ETH 1.0 chain. `https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1559.md`, 2019. Accessed: 2020-09-28.

4 CoinMarketCap. Today's Cryptocurrency Prices by Market Cap. `https://coinmarketcap.com/`, 2017. Accessed: 2023-07-21.

5 Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 910–927. IEEE, 2020.

6 Ethereum. Phase 0 – Honest Validator. `https://github.com/ethereum/consensus-specs/blob/dev/specs/phase0/validator.md#lookahead`, 2022. Accessed: 2023-01-27.

7 Matheus VX Ferreira, Daniel J Moroz, David C Parkes, and Mitchell Stern. Dynamic posted-price mechanisms for the blockchain transaction-fee market. In *Proceedings of the 3rd ACM conference on Advances in Financial Technologies*, pages 86–99, 2021.

8 Adem Efe Gencer, Soumya Basu, Ittay Eyal, Robbert Van Renesse, and Emin Gün Sirer. Decentralization in bitcoin and ethereum networks. In *Financial Cryptography and Data Security: 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26–March 2, 2018, Revised Selected Papers 22*, pages 439–457. Springer, 2018.

9 Dominic Grandjean, Lioba Heimbach, and Roger Wattenhofer. Ethereum proof-of-stake consensus layer: Participation and decentralization. *arXiv preprint arXiv:2306.10777*, 2023.

10 Lioba Heimbach and Roger Wattenhofer. Sok: Preventing transaction reordering manipulations in decentralized finance. In *4th ACM Conference on Advances in Financial Technologies (AFT), Cambridge, Massachusetts, USA*, September 2022.

11 Jens Leth Hougaard and Mohsen Pourpouneh. Farsighted miners under transaction fee mechanism eip1559. Technical report, IFRO Working Paper, 2022.

12 Gur Huberman, Jacob D Leshno, and Ciamac Moallemi. Monopoly without a monopolist: An economic analysis of the bitcoin payment system. *The Review of Economic Studies*, 88(6):3011–3040, 2021.

13 Ron Lavi, Or Sattath, and Aviv Zohar. Redesigning bitcoin's fee market. *ACM Transactions on Economics and Computation*, 10(1):1–31, 2022.

14 Stefanos Leonardos, Barnabé Monnot, Daniël Reijsbergen, Efstratios Skoulakis, and Georgios Piliouras. Dynamical analysis of the eip-1559 ethereum fee market. In *Proceedings of the 3rd ACM Conference on Advances in Financial Technologies*, AFT '21, pages 114–126, New York, NY, USA, 2021. Association for Computing Machinery. `doi:10.1145/3479722.3480993`.

15 Stefanos Leonardos, Daniël Reijsbergen, Daniël Reijsbergen, Barnabé Monnot, and Georgios Piliouras. Optimality despite chaos in fee markets. *arXiv preprint arXiv:2212.07175*, 2022.

16 Yulin Liu, Yuxuan Lu, Kartik Nayak, Fan Zhang, Luyao Zhang, and Yinhong Zhao. Empirical analysis of eip-1559: Transaction fees, waiting times, and consensus security. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, pages 2099–2113, New York, NY, USA, 2022. Association for Computing Machinery. `doi:10.1145/3548606.3559341`.

17 Kaihua Qin, Liyi Zhou, and Arthur Gervais. Quantifying blockchain extractable value: How dark is the forest? In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 198–214. IEEE, 2022.

18 Daniël Reijsbergen, Shyam Sridhar, Barnabé Monnot, Stefanos Leonardos, Stratis Skoulakis, and Georgios Piliouras. Transaction fees on a honeymoon: Ethereum's eip-1559 one month later. In *2021 IEEE International Conference on Blockchain (Blockchain)*, pages 196–204. IEEE, 2021.

19 Tim Roughgarden. Transaction fee mechanism design for the Ethereum blockchain: An economic analysis of EIP-1559. *arXiv preprint arXiv:2012.00854*, 2020.

20 Tim Roughgarden. Transaction fee mechanism design. *ACM SIGecom Exchanges*, 19(1):52–55, 2021.

21 Christof Ferreira Torres, Ramiro Camino, et al. Frontrunner jones and the raiders of the dark forest: An empirical study of frontrunning on the ethereum blockchain. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1343–1359, 2021.

A Omitted Proofs

▶ **Theorem 1.** *In expectation, it is rational for miner X to deviate from the honest strategy, if*

$$p_x > \frac{\varepsilon}{\phi \cdot b^* + (1-\alpha)\varepsilon}.$$

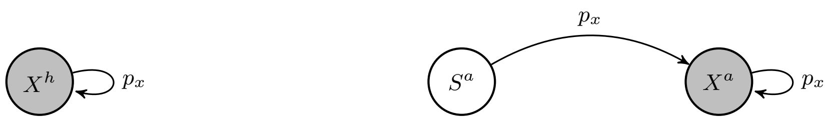
Proof. We commence with the honest strategy and calculate the expected payout for consecutive turns of X as proposer by modeling the process as a Markov chain (cf. Figure 7a).

When it is miner X's first time as proposer in a consecutive turn, i.e., the previous block was not mined by X, we are in state X^h. In this state, miner X proposes a block at target size s^* and receives tips at price ε. Thus, the payout for miner X in state X^h is $P[X^h] = s^* \cdot \varepsilon$. With probability p_x, X stays in state X^h for the next block and also proposes the next block. Else, with probability $1 - p_x$, we enter an absorbing state. We calculate the expected number of times X proposes consecutive blocks and, thereby, the expected payout for X.

By linearity of expectation, it follows that the expected payout for a sequence of consecutive turns by X as a proposer following the honest strategy is given by

$$\mathbb{E}[R[X^h]] = p_x \cdot \mathbb{E}[R[X^h]] + P[X^h] \iff \mathbb{E}[R[X^h]] = \frac{s^* \cdot \varepsilon}{1 - p_x}, \tag{A.1}$$

as miner X is awarded $P[X^h]$ every time she proposes a block. Note that the expected reward of the honest strategy i.e., $\mathbb{E}[R[H]]$ corresponds to the expected payout of the Markov process starting in state X^h, i.e., $\mathbb{E}[R[H]] = \mathbb{E}[R[X^h]]$.



(a) honest strategy of miner X. **(b)** attacking strategy of miner X.

Figure 7 The honest strategy (cf. Figure 7a) and deviation from the honest strategy (cf. Figure 7b) modeled with discrete Markov chains. All states with a nonzero payout for miner X are highlighted in gray. We transition between states with every block. Note that for all remaining probabilities, the Markov process enters an absorbing state, and X's consecutive turns as a proposer finish.

We now examine the reward for miner X if she chooses to carry out the attack, modeling the deviation from the honest strategy as a Markov chain (as shown in Figure 7b). The starting point for the attack, denoted as state S^a, is when miner X submits an empty block. Therefore, the payout in state S^a is 0. But with a probability of p_x, miner X is chosen as the proposer for the next block and enters state X^a, where she submits a block at the target size. The payout in state X^a can be calculated by $P[X^a] = s^* (\phi \cdot b^* + (1-\alpha)\varepsilon)$.

If X is not selected to propose another block in a row, we enter an absorbing state and the attack stops. We make the approximation that whenever X's consecutive turns as proposer finish, honest miners will bring the base fee back to b^* before X gets to mine another block. It is possible that the honest miners do not bring the base fee back up to the target before X is selected to propose again, and X could continue the attack at a lower cost. However, assuming that the base fee had fully recovered simplifies the analysis and only makes our results stronger, as it reduces X's attack rewards. From state X^a we remain in state X^a for the next block with probability p_x, i.e., X is selected to propose another block or enter the absorbing state with probability $1 - p_x$.

The expected payout for miner X in state S^a is given by $\mathbb{E}[R[S^a]] = p_x \cdot \mathbb{E}[R[X^a]] + P[S^a]$, where $\mathbb{E}[R[X^a]]$ is the expected payout starting from state X^a and we have $\mathbb{E}[R[X^a]] = p_x \cdot \mathbb{E}[R[X^a]] + P[X^a]$. It follows that the expected payout of the attack is

$$\mathbb{E}[R[A]] = \mathbb{E}[S^a] = \frac{p_x \cdot s^* \left(\phi \cdot b^* + (1-\alpha)\varepsilon\right)}{(1-p_x)}. \tag{A.2}$$

Note that the payout of the attack is given by the expected payout starting from state S^a, as the attack commences in said state.

We conclude that it is rational for miner X to deviate from the honest strategy if $\mathbb{E}[R[A]] > \mathbb{E}[R[H]]$. It follows that X attacks when

$$p_x > \tfrac{\varepsilon}{\phi \cdot b^* + (1-\alpha)\varepsilon}. \qquad ◀$$

▶ **Theorem 2.** *In expectation, it is rational for a miner Y to deviate from the honest strategy and join X in keeping the base fee low, if*

$$p_y > \frac{\Delta((1-\alpha)\varepsilon + \phi \cdot b^*)}{(1-\alpha)\Delta \cdot \varepsilon + (1+\Delta)\phi \cdot b^* - \alpha \cdot \varepsilon}.$$

Proof. We commence with the honest strategy of miner Y which we model as a Markov chain in Figure 8a. Miner Y starts in state Y_X^h and is tasked with proposing a block after miner X at an artificially lowered base fee b, where $b = (1-\phi)b^* < b^*$. The payout for proposing a block at twice the target size s^* is given by

$$P[Y_X^h] = s^* \left(\phi \cdot b^* + (1-\alpha)\varepsilon\right)(1+\Delta). \tag{A.3}$$

From state Y_X^h, we move to state Y_Y^h, where Y proposes a target size block, with probability p_y. The payout for miner Y in state Y_Y^h is given as

$$P[Y_Y^h] = s^* \cdot \varepsilon, \tag{A.4}$$

and we remain in this state for a subsequent block with probability p_y.

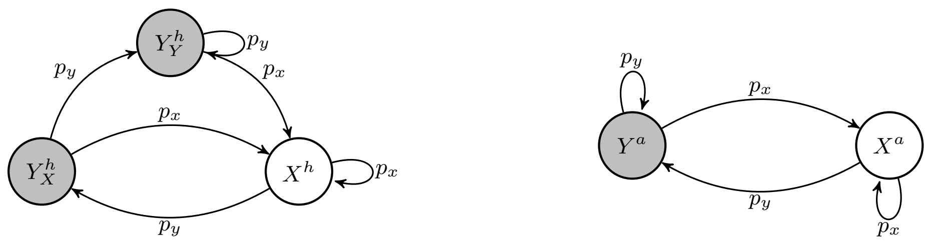
In both state Y_X^h and state Y_Y^h, the probability of moving to state X^h is p_x. When we re-enter state X^h for the first time, X will propose an empty block to lower the base fee again. Then, with a probability p_x, we remain in state X for the next block, where miner X will propose target size blocks until her consecutive turn as a proposer is interrupted. Regardless, the payout for miner Y whenever we are in state X^h is zero. We move to state Y_Y^h with probability p_y from state X^h and enter an absorbing state with probability $1 - p_y - p_x$ from all states.

Next, we calculate the expected payout of the honest strategy and start with the expected reward for miner Y starting from state Y_X^h $\mathbb{E}[R[Y_X^h]] = p_x \cdot \mathbb{E}[R[X^h]] + p_y \cdot \mathbb{E}[R[Y_Y^h]] + P[Y_X^h]$ where $\mathbb{E}[R[X^h]] = p_x \cdot \mathbb{E}[R[X^h]] + p_y \cdot \mathbb{E}[R[Y_X^h]]$, and $\mathbb{E}[R[Y_Y^h]] = p_x \cdot \mathbb{E}[R[X^h]] + p_y \cdot \mathbb{E}[R[Y_Y^h]] + P[Y_Y^h]$. By solving the system of linear equations, we find that the expected reward from the honest strategy is given by

$$\mathbb{E}[R[H]] = \mathbb{E}[R[Y_X^h]] \quad = \tfrac{(1-p_x)(((1-p_y)(1+\Delta)\phi \cdot b^*) + \varepsilon(1+\Delta-\alpha(1+\Delta)(1-p_y)-\Delta p_y))s^*}{(1-p_x-p_y)}. \tag{A.5}$$

We now consider the deviation from the honest strategy, whereby miner Y also keeps the base fee artificially low, and model the strategy with a Markov chain in Figure 8b. State Y^a is the starting state of the deviating strategy in which Y proposes a block with target size s^* at an artificially lowered base fee $(1-\phi)b^*$. Thus, the state's payout is

$$P[Y^a] = s^* \left(\phi \cdot b^* + (1-\alpha)\varepsilon\right). \tag{A.6}$$



(a) honest strategy of miner Y. **(b)** attacking strategy of miner Y.

Figure 8 The honest strategy (cf. Figure 8a) and the deviation from the honest strategy, i.e., keep the base fee artificially low, (cf. Figure 8b) modeled with discrete Markov chains for miner Y. We transition between states with every block. All states with a nonzero payout for miner Y are highlighted in gray. Note that for all remaining probabilities, the Markov process enters an absorbing state.

For the next block, we stay in state Y^a with probability p_y, move to state X^a with probability p_x, or else move to an absorbing state, i.e., the consecutive turns of X and Y as proposers end and the attack finishes. In state X^a, proposer X will also propose a target size block, but the payout for miner Y is zero as we assume no collaboration between the two. The transition probabilities from state X^a are identical to those from state Y^a: we move to state Y^a with probability p_y, stay in state X^a with probability p_x and enter an absorbing state with probability $1 - p_y - p_x$ for the next block. Thus, the expected reward for miner Y starting in state Y^a is given by $\mathbb{E}[R[Y^a]] = p_x \cdot \mathbb{E}[R[X^a]] + p_y \cdot \mathbb{E}[R[Y^a]] + P[Y^a]$, where $\mathbb{E}[R[X^a]]$ is the expected payout for miner Y starting from state X^a and we have $\mathbb{E}[R[X^a]] = p_x \cdot \mathbb{E}[R[X^a]] + p_y \cdot \mathbb{E}[R[Y^a]]$. We follow that the expected payout of the deviating strategy for miner Y is given by

$$\mathbb{E}[R[A]] = \mathbb{E}[R[Y^a]] = \frac{(1-p_x)s^*(\phi \cdot b^* + (1-\alpha)\varepsilon)}{(1-p_x-p_y)}. \tag{A.7}$$

We conclude it is rational for miner Y to deviate from the honest strategy when $\mathbb{E}[R[A]] - \mathbb{E}[R[H]] > 0$. It follows that Y attacks when

$$p_y > \frac{\Delta((1-\alpha)\varepsilon+\phi\cdot b^*)}{(1-\alpha)\Delta\cdot\varepsilon+(1+\Delta)\phi\cdot b^*-\alpha\cdot\varepsilon}.$$ ◀

▶ **Theorem 3.** *In expectation, it is rational for a miner Y to deviate from the honest strategy and lower the base fee, if*

$$p_y > \frac{\varepsilon(1-p_x)}{(1-\alpha)\varepsilon + \phi\cdot b^*(1-p_x) + (1+\Delta)\varepsilon\alpha p_x - \Delta p_x(\varepsilon + \phi\cdot b^*)}.$$

Proof. We, again, model the honest strategy for miner Y as a Markov chain (cf. Figure 9a). The honest strategy starts in state Y^h, where miner Y proposes a block at target size s^* and receives a payout of

$$P[Y^h] = s^* \cdot \varepsilon. \tag{A.8}$$

The transition probability to state X^h is p_x and to state Y^h, i.e., Y proposes consecutive blocks, is p_y. Otherwise, the consecutive turn of X and Y as miners stops and we enter an absorbing state. We note that the states X^h, Y^h_X and Y^h_Y correspond exactly to the eponymous states in Theorem 2 (cf. Figure 8a). Thus, the actions of the miners, the payout for miner Y, and the transition probabilities are as previously described in the proof of Theorem 2.

The strategy's payout is the expected reward starting from state Y^h, which is

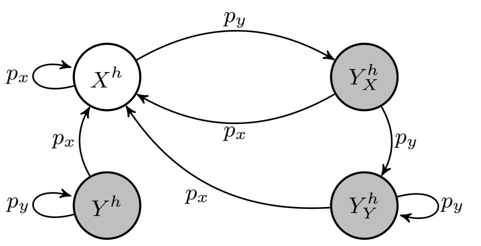
$$\mathbb{E}[R[Y^h]] = p_x \cdot \mathbb{E}[R[X^h]] + p_y \cdot \mathbb{E}[R[Y^h] + P[Y^h]. \tag{A.9}$$

As previously in Theorem 2, we have

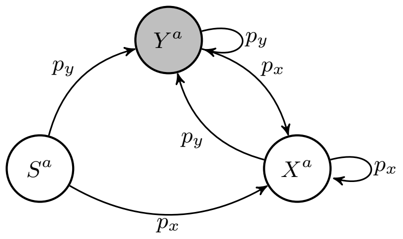
$$\mathbb{E}[R[Y^h_X]] = p_x \cdot \mathbb{E}[R[X^h]] + p_y \cdot \mathbb{E}[R[Y^h_Y]] + P[Y^h_X], \tag{A.10}$$
$$\mathbb{E}[R[X^h]] = p_x \cdot \mathbb{E}[R[X^h]] + p_y \cdot \mathbb{E}[R[Y^h_X]], \tag{A.11}$$
$$\mathbb{E}[R[Y^h_Y]] = p_x \cdot \mathbb{E}[R[X^h]] + p_y \cdot \mathbb{E}[R[Y^h_Y]] + P[Y^h_Y]. \tag{A.12}$$

Solving the system of linear equation, we conclude that the expected payout of the honest strategy is

$$\mathbb{E}[R[H]] = \mathbb{E}[R[Y^h]] = \frac{(((1+\Delta)b^*\phi p_x p_y) + \varepsilon(1 - p_x + ((1-\alpha)\Delta - \alpha)p_x p_y))s^*}{1 - p_x - p_y}. \tag{A.13}$$



(a) honest strategy of miner Y.



(b) attacking strategy of miner Y.

Figure 9 The honest strategy (cf. Figure 9a) and the deviation from the honest strategy (cf. Figure 9b) modeled with discrete Markov chains. We transition between states with every block. All states with a nonzero payout for miner Y are highlighted in gray. Note that for all remaining probabilities, the Markov process enters an absorbing state and the consecutive turn of X and Y as proposers finishes.

We proceed with the attack strategy, which we model in Figure 9b. Miner Y starts in state S^a and mines an empty block and, therefore, receives no rewards. With probability p_y we move to state Y^a for the next block, i.e., Y proposes a target size (s^*) block, with probability p_x we move to state X^a, i.e., X proposes a target size (s^*) block, and with probability $1 - p_x - p_y$ we move to an absorbing state, i.e., the attack ends. Notice that the states X^a and Y^a are identical to those described in the proof of Theorem 3. Thus, the expected returns starting in the respective states are as follows

$$\mathbb{E}[R[S^a]] = p_x \cdot \mathbb{E}[R[X^a]] + p_y \cdot \mathbb{E}[R[Y^a]], \tag{A.14}$$
$$\mathbb{E}[R[Y^a]] = p_x \cdot \mathbb{E}[R[X^a]] + p_y \cdot \mathbb{E}[R[Y^a]] + P[Y^a], \tag{A.15}$$
$$\mathbb{E}[R[X^a]] = p_x \cdot \mathbb{E}[R[X^a]] + p_y \cdot \mathbb{E}[R[Y^a]], \tag{A.16}$$

and find that the expected reward of the attack strategy is

$$\mathbb{E}[R[A]] = \mathbb{E}[R[S^a]] = \frac{(\phi \cdot b^* + (1-\alpha)\varepsilon)p_y s^*}{1 - p_x - p_y}. \tag{A.17}$$

To conclude, it is rational behavior for Y to deviate from the honest strategy, when $\mathbb{E}[R[A]] - \mathbb{E}[R[H]] > 0$, which holds when

$$p_y > \frac{\varepsilon(1-p_x)}{(1-\alpha)\varepsilon + \phi \cdot b^*(1-p_x) + (1+\Delta)\varepsilon\alpha p_x - \Delta p_x(\varepsilon + \phi \cdot b^*)}. \qquad ◀$$

B Delay Incurred by the Mitigation of Section 6

Suppose we are in a steady state with a base fee b^*, when after block height τ a fixed change in demand occurs for which the new steady state will be achieved with the new base fee $\beta \cdot b^*$. We denote by T the number of consecutively full blocks it takes to reach the new base fee $\beta \cdot b^*$.

After k consecutively full blocks, according to Eq. 6.1

$$\begin{aligned} s_{avg}[\tau + k] &= (1-q)2s^* + q \cdot s_{avg}[\tau + k - 1] \\ &= \big(2(1-q)(q^0 + q^1 + \ldots + q^{k-1}) + q^k\big)\, s^* \\ &= \left(2(1-q)\left(\frac{1-q^k}{1-q}\right) + q^k\right) s^* \\ &= (2-q^k)s^*. \end{aligned}$$

Plugging the above into Eq. 6.2 yields

$$\begin{aligned} b[\tau + k] &= b_{avg}[\tau + k - 1] \cdot \left(1 + \phi \cdot \frac{(2-q^k)s^* - s^*}{s^*}\right) = b_{avg}[\tau + k - 1] \cdot \big(1 + \phi(1-q^k)\big) \\ &= b[\tau] \cdot \big(1 + \phi(1-q^1)\big)\big(1 + \phi(1-q^2)\big) \cdots \big(1 + \phi(1-q^k)\big) \\ &= b^* \cdot \prod_{i=1}^{k}(1 + \phi(1-q^i)). \end{aligned}$$

Therefore, T is the smallest integer that satisfies

$$b[\tau + T] = b^* \cdot \prod_{i=k}^{T}(1 + \phi(1-q^k)) \geq \beta \cdot b^* \quad \Longleftrightarrow \quad \prod_{k=1}^{T}(1 + \phi(1-q^k)) \geq \beta.$$

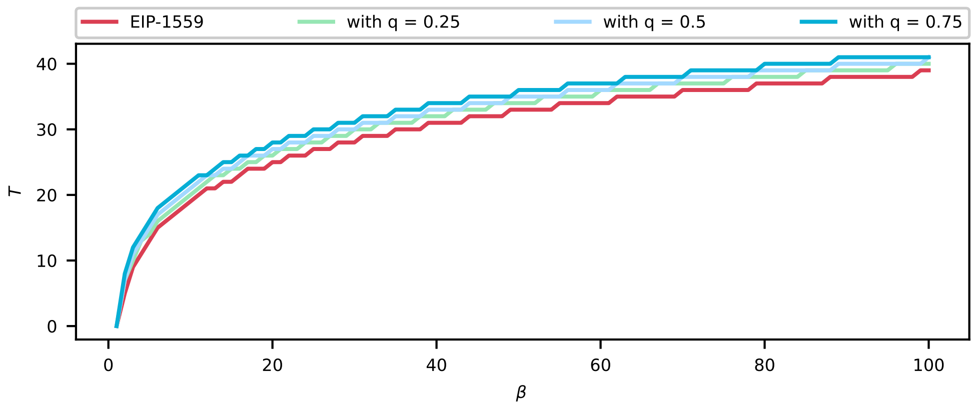


Figure 10 The number of consecutive full blocks T required to increase the base fee by factor β for EIP-1559 and the proposed mitigation for $q \in \{1/4, 1/2, 3/4\}$.

Figure 10 plots $T(\beta)$ for both EIP-1559 and our mitigation. We set $\phi = 1/8$ (current Ethereum) and use $q \in \{1/4, 1/2, 3/4\}$. All plots follow a logarithmic trend and the response times to dramatic changes in demand are only mildly affected by the proposed mitigation (even for β factors as large as 100).

On the Node-Averaged Complexity of Locally Checkable Problems on Trees

Alkida Balliu ✉
Gran Sasso Science Institute, L'Aquila, Italy

Sebastian Brandt ✉
Helmholtz Center for Information Security, Saarbrücken, Germany

Fabian Kuhn ✉
University of Freiburg, Germany

Dennis Olivetti ✉
Gran Sasso Science Institute, L'Aquila, Italy

Gustav Schmid ✉
University of Freiburg, Germany

Abstract

Over the past decade, a long line of research has investigated the distributed complexity landscape of locally checkable labeling (LCL) problems on bounded-degree graphs, culminating in an almost-complete classification on general graphs and a complete classification on trees. The latter states that, on bounded-degree trees, any LCL problem has deterministic *worst-case* time complexity $O(1)$, $\Theta(\log^* n)$, $\Theta(\log n)$, or $\Theta(n^{1/k})$ for some positive integer k, and all of those complexity classes are nonempty. Moreover, randomness helps only for (some) problems with deterministic worst-case complexity $\Theta(\log n)$, and if randomness helps (asymptotically), then it helps exponentially.

In this work, we study how many distributed rounds are needed *on average per node* in order to solve an LCL problem on trees. We obtain a partial classification of the deterministic *node-averaged* complexity landscape for LCL problems. As our main result, we show that every problem with worst-case round complexity $O(\log n)$ has deterministic node-averaged complexity $O(\log^* n)$. We further establish bounds on the node-averaged complexity of problems with worst-case complexity $\Theta(n^{1/k})$: we show that all these problems have node-averaged complexity $\widetilde{\Omega}(n^{1/(2^k-1)})$, and that this lower bound is tight for some problems.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases distributed graph algorithms, locally checkable labelings, node-averaged complexity, trees

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.7

Related Version *Full Version*: `https://arxiv.org/abs/2308.04251` [1]

Funding This work has been partially funded by the European Union - NextGenerationEU under the Italian Ministry of University and Research (MUR) National Innovation Ecosystem grant ECS00000041 - VITALITY – CUP: D13C21000430001, and by the German Research Foundation (DFG), Grant 491819048.

1 Introduction

The family of locally checkable labeling (LCL) problems was introduced in the seminal work of Naor and Stockmeyer [22] and since then, understanding the distributed complexity of computing LCLs has been at the core of the research on distributed graph algorithms. Roughly speaking, LCLs are labelings of the nodes or edges of a graph $G = (V, E)$ with labels from a finite alphabet such that some local, constant-radius condition holds at all the nodes. In the distributed context, G represents a network and one typically assumes that the nodes of G can communicate over the edges of G in synchronous rounds. If this communication is unrestricted, this is known as the LOCAL model of computation and if messages must consist of $O(\log n)$ bits (where n is the number of nodes), it is known as the CONGEST model. In our paper, we focus on the LOCAL model and we therefore do not explicitly analyze the required message sizes of our algorithms. We however believe that all our algorithms can be made to work in the CONGEST model with minor modifications.

Often LCL problems are studied in the context of bounded-degree graphs. In this case, LCLs include problems such as properly coloring the nodes of G with $\Delta + 1$ colors, where Δ is the maximum degree of G. By now, researchers have obtained a thorough understanding of the complexity landscape of distributed LCL problems in general bounded-degree graphs [13, 12, 16, 23, 6, 2] and also in more special graph families such as in particular in bounded-degree trees [17, 7, 12, 13, 3, 10]. Most of this work focuses on the classic notion of worst-case complexity: If all nodes start a computation at time 0 and communicate in synchronous rounds, how many rounds are needed until *all nodes* have decided about their outputs. In some case however, the worst-case round complexity might be determined by a small number of nodes that require a lot of time to compute their outputs, while most of the nodes find their outputs much faster. Consider for example the simple randomized $(\Delta + 1)$-coloring algorithm where in every step, every node picks a random available color and permanently keeps this color if there is no conflict. It is not hard to show that in every step, every uncolored node becomes colored with constant probability [18]. Hence, while we need $\Omega(\log n)$ steps (and thus also $\Omega(\log n)$ rounds) until all nodes are colored, for each individual node, the expected time to become colored is constant and consequently the time that nodes need on average to become colored is also constant w.h.p. In some contexts (e.g., when considering the energy cost of a distributed algorithm), this average completion time per node is more meaningful than the worst-case completion time and consequently, researchers have recently showed interest in determining the *node-averaged* time complexity of distributed graph algorithms [15, 9, 14, 5]. In the present paper, we continue this work and we study the *node-averaged complexity of LCL problems in bounded-degree trees.* Before describing our contributions, we first briefly summarize some of the relevant previous work.

Previous results on node-averaged complexity. The first paper that explicitly considered the node-averaged complexity of distributed graph algorithms is by Feuilloley [15]. The paper mainly considers LCL problems on paths and cycles (i.e., on graphs of maximum degree 2). It is known that on paths and cycles, when considering the worst-case complexity of LCL problems, randomization does not help and the only complexities that exist are $O(1)$, $\Theta(\log^* n)$, and $\Theta(n)$ [22, 12, 13]. In [15], it is shown that for deterministic algorithms, the worst-case complexity and the node-averaged complexity of LCL problems on paths and cycles is the same. This for example implies that the classic $\Omega(\log^* n)$ lower bound of [20] for coloring cycles with a constant number of colors also applies to node-averaged complexity. While this is true for deterministic algorithms, it is also shown in [15] that the randomized node-averaged complexity of 3-coloring paths and cycles is constant. As sketched above and also explicitly proven in [9], the same is true for the more general problem of computing a $(\Delta + 1)$-coloring in arbitrary graphs. While the results of [15] imply results for general LCLs on paths and cycles, the additional work on node-averaged complexity focused on the complexity of specific graph problems, in particular on the complexity of well-studied classic problems such as computing a maximal independent set (MIS) or a vertex coloring of the given graph. Barenboim and Tzur [9] show that in graphs of small arboricity, some coloring problems have a deterministic node-averaged complexity that is significantly smaller than the corresponding worst-case complexity. For example, it is shown that if the arboricity is constant, an $O(k)$-vertex coloring can be computed in node-averaged complexity $O(\log^{(k)} n)$ for any fixed integer $k \geq 1$, where $O(\log^{(k)} n)$ denotes the k times iterated logarithm of n. As one of the main results of [5], it was shown that the MIS lower bound of [19] can be generalized to show that even with randomization, computing an MIS on general (unbounded degree) graphs requires node-averaged complexity $\Omega(\sqrt{\log n / \log\log n})$. Hence, while the problem of coloring with $(\Delta + 1)$ colors and, as also shown in [5], the problem of computing a 2-ruling set have randomized algorithms with constant node-averaged complexity, the same thing is not true for the problem of computing an MIS.

LCL complexity in bounded-degree trees. One of the goals of this paper is to make a step beyond understanding individual problems and to start studying the landscape of possible node-averaged complexities of general LCL problems. We do this by studying LCL problems on bounded-degree trees, a graph family that we believe is relevant and that has recently been studied intensively from a worst-case complexity point of view (e.g., [12, 13, 11, 7, 16, 23, 17]). In bounded-degree trees, for deterministic algorithms, exactly the following worst-case complexities are possible: $O(1)$, $\Theta(\log^* n)$, $\Theta(\log n)$, and $\Theta(n^{1/k})$ for some fixed integer $k \geq 1$. It was shown in [17] (and earlier for a special subclass of LCLs in [7] and for paths in [22, 13]) that on bounded-degree trees, there are no deterministic or randomized optimal algorithms with a time complexity in the range $\omega(1)$ to $o(\log^* n)$. Further, in [12], it was shown that even for general bounded-degree graphs, there are no deterministic LCL complexities in the range $\omega(\log^* n)$ to $o(\log n)$. Finally, it was shown in [13] that every LCL problem that requires $\omega(\log n)$ rounds on bounded-degree trees has a worst-case deterministic and randomized complexity of the form $\Theta(n^{1/k})$ for some fixed integer $k \geq 1$ (and all those complexities also exist). It is further known that randomization can only help for LCL problems with a deterministic complexity of $\Theta(\log n)$. Those problems have a randomized complexity of either $\Theta(\log n)$ or $\Theta(\log\log n)$ (and both cases exist) [13, 11].

1.1 Our Contributions

As our main result, we show that the $\Theta(\log n)$ complexity class vanishes when considering the node-averaged complexity of LCLs on bounded-degree trees.

▶ **Theorem 1.** *Let Π be an LCL problem for which there is an $O(\log n)$-round deterministic algorithm on bounded-degree trees. Then, Π can be solved deterministically with node-averaged complexity $O(\log^* n)$ on bounded-degree trees.*

A standard example for an LCL problem that requires $\Theta(\log n)$ rounds deterministically is the problem of 3-coloring a tree. So for 3-coloring Theorem 1 states that there is a deterministic distributed 3-coloring algorithm, for bounded degree trees, with node-averaged complexity $O(\log^* n)$ rounds. Meaning that the average node terminates after $O(\log^* n)$ rounds. Note that for 3-coloring trees deterministically, this is tight. As shown in [15], 3-coloring has deterministic node-averaged complexity $\Omega(\log^* n)$ even on paths. Below, we will use the 3-coloring problem as a simple example to illustrate some of the challenges in obtaining the above theorem, but first we state the rest of our results.

In addition to Theorem 1, we also investigate the node-averaged complexity of LCL problems that require polynomial time in the worst case (i.e., time $\Theta(n^{1/k})$ for some integer $k \geq 1$). We show that for such problems, also the node-averaged complexity is polynomial. However at least in some cases, it is possible to obtain a node-averaged complexity that is significantly below the worst-case complexity. In [13], the hierarchical $2\frac{1}{2}$-coloring problem with parameter k is defined as an example problem with worst-case complexity $\Theta(n^{1/k})$. We show that the node-averaged complexity of this LCL problem is significantly smaller.

▶ **Theorem 2.** *The deterministic node-averaged complexity of the hierarchical $2\frac{1}{2}$-coloring problem with parameter k is $O(n^{1/(2^k-1)})$.*

Finally, we show that for a problem with worst-case complexity $\Theta(n^{1/k})$, this is essentially the best possible node-averaged complexity. Meaning that we also prove that our algorithm for hierarchical $2\frac{1}{2}$-coloring problems is optimal up to one $\log n$ factor.

▶ **Theorem 3.** *Let Π be an LCL problem with (deterministic or randomized) worst-case complexity $\Omega(n^{1/k})$. Then, the randomized node-averaged complexity of Π is $\Omega(n^{1/(2^k-1)}/\log n)$.*

Note that the algorithm of Theorem 2 is deterministic, and that the lower bound of Theorem 3 holds for randomized algorithms as well.

1.2 High-level Ideas and Challenges

We next discuss some of the ideas that lead to the known results about solving LCL problems on bounded-degree trees and we highlight some of the challenges that one has to overcome and some of the ideas we use to prove Theorems 1–3.

Rake-and-compress decomposition. We start by sketching a generic algorithm that can be used to solve all LCL problems in bounded-degree trees. The generic algorithm can be used to obtain algorithms with an asymptotically optimal worst-case complexity for all problems with worst-case complexity $\Omega(\log n)$. As a first step, the algorithm uses a technique that is known as *rake-and-compress* [21] to partition the nodes of a given tree $T = (V, E)$ into $O(\log n)$ layers such that each layer is either a rake layer that consists of a set of independent nodes or it is a compress layer that consists of a sufficiently separated set of paths. Every node in a rake layer has at most one neighbor in a higher layer, and in each path of a compress layer, the two nodes at the end have exactly one neighbor in a higher layer and the other nodes on the path have no neighbors in a higher layer.[1] Such a decomposition can be computed in an iterative process that produces the layers in increasing order. Given some tree (or forest), a rake layer can be obtained by taking the set of all leaf nodes[2] and a compress layer can be created by the paths (or more precicely by the inner part of the paths) induced by degree-2 nodes. It is not hard to show that when alternating rake and compress layers, this process completes after creating $O(\log n)$ layers [21].

Applying the decomposition. As an example of how to use rake-and-compress to solve an LCL problem, we look at the case of 3-coloring the nodes of a tree T. Given a decomposition into rake and compress layers, this can be done in $O(\log n)$ rounds. First, color each of the paths of the compress layers with $O(1)$ colors. This requires $O(\log^* n)$ rounds. Then, the 3-coloring of T is computed by starting at the highest layer of the decomposition. When processing a rake layer, each node is colored with a color different from its (at most one) neighbor in a higher layer. When processing a compress layer, we just have to 3-color the

[1] The actual decomposition that we use is a bit more complicated and the formal definition (see Definition 4) requires some additional details.
[2] When two degree-1 nodes are neigbors, one just takes one of the two nodes.

paths of the layer such that each node at the end of a path picks a color that differs from the color of its neighbor in a higher layer. Given the initial $O(1)$-coloring of the path, this can be done in constant time for each path. The time to compute the coloring is therefore proportional to the number of layers and thus $O(\log n)$. The generic algorithm for solving more general LCL problems is more involved, but still similar at a high level. While creating the decomposition, for each node v, one can create a list of labels that can be assigned to v such that the labeling of lower layer nodes that depend on v can still be completed. The LCL problem needs to allow labelings that are flexible enough such that when having long paths of nodes that each can be the root of an arbitrary subtree, the nodes of the path can still be labeled efficiently (in constant time given an appropriate initial coloring of the path).

Implementation with low node-averaged complexity. The main challenge to achieve node-averaged complexity $o(\log n)$ is the following. The generic algorithm first computes the decomposition and it then computes the labeling by starting with the nodes in the highest layers. In the worst case, we thus need $\Theta(\log n)$ rounds before even the label of a single node is determined. Moreover, most of the nodes are in the lowest layers, which are labeled at the very end of the algorithm. We therefore need to label most of the nodes already in the "bottom-up" phase when creating the rake and compress layers. For some problems, this is challenging: for example, in the 3-coloring problem, if we ever obtain a node with 3 neighbors of lower layers that have 3 different colors, then we cannot complete the solution in any valid way. Hence, we have to label the nodes in such a way that the "top-down" phase is still able to extend the partial labeling to a valid labeling of all the nodes. In the following high-level discussion, for simplicity, we assume that the tree has diameter $O(\log n)$ so that it suffices to create rake layers and we do not need compress layers. We further only look at the problem of 3-coloring T. This problem is significantly easier to handle than the general case. The solution for 3-coloring however already requires some of the ideas of the general case.

Let us therefore assume that we have an $O(\log n)$-diameter $O(1)$-degree tree T. If we only construct rake layers, we obtain $O(\log n)$ layers, where each layer is an independent set and except for a single node u in the top layer, every node has exactly one neighbor in a higher layer. We refer to u as the root and for each other node, we refer to the single neighbor in a higher layer as the parent. Note that when assigning a color to a node v in the top-down phase, only v's parent has already been assigned a color. To complete the top-down phase, it therefore suffices if every node v can choose its color from an arbitrary subset S_v of size 2 of the colors. Hence, if we try to color some nodes already in the bottom-up phase, we have to make sure that all the uncolored nodes still have at least two available colors. This is for example guaranteed as long as every uncolored node has at most one colored neighbor.

When constructing the layering we therefore proceed as follows. We only color nodes that have already been assigned to some rake layer. Whenever we decide to color a node v in the bottom-up phase, we also directly color the whole subtree of v.[3] The high-level idea of the algorithm to achieve this is as follows. After each rake step, i.e., after each creation of a new layer, we check whether or not there are some nodes that can be colored. Consider the situation after the t^{th} rake step, let $G^{(t)}$ be the set of nodes that have not been raked at that time (i.e., that have not been assigned to some layer), and let $R^{(t)}$ be the set of nodes that have already been assigned to some layer. Note that if a node $u \in G^{(t)}$ has some neighbor $v \in R^{(t)}$, then u will in the end be the unique neighbor of v in a higher layer. We

[3] After coloring the root of a subtree, the coloring of the subtree can be done in parallel while proceedings with the rest of the algorithm.

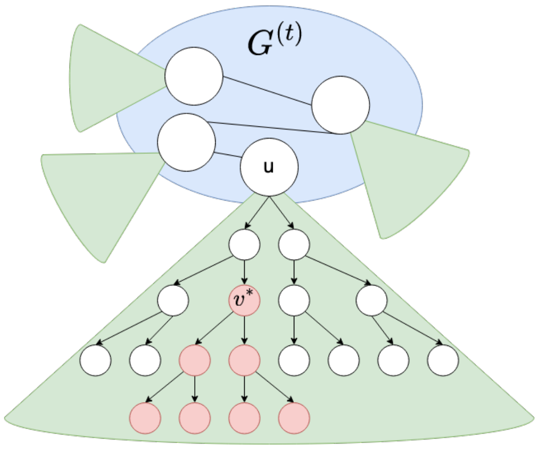


Figure 1 The graph $G^{(t)}$ of nodes that are not yet raked away is colored blue. The already raked away nodes $R^{(t)}$ are colored green. The node u chooses v^* since it has the largest subtree, colored in red, attached and both v^* as well as its entire subtree become colored.

can therefore think of the nodes in $G^{(t)}$ as the roots of the already raked subtrees. This is illustrated in Figure 1. After each rake step t, each node $u \in G^{(t)}$ tries to color some node at distance 2 in its subtree.[4] Node u chooses v^* to be a node at distance 2 in its subtree such that the subtree rooted at v^* has the largest number of uncolored nodes among all nodes at distance 2 of u in the subtree of u (observe that nodes can keep track of such numbers). If there are no colored 2-hop neigbors of v^* outside the subtree of v^* (i.e., no colored siblings of v^*), then u decides to color v^* and its complete subtree. Otherwise, no new nodes in u's subtree are getting colored. If v^* and its subtree get colored, then a constant fraction of the uncolored nodes in u's subtree get colored. Otherwise, a sibling v' of v^* with a larger subtree has already been colored while u was the root of the tree. Note that at this time, the subtree of v^* was already in the same state and therefore v' colored more nodes than v^* does. One can use this to show that whenever the height of a raked subtree increases, a constant fraction of the uncolored nodes gets colored. One can further show that this suffices to show that over the whole tree, a constant fraction of the remaining nodes gets colored every constant number of rounds and thus the node-averaged complexity is constant. The algorithm and the analysis for the general family of LCLs for which Theorem 1 holds uses similar basic ideas, dealing with the general case is however significantly more involved.

Improved upper bounds in the polynomial regime. We prove that the node-averaged complexity of the hierarchical $2\frac{1}{2}$-coloring problem with parameter k is $O(n^{1/(2^k-1)})$. In order to give some intuition for this, we focus on the case $k = 2$ where the worst-case complexity is $\Theta(\sqrt{n})$. Instead of providing a formal definition of the problem, it is helpful to present the problem by describing how a worst-case instance for the problem looks like, and how a solution in such an instance looks like. A worst-case instance for this problem consists of a path P of length $\Theta(\sqrt{n})$, where to each node v_j of P is attached a path Q_j of length $\Theta(\sqrt{n})$. We call the nodes of the path P *p-nodes* and we call the nodes of a path Q_j *q-nodes*. For each path Q_j, the algorithm has to decide to either 2-color it or to mark the whole path as

[4] By only coloring nodes at distance at least 2 from u, we make sure that neighbors of nodes that are not yet layered remain uncolored.

decline. Then, the subpaths of P induced by nodes that are neighbors of q-nodes that output decline need to be labeled with a proper 2-coloring. In particular, *decline* is not allowed on p-nodes. Let us now describe an algorithm with optimal worst-case complexity for instances with a similar structure, but where the paths may have different lengths. For q-nodes, the algorithm first checks if the length of the path containing those nodes is $O(\sqrt{n})$ (note that, in order to perform this operation, the algorithm needs to know n, and it is actually unknown whether an LCL problem can have $\Theta(\sqrt{n})$ worst-case complexity when n is unknown). In such a case, the algorithm is able to produce a proper 2-coloring of the path. Otherwise, the path is marked as decline. Then, it is possible to prove that the subpaths of P induced by nodes having q-node neighbors that output decline must be of length $O(\sqrt{n})$, and hence they can be properly 2-colored in $O(\sqrt{n})$ rounds. We observe that in the worst-case instance described above, the majority of the nodes of the graph are q-nodes, and hence, from an average point of view, it would be fine if p-nodes spend more time. In fact, it is possible to improve the node-averaged complexity of the described algorithm by letting q-nodes run for at most $O(n^{1/3})$ rounds and p-nodes for at most $O(n^{2/3})$ rounds. In this case, a worst-case instance contains a path P of length $O(n^{2/3})$ and all paths Q_j are of length $O(n^{1/3})$. We obtain that both the p-nodes and the q-nodes contribute $O(n^{4/3})$ to the sum of the running times, obtaining a node-averaged complexity of $O(n^{1/3})$.

Lower bounds in the polynomial regime. It is known by [10] that if an LCL problem Π has worst-case complexity $o(n^{1/k})$, then it can actually be solved in $O(n^{1/(k+1)})$ rounds. The intuition about what determines the exact value of k in the complexity of a problem is related to how many compress layers of a rake-and-compress decomposition one can handle. In the example presented above, namely 3-coloring, one can handle an arbitrary number of compress paths and that is the reason why the problem can be solved in $O(\log n)$ rounds. In particular, no matter how many rake or compress operations have been applied, we can handle any compress path by producing a 3-coloring on it and leaving the endpoints uncolored (such nodes can decide their color after their higher layer neighbors picked a color), and this can be done fast. Not all problems are of this form, that is, for some problems we cannot handle an arbitrary amount of compress paths: it is possible to define problems in which different labels need to be used in compress paths of different layers (hierarchical $2\frac{1}{2}$ coloring is indeed such a problem where in fact p-nodes are not allowed to output *decline*). For such problems, it may not be possible at all to efficiently produce a valid labeling for long compress paths of layers that are too high, say of layers strictly more than k. In order to solve this issue, we can modify the generic algorithm sketched above by increasing the number of rake operations that are performed between each pair of compress operations. When using $\Omega(n^{1/(k+1)})$ rake operations at the beginning and between any two compress operations, the total number of compress layers is at most k. This however makes the algorithm slower, resulting in a complexity of $\Theta(n^{1/(k+1)})$ (while 3-coloring has worst-case complexity $\Theta(\log n)$).

In other words, for some LCL problems, compress paths are something that is difficult to handle, and the number of compress layers that we can recursively handle is what determines the complexity of a problem. If we can handle an arbitrary amount of compress layers, then the problem can be solved in $O(\log n)$ rounds, but if we can handle only a constant amount of compress layers, say k, then the complexity of the problem is $\Theta(n^{1/(k+1)})$. In [10] it is proved that, if a problem has complexity $o(n^{1/k})$, then it is possible to handle k compress layers, implying a complexity of $O(n^{1/(k+1)})$. We show that the same can be obtained by starting from an algorithm $\mathcal{A}$ with node-averaged complexity $o(n^{1/(2^k-1)}/\log n)$, implying that if a problem has complexity $\Omega(n^{1/k})$, then it cannot have node-averaged complexity

$o(n^{1/(2^k-1)}/\log n)$, since otherwise it would imply that the problem can actually be solved in $O(n^{1/(k+1)})$ rounds in the worst case, which then leads to a contradiction. Starting from an algorithm that only has guarantees on its node-averaged complexity instead of on its worst-case complexity introduces many additional challenges that we need to tackle. For example, in [10] it is argued that an $o(n^{1/k})$-rounds algorithm can never see both the endpoints of a carefully crafted path that is too long. This kind of reasoning, that is very common when we deal with worst-case complexity, does not work for node-averaged complexity.

Road Map. Section 2 provides some important definitions and a high-level description of the generic algorithm to solve LCLs on bounded-degree trees with optimal worst-case complexity. In Section 3, we present an algorithm with node-averaged complexity $O(\log^* n)$ that is able to solve all problems that have $O(\log n)$ worst-case complexity. The algorithm is based on the one discussed in Section 2.1, but we need to tackle several challenges to improve its node-averaged complexity. The proofs of Theorems 2 and 3 are in Appendices A and B.

2 Preliminaries

Node-averaged complexity. We start by defining the notion of node-averaged complexity as in [5]. Let $\mathcal{A}$ be an algorithm that solves a problem Π. Assume $\mathcal{A}$ is run on a given graph $G = (V, E)$. Let $v \in V$. We define $T_v^G(\mathcal{A})$ to be the number of rounds after which v terminates when running $\mathcal{A}$. The node-averaged complexity of an algorithm $\mathcal{A}$ on a family of graphs $\mathcal{G}$ is defined as follows.

$$\mathsf{AVG}_V(\mathcal{A}) := \max_{G\in\mathcal{G}} \frac{1}{|V|} \cdot \mathbb{E}\left[\sum_{v\in V(G)} T_v^G(\mathcal{A})\right] = \max_{G\in\mathcal{G}} \frac{1}{|V|} \cdot \sum_{v\in V(G)} \mathbb{E}\left[T_v^G(\mathcal{A})\right]$$

The complexity of Π is defined as the lowest complexity of all the algorithms that solve Π.

LCLs in the black-white formalism. We next define the class of problems that we consider: LCLs in the black-white formalism. A problem Π described in the black-white formalism is a tuple $(\Sigma_{\text{in}}, \Sigma_{\text{out}}, C_W, C_B)$, where:

- Σ_{in} and Σ_{out} are finite sets of labels.
- C_W and C_B are both multisets of pairs, where each pair $(\ell_{\text{in}}, \ell_{\text{out}})$ is in $\Sigma_{\text{in}} \times \Sigma_{\text{out}}$.

Solving a problem Π on a graph G means that:

- $G = (W \cup B, E)$ is a graph that is properly 2-colored, and in particular each node $v \in W$ is labeled $c(v) = W$, and each node $v \in B$ is labeled $c(v) = B$.
- To each edge $e \in E$ is assigned a label $i(e) \in \Sigma_{\text{in}}$.
- The task is to assign a label $o(e) \in \Sigma_{\text{out}}$ to each edge $e \in E$ such that, for each node $v \in W$ (resp. $v \in B$) it holds that the multiset of incident input-output pairs is in C_W (resp. in C_B).

Note that when expressing a given LCL problem on a tree T in the black-white formalism, we often have to modify the tree T as follows. We subdivide every edge e of T by inserting one node in the middle of the edge. Each edge is then split into two "half-edges" and the new tree is trivially properly 2-colored (say the original nodes of T are the black nodes and the newly inserted nodes for each edge of T are the white nodes). In the full version [1], we prove that on trees, for any standard LCL, we can define an LCL in the black-white formalism that has the same asymptotic node-averaged complexity as the original one, implying that our results hold for all standard LCLs as well.

2.1 A Generic Way to Solve All LCLs

The content of this section is heavily based on results presented in [13, 10, 4]. We define two elementary procedures.

- **Rake:** Every node of degree 1 or 0 gets removed.
- **Compress(b):** Every path of degree-2 nodes of length at least b is split into subpaths of length in $[\ell, 2\ell]$ by ignoring one node between each two such subpaths. We remove all of the subpaths and leave the ignored nodes to be removed by the next rake step.

We first start by providing an $O(D)$ rounds algorithm: iteratively apply rake until the entire tree is removed; at each step, each node v that becomes a leaf computes the set L of all labels that can be put on the edge connecting v to its parent, satisfying that, for any choice in L, it is possible to pick labels from the sets assigned to the edges connecting v to its children, in a valid manner; once all edges get a set assigned, it is possible to pick a valid labeling for all the edges by processing nodes in reverse order.

Informally, we call the set of labels computed by a node the *class* of the subtree rooted at that node. In order to obtain algorithms that are faster than $O(D)$ rounds, we must also handle nodes of degree 2, and hence we also have to take care of paths obtained with the compress operation. Observe that each such path has *two* parents, that is, the two nodes of higher layers connected to its endpoints. We hence need a way to assign a class to the whole path, as a function of the classes of the subtrees of lower layers connected to the nodes of the path. However, this is challenging: how do we even *define* the class of a path? We would like that, when we process nodes in reverse order, no matter what are the *two* labels that get chosen for the edges connecting the endpoints of the path to nodes of higher layers, we can still complete the labeling inside the path (and in the subtrees connected to it). Hence, we still want to assign a set of labels to each edge connecting the path to its parents, but now these two sets must satisfy some sort of independence property. We call a pair of sets of labels that satisfy this property an *independent class* of the path.

In [13], Chang and Pettie showed that for every LCL that is solvable in $O(\log n)$ rounds, there exists some constant ℓ such that, if the paths have length at least ℓ, then there is always a way to compute an independent class, such that the algorithm sketched above works. In [10] it is then shown that a similar procedure works for all problems with complexity $\Theta(n^{1/k})$. The actual complexity of a problem is determined by the number of compress layers for which it is possible to compute an independent class. Let k be this number. If $k = \infty$, the problem Π has worst-case complexity $O(\log n)$, while if k is some constant then Π has worst case complexity $\Theta(n^{1/k})$. We will now see how to decompose any tree such that we apply the compress operation only k times.

Tree decompositions. All problems with worst-case complexity $O(\log n)$ or $O(n^{1/k})$ for any $k \in \mathbb{N}$ can be solved by following a generic algorithm [13, 10, 4]. This algorithm decomposes the tree into layers by iteratively removing nodes in a rake-and-compress manner [21] (and then uses the computed decomposition to solve the given problem). We first define the decomposition and then elaborate on how fast (and how) it can be computed.

▶ **Definition 4** ((γ, ℓ, L)-decomposition). *Given three integers γ, ℓ, L, a (γ, ℓ, L)-decomposition is a partition of $V(G)$ into $2L - 1$ layers $V_1^R = (V_{1,1}^R, \ldots, V_{1,\gamma}^R), \ldots, V_L^R = (V_{L,1}^R, \ldots, V_{L,\gamma}^R)$, $V_1^C, \ldots, V_{L-1}^C$ such that the following hold.*

1. *Compress layers: The connected components of each $G[V_i^C]$ are paths of length in $[\ell, 2\ell]$, the endpoints have exactly one neighbor in a higher layer, and all other nodes do not have any neighbor in a higher layer.*

2. *Rake layers: The diameter of the connected components in $G[V_i^R]$ is $O(\gamma)$, and for each connected component at most one node has a neighbor in a higher layer.*
3. *The connected components of each sublayer $G[V_{i,j}^R]$ consist of isolated nodes. Each node in a sublayer $V_{i,j}^R$ has at most one neighbor in a higher layer or sublayer.*

In [13, 10] it is shown how to compute a (γ, ℓ, L)-decomposition where, at the end, each node knows the layer that it belongs to. On a high level, the algorithm alternates between performing γ Rake operations and one Compress(ℓ) operation, until the empty tree is obtained. The following lemma provides upper bounds for the deterministic worst-case complexity of computing a (γ, ℓ, L)-decomposition using the algorithm of [13, 10].

▶ **Lemma 5** ([13, 10]). *Assume $\ell = O(1)$. Then the following hold.*
- *For any positive integer k and $\gamma = n^{1/k}(\ell/2)^{1-1/k}$, a (γ, ℓ, k)-decomposition can be computed in $O(kn^{1/k})$ rounds.*
- *For $\gamma = 1$ and $L = O(\log n)$, a (γ, ℓ, L)-decomposition can be computed in $O(\log n)$ rounds.*

By Lemma 5, we get that if in the former case we set $L = k$, and in the latter case we set $L = O(\log n)$, then a (γ, ℓ, L)-decomposition can be computed within a running time that matches the target complexity. In [13, 10] it is shown how to determine the value of ℓ in each case. We now define a total order on the layers of a (γ, ℓ, L)-decomposition in the natural way. This will be useful in the design of our algorithm in Section 3.

▶ **Definition 6** (layer ordering). *We define the following total order on the (sub)layers of a (γ, ℓ, L)-decomposition.*
- $V_{i,j}^R < V_{i',j'}^R$, *iff* $i < i' \vee (i = i' \wedge j < j')$
- $V_{i,j}^R < V_i^C$
- $V_i^C < V_{i+1,j}^R$

Accordingly, we will use terms such as "lower layer" to refer to a layer that appears earlier in the total order than some considered other layer.

The generic algorithm with optimal worst-case complexity. We now explain the algorithm due to [13, 10, 4] that is able to solve any LCL Π with worst-case complexity $\Theta(\log n)$ or $\Theta(n^{1/k})$ asymptotically optimally.

1. Determine ℓ and k from the description of Π. Compute γ, L accordingly.
2. Compute a (γ, ℓ, L)-decomposition, while also propagating label sets up.
3. Any node without neighbours in a higher layer picks a solution based on the label sets of their children and propagate their choice downwards.
4. Nodes that receive a choice from their parents simply pick a label respecting the label sets of their children.

Because of the way the label sets were chosen all nodes will be able to pick a valid label. For the details of how the label sets are chosen and why all nodes will be able to pick a valid label, we refer to the full version of the paper [1].

We note that the generic algorithm does not require a specific (γ, ℓ, L)-decomposition – any (γ, ℓ, L)-decomposition (for the parameters γ, ℓ, L determined in the beginning of the generic algorithm) works. We will make use of this fact when designing algorithms with a good node-averaged complexity in Section 3.

3 Algorithm for Intermediate Worst-Case Complexity Problems

In this section, we provide an algorithm with node-averaged complexity $O(\log^* n)$ on bounded-degree trees for all LCLs that can be solved in worst-case complexity $O(\log n)$ on such trees. The main difference to the generic algorithm is that we compute our (γ, ℓ, L)-decomposition in such a way that we obtain many more local maxima, that is, nodes with a layer number that is higher than the ones of all its neighbors. We achieve this by leaving slack at the end of compress paths and then inserting new compress paths to create local maxima.

3.1 The Decomposition Algorithm

Our algorithm is essentially a modified rake and compress procedure. During the execution of the algorithm we will maintain a partial (γ, ℓ, L)-decomposition and change it to create local maxima where we want them. For now we start by distinguishing between nodes that have already been assigned a layer and those that have not.

▶ **Definition 7** (free and assigned nodes). *We call a node that has not been assigned to a layer a* free *node. A node that has been assigned to a layer is called* assigned*.*

Let G denote our input tree and assume that a subset of nodes has already been assigned to some layers. Let G' denote the subgraph of G induced by all assigned nodes. Recall that we have a total ordering on the layers of a decomposition due to Definition 6 (that naturally extends to partial decompositions). Our aim is to create local maxima where they are most useful to us.

▶ **Definition 8** (local maximum). *A* local maximum *is an assigned node $v \in V(G')$ with the following two properties:*
1. *Node v and all of its neighbors are assigned, i.e., they are all contained in $V(G')$.*
2. *For each neighbor w of v, the layer of w is strictly smaller than the layer of v.*

In our algorithm, we will artificially promote some nodes to a higher layer in order to produce local maxima. We choose which node to promote according to the *quality* of the nodes. The quality of a node v is the number of nodes that are waiting for v to propagate its label downwards. So if v terminates and chooses an output, then all of these nodes can also terminate. To keep track of these dependencies, we orient (some of the) edges of the input graph in such a way that any node u will have an oriented path from v to u if and only if u will be able to terminate if v does (see Figure 2). In practice, this means that, if u is raked away, we orient the single edge from u's parent towards u and we orient the ends of compress paths inwards. If an edge is not explicitly oriented it is not considered oriented at all. For some given v this orientation now defines $H(v)$, a subgraph of nodes that can be reached over oriented paths. In other words, $H(v)$ contains all the nodes that are only waiting for v to choose an output and hence these could all terminate if v became a local maximum.

▶ **Definition 9** (quality). *For any node $v \in V(G)$, let $H(v)$ denote the set of all nodes w that can be reached from v via a path $(v = v_0, v_1, \ldots, v_j = w)$ such that the following hold:*
1. *The edge $\{v_{i-1}, v_i\}$ is oriented from v_{i-1} to v_i, for each $1 \leq i \leq j$.*
2. *All nodes on the subpath from v_1 to w are assigned, i.e., they are all contained in $V(G')$.*
3. *The layer of v_i is smaller than or equal to the layer of v_{i-1}, for each $2 \leq i \leq j$, and if v_0 is assigned, then the layer of v_1 is smaller than or equal to the layer of v_0.*

If v is a local maximum, or a descendant of a local maximum, then the quality $q(v) := 0$. Otherwise the quality *$q(v)$ of a node v is the number of nodes in $H(v)$, i.e., $q(v) := |H(v)|$.*

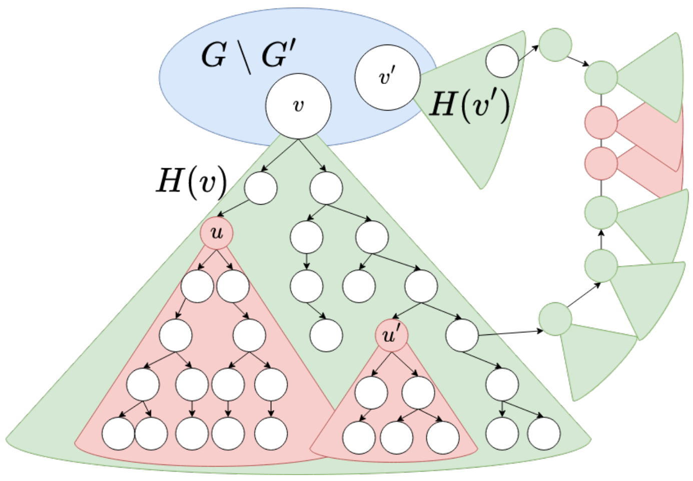


Figure 2 v and v' are free nodes. $H(v)$ (respectively, $H(v')$) contains all nodes inside the green cone attached to v (respectively, v'). The path connecting $H(v)$ and $H(v')$ is a compress path were the ends are oriented inwards. As a result the green nodes at the end of that path contribute to $H(v)$ and $H(v')$ respectively. u and u' are local maxima, so because of Item 3 in Definition 9 they and the red trees hanging from them do not contribute to $H(v)$ (respectively, $H(v')$).

We chose the name quality, since we will later decide on which nodes to fix by trying to maximize this quantity. To be more precise when stating the algorithm we also introduce the following notions based on the orientation.

▶ **Definition 10** (child, parent, descendant, ancestor, orphan). *For any edge $\{w, w'\}$ oriented from w to w', we call w' a* child *of w and w the* parent *of w'. For any oriented path $(w, \dots, w')$ that is consistently oriented from w to w', we call w' a* descendant *of w and w an* ancestor *of w'. We call a node with no edges oriented towards itself an* orphan*.*

In previous work the rake and compress procedure is done by iteratively performing some rakes and then one compress. We change the ordering by first performing one compress and then some rakes. To still get the same guarantees our algorithm initially performs γ rakes. Each compress operation is done in a modified, non-standard, way: normally to remove as many nodes as possible in each iteration, we would like to compress paths that are as short as possible. Instead we make sure we have some extra slack at the end of each compress path. This is to ensure that compress paths are always far enough away from nodes that we want to promote. We then rake away this slack, by performing a set of γ rakes.

After we are done with compress and rakes we want to promote some node v^*. We define the set $C_b(r)$ as the set of descendants of r that have distance exactly b from r. In Figure 2, e.g., $C_2(v)$ are the nodes inside the green cone of v that are at distance exactly 2 from v, so $u \in C_2(v)$. We determine the node that we want to promote as the node v^* of highest quality among all nodes in $C_b(r)$. We will later see that if v^* is promoted, the quality of r is reduced by a constant factor. We further define $G^{(i)}$ as the set of free nodes at the end of iteration i. The full details of the algorithm are given in the full version [1]. Note that Algorithm 1 provides a description of the steps of the algorithm without specifying how the algorithm is implemented in a distributed manner. We will take care of the latter in Section 3.3.

We will use this result to show that $O(\log n)$ iterations of Algorithm 1 are enough.

▶ **Lemma 11** ([10]). *Given a tree with n nodes, by performing α rakes and 1 compress with minimum path length β, the number of remaining nodes is at most $\frac{\beta}{2\alpha}n$.*

Algorithm 1 Compute Decomposition *Informal.*

```
   Input: $G = (V, E), \Pi$
1  compute $\ell$ from $\Pi$
2  $b \leftarrow \ell + 2$
3  $\gamma \leftarrow \ell + 3$
4  Perform $\gamma$ orienting Rakes
5  for $O(\log n)$ times (until every node is assigned to a layer) do
6      for each path with length at least $4\ell + 9$ do
7          Ignore the first and last $\gamma$ nodes                ▷ Will be raked away.
8          Perform normal compress on the truncated path
9          Orient the ends inwards
10     Perform $\gamma$ orienting Rakes          ▷ Thereby raking away the slack from compress
11     for each free node $r$ do
12         $v^*$ is the descendant at distance $b$ with the largest quality
13         if The path from $r$ to $v^*$ does not intersect with any compress path then
14             Promote $v^*$ into a local maximum by reassigning the path from $r$ to $v^*$ to
                 a compress layer with $v^*$ as the end point in the next higher rake layer
```

To show that the algorithm is correct we prove that it computes a valid (γ, ℓ, L)-decomposition by showing that it always maintains a valid partial (γ, ℓ, L)-decomposition.

▶ **Lemma 12.** *After every iteration i the partial assignment of nodes to layers at time t forms a partial $(\gamma, \ell, i + 1)$-decomposition. Also at the end we get a (γ, ℓ, L)-decomposition with $L \in O(\log n)$.*

Proof. We prove the lemma by induction on the current iteration i. Everything except the promotion of a node is trivial, so for this case we show that all three properties of Definition 4 continue to hold. The nodes in the promoted path of length $b - 1$ (without v^*) are now put into a compress layer. Since the path has length $b - 1$ and does not intersect with any other compress layer, we guarantee that Item 1 still holds for all compress layers. Item 2 and Item 3 still hold since we only take away from old rake layers and hence cannot invalidate these properties. Since we satisfy Lemma 11 every iteration we need only $L \in O(\log n)$ iterations. ◀

In the next section our main goal is to prove that enough of these local maxima actually exist and are nicely distributed in the graph.

3.2 Local Maxima and Bounding Quality

We will first see that during the execution of our algorithm we can actually decompose the graph into a bunch of subtrees as seen in Figure 2. Consider iteration i and the corresponding set of still free nodes $G^{(i)}$. We now give one of the most important definitions, that of a subtree of assigned nodes (refer to Figure 2 to get an intuition).

▶ **Definition 13** (subtree of assigned nodes). *For any node $v \in V(G)$ in any iteration i, the subtree of assigned nodes $T^{(i)}(v)$ denotes the set that contains v and all nodes w that can be reached from v via a path $(v = v_0, v_1, \ldots, v_j = w)$ such that edge $\{v_{a-1}, v_a\}$ is oriented from v_{a-1} to v_a, for each $1 \le a \le j$. Additionally $h^{(i)}(v) = \max\{\text{dist}(v, u)\}_{u \in T^{(i)}(v)}$ is the height of the tree.*

We now express the entire input graph G in terms of these trees. Notice that any node must either be an orphan or be in the subtree of assigned nodes of some orphan. As a result the union over all trees of all orphans will cover the entire tree. We define sets $N^{(i)}$ which contain all of the orphans that are created during our compress procedures. For every path that gets compressed (i.e., not including promoted compress paths) up until the end of iteration i, we add to $N^{(i)}$ all nodes in the path except the oriented parts at the beginning and the end. The sets are chosen exactly such that all orphans are in one or the other set, so we obtain the following.

▶ **Lemma 14.** *For each positive integer i,the following holds:*

$$V(G) = \left(\bigcup_{v \in G^{(i)}} T^{(i)}(v) \right) \cup \left(\bigcup_{v \in N^{(i)}} T^{(i)}(v) \right)$$

Furthermore the two big unions are disjoint.

Notice that for each of these subtrees of assigned nodes, the quality of the root counts exactly how many nodes in this tree still need to terminate. Notice however that all nodes in $N^{(i)}$ are either local maxima or between two local maxima and they can thus already terminate. So by giving a good upper bound on the quality of nodes in $G^{(i)}$, we will show that in each iteration a constant fraction of the remaining nodes terminate.

Creating Local maxima. We next narrow down our view to some concrete iteration i and will hence drop some of the (i) in the exponents. Just from the design of the algorithm we get that if the if-statement in Algorithm 1 is true and we promote v^*, v^* will indeed be a local maximum.

▶ **Lemma 15.** *After v^* is promoted, v^* will be a local maximum.*

Now we will see that if we do promote v^* to become a local maximum, we get that $q(v^*)$ will be a large part of $q(r)$, thereby showing that with each promotion we reduce the quality by a constant fraction.

▶ **Lemma 16.** *If v^* is promoted, then $q(v^*) \geq \frac{q(r)}{2\Delta^b}$.*

Proof. The statement follows from the fact that $q(r) \leq \Delta^b + \sum_{v \in C_b(r)} q(v)$. This is true, because we can separate $H(r)$ into the nodes that are close and those that are far. Concretely a node u is close, if the path to r is strictly less than b. But only Δ^b such nodes can exist. A node u is far, if the path to r is at least b nodes long, at which point it has to pass through one of the nodes $v \in C_b(r)$. Now since $u \in H(r)$ the unique path from r to u satisfies the criteria for u to be in $H(r)$, then the subpath from w to u must also satisfy these criteria and u is therefore included in $q(w)$. We then get

$$q(r) \leq \Delta^b + \sum_{v \in C_b(r)} q(v) \leq \Delta^b + |C_b(r)| \cdot q(v^*) \leq \Delta^b + \Delta^b q(v^*) \leq 2\Delta^b q(v^*).$$

As a result $u \in H(v)$ and therefore u is accounted for in $q(v)$. The second inequality comes from the fact v^* has the highest quality among nodes in $C_b(r)$. ◀

However the if-condition may not hold in every iteration, but this then means that there is a compress path within distance b from r. This compress path cannot be from the normal compress procedure, because of the γ nodes of slack we leave at the ends of every path. As a consequence this path is due to a previous promotion; let x be that promoted node. This allows us to prove the following.

▶ **Lemma 17.** *If the if-condition does not hold, then there exists a promoted node x at distance at most $2b-1$ from r, such that $q(x) \geq \frac{q(v^*)}{2\Delta^b}$ immediately before x was promoted.*

Upperbounding the quality of a free node. The next lemma will be the main technical result that shows that enough nodes are in subtrees of local maxima. We will show this implicitly by upperbounding the quality of the remaining free nodes. However we will have to introduce some more notation. We are partitioning each assigned subtree $T^{(i)}(v)$ into $\alpha = \left\lceil \frac{h^{(i)}(v)+1}{b} \right\rceil$ subsets $S_0^{(i)}(v), \ldots, S_{\alpha-1}^{(i)}(v)$, where, for each $0 \leq j \leq \alpha - 1$,

$$S_j^{(i)}(v) := \{u \in T^{(i)}(v) \mid b \cdot j \leq \operatorname{dist}(u,v) < b \cdot (j+1)\}.$$

Note that we have $\bigcup_{0 \leq j \leq \alpha-1} S_j^{(i)}(v) = T^{(i)}(v)$.

▶ **Lemma 18.** *There exists a constant $0 < \lambda < 1$ (that only depends on Π and Δ) such that for all iterations i, the following inequality holds at the end of iteration i, for all nodes $r \in G^{(i)}$:*

$$q^{(i)}(r) \leq \sum_{j=0}^{\lceil (h^{(i)}(r)+1)/b \rceil - 1} \lambda^j |S_j^i(r)|$$

Intuitively, the deeper in the subtree the nodes are, the more of them are already fixed. Since most of the nodes get removed in the first few iterations, this suffices. The full proof can be found in the full version [1]. Glossing over a lot of details, the main idea is to use induction over the height of the tree and use the induction hypothesis on all of the nodes in $C_b(r)$ to obtain an initial bound on the quality of r. Then we note that the promotion of v^* is not yet taken into account in this bound and we get the desired result. If v^* cannot be promoted, by Lemma 17 some node x was recently promoted. A careful analysis shows that also x was not yet taken into account in the initial bound and we get the desired results.

3.3 Distributed Algorithm and Node Averaged Complexity

In this section, we will describe how we implement Algorithm 1 in a distributed manner and how we will use it to design an algorithm $\mathcal{A}$ that solves the given LCL problem Π, and we will prove an upper bound of $O(\log^* n)$ for the node-averaged complexity of the latter algorithm. We start by describing our distributed implementation of Algorithm 1. For the remainder of the section, set $s := 10\ell$.

Distributed implementation. The computation of ℓ from Π can be performed by every node without any communication. Next, the nodes compute a distance-s coloring with a constant number of colors. Since Δ and ℓ are constant, this can be done in worst-case complexity $O(\log^* n)$, e.g., by using an algorithm of [8]. Using this coloring we can execute compress operations in a constant number of rounds, by simply iterating through the color classes. The γ rakes can trivially be implemented in γ rounds. the promotion requires only to see up to a constant distance, as long as the qualities of all nodes are known. However these can be computed on the fly during the algorithm.

▶ **Lemma 19.** *Assume a distance-s coloring with a constant number of colors is given. Then iteration i of Algorithm 1 can be executed in a constant number of rounds, for each even positive integer i.*

Next we describe our algorithm $\mathcal{A}$ for solving a given LCL problem Π.

Algorithm for Π. Algorithm $\mathcal{A}$ proceeds as follows. Use Algorithm 1 to compute a (γ, ℓ, L)-decomposition, where the values of γ and ℓ depend on Π, and $L \in O(\log n)$ (due to Lemma 12). As soon as a node becomes a local maximum it starts to execute the steps from the original generic algorithm to compute its class, pick an output and start propagating downwards. As a result the entire subtree of assigned nodes of such a local maximum will terminate. The only thing needed for this are the label sets, but they can be propagated upwards during the execution of the decomposition algorithm with no additional cost. As a result we obtain the following lemma.

▶ **Lemma 20.** *Assume a distance-s coloring with a constant number of colors is given. Then there exists an integer constant t such that the following holds: if a node v becomes a local maximum in iteration i of Algorithm 1, then the entire tree $T^{(i)}(v)$ will have terminated after ti rounds in $\mathcal{A}$.*

To make the runtime analysis a bit cleaner, we are going to mark all nodes in $T^{(i)}(v)$, once v becomes a local maximum. We emphasize that this is solely for the purpose of the analysis and this does not change the algorithm at all. More specifically, once any node v becomes a local maximum, all of the nodes in $T^{(i)}(v)$ become marked instantly (in 0 rounds). We obtain the following corollary from Lemma 20.

▶ **Corollary 21.** *Assume a distance-s coloring with a constant number of colors is given. If a node v becomes marked in iteration i, then v will have terminated in round ti, where t is the constant from Lemma 20.*

Consider some iteration i, by applying Lemma 18 on every remaining free node in $G^{(i)}$ we can upperbound the total quality and therefore the total number of unmarked nodes. For the full proof refer to the full version of the paper [1].

▶ **Lemma 22.** *There exists a constant $0 < \sigma < 1$, such that for every iteration $i \geq 10$ of Algorithm 1 at most $2\Delta^b n\sigma^i$ nodes are not marked.*

We obtain the following lemma.

▶ **Lemma 23.** *On average, nodes become marked in $O(1)$ iterations.*

Then using this lemma together with Corollary 21 we get that an average node terminates after a constant number of rounds. However, we still have to pay for the input distance coloring which takes $O(\log^* n)$, as discussed in the beginning of the section. So by first computing this input coloring and then running the algorithm, we obtain a total node-averaged complexity of $O(\log^* n)$, proving Theorem 1.

References

1 Alkida Balliu, Sebastian Brandt, Fabian Kuhn, Dennis Olivetti, and Gustav Schmid. On the node-averaged complexity of locally checkable problems on trees. *CoRR*, abs/2308.04251, 2023. doi:10.48550/arXiv.2308.04251.

2 Alkida Balliu, Sebastian Brandt, Dennis Olivetti, and Jukka Suomela. How much does randomness help with locally checkable problems? In *Proc. 39th ACM Symposium on Principles of Distributed Computing (PODC 2020)*, pages 299–308. ACM Press, 2020. doi:10.1145/3382734.3405715.

3 Alkida Balliu, Sebastian Brandt, Dennis Olivetti, and Jukka Suomela. Almost global problems in the LOCAL model. *Distributed Comput.*, 34(4):259–281, 2021. doi:10.1007/s00446-020-00375-2.

4 Alkida Balliu, Keren Censor-Hillel, Yannic Maus, Dennis Olivetti, and Jukka Suomela. Locally checkable labelings with small messages. In *35th International Symposium on Distributed Computing, DISC 2021*, pages 8:1–8:18, 2021. doi:10.4230/LIPIcs.DISC.2021.8.

5 Alkida Balliu, Mohsen Ghaffari, Fabian Kuhn, and Dennis Olivetti. Node and edge averaged complexities of local graph problems. In *Proc. 41st ACM Symp. on Principles of Distributed Computing (PODC)*, pages 4–14, 2022.

6 Alkida Balliu, Juho Hirvonen, Janne H. Korhonen, Tuomo Lempiäinen, Dennis Olivetti, and Jukka Suomela. New classes of distributed time complexity. In *Proc. 50th ACM Symposium on Theory of Computing (STOC 2018)*, pages 1307–1318. ACM Press, 2018. doi:10.1145/3188745.3188860.

7 Alkida Balliu, Juho Hirvonen, Dennis Olivetti, and Jukka Suomela. Hardness of minimal symmetry breaking in distributed computing. In *Proc. 38th ACM Symposium on Principles of Distributed Computing (PODC 2019)*, pages 369–378. ACM Press, 2019. doi:10.1145/3293611.3331605.

8 Leonid Barenboim, Michael Elkin, and Fabian Kuhn. Distributed $(\Delta + 1)$-coloring in linear (in Δ) time. *SIAM J. on Computing*, 43(1):72–95, 2015.

9 Leonid Barenboim and Yaniv Tzur. Distributed symmetry-breaking with improved vertex-averaged complexity. In *Proc. 20th Int. Conf. on Distributed Computing and Networking (ICDCN)*, pages 31–40, 2019.

10 Yi-Jun Chang. The complexity landscape of distributed locally checkable problems on trees. In *Proc. 34th International Symposium on Distributed Computing (DISC 2020)*, volume 179 of *LIPIcs*, pages 18:1–18:17. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.DISC.2020.18.

11 Yi-Jun Chang, Qizheng He, Wenzheng Li, Seth Pettie, and Jara Uitto. Distributed edge coloring and a special case of the constructive lovász local lemma. *ACM Trans. Algorithms*, 16(1):8:1–8:51, 2020.

12 Yi-Jun Chang, Tsvi Kopelowitz, and Seth Pettie. An exponential separation between randomized and deterministic complexity in the LOCAL model. *SIAM J. Comput.*, 48(1):122–143, 2019. doi:10.1137/17M1117537.

13 Yi-Jun Chang and Seth Pettie. A time hierarchy theorem for the LOCAL model. *SIAM J. Comput.*, 48(1):33–69, 2019. doi:10.1137/17M1157957.

14 Soumyottam Chatterjee, Robert Gmyr, and Gopal Pandurangan. Sleeping is efficient: MIS in $O(1)$-rounds node-averaged awake complexity. In *Proc. 39th ACM Symp. on on Principles of Distributed Computing (PODC)*, pages 99–108, 2020.

15 Laurent Feuilloley. How long it takes for an ordinary node with an ordinary ID to output? In *Proc. 24th Int. Coll. on Structural Information and Communication Complexity (SIROCCO)*, volume 10641, pages 263–282, 2017.

16 Manuela Fischer and Mohsen Ghaffari. Sublogarithmic distributed algorithms for Lovász local lemma, and the complexity hierarchy. In *Proc. 31st International Symposium on Distributed Computing (DISC 2017)*, volume 91 of *LIPIcs*, pages 18:1–18:16, 2017. doi:10.4230/LIPIcs.DISC.2017.18.

17 Christoph Grunau, Václav Rozhon, and Sebastian Brandt. The landscape of distributed complexities on trees and beyond. In *Proc. 41st ACM Symposium on Principles of Distributed Computing (PODC 2022)*, pages 37–47, 2022. doi:10.1145/3519270.3538452.

18 Öjvind Johansson. Simple distributed $Delta$+1-coloring of graphs. *Inf. Process. Lett.*, 70(5):229–232, 1999.

19 Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. Local computation: Lower and upper bounds. *J. ACM*, 63(2):17:1–17:44, 2016.

20 Nathan Linial. Locality in distributed graph algorithms. *SIAM J. Comput.*, 21(1):193–201, 1992.

21 Gary L. Miller and John H. Reif. Parallel tree contraction and its application. In *26th Annual Symposium on Foundations of Computer Science, Portland, Oregon, USA, 21-23 October 1985*, pages 478–489. IEEE Computer Society, 1985. doi:10.1109/SFCS.1985.43.

22 Moni Naor and Larry J. Stockmeyer. What can be computed locally? *SIAM J. Comput.*, 24(6):1259–1277, 1995. doi:10.1137/S0097539793254571.

23 Václav Rozhoň and Mohsen Ghaffari. Polylogarithmic-time deterministic network decomposition and distributed derandomization. In *Proc. 52nd ACM Symp. on Theory of Computing (STOC)*, pages 350–363, 2020.

A Improved Algorithms in the Polynomial Regime

In this section we show that, for infinitely many LCL problems with polynomial worst-case complexity, we can improve their node-averaged complexity. More precisely, in this section we show that, for a class of problems with worst-case complexity $\Theta(n^{1/k})$, we can provide an algorithm with node-averaged complexity $O(n^{1/(2^k-1)})$. As we show in Appendix B, this complexity is almost tight, since for all problems with worst-case complexity $\Theta(n^{1/k})$ we can show a lower bound of $\widetilde{\Omega}(n^{1/(2^k-1)})$.

The Hierarchical $2\frac{1}{2}$-Coloring Problems

We now define a class of problems, already presented in [13], called hierarchical $2\frac{1}{2}$-coloring, that is parametrized by an integer $k \in \mathbb{Z}^+$. It has been shown in [13] that the problem with parameter k has worst-case complexity $\Theta(n^{1/k})$. We now give a formal definition of this class of problems.

The set of input labels is $\Sigma_{\text{in}} = \emptyset$. The set of output labels contains four possible labels, that is, $\Sigma_{\text{out}} = \{W, B, E, D\}$, and these labels stand for *white*, *black*, *exempt*, and *decline*. Each node has a level in $\{1, \dots, k+1\}$, that can be computed in constant time, and the constraints of the nodes depend on the level that they have. The level of a node is computed as follows.

1. Let $i \leftarrow 1$.
2. Let V_i be the set of nodes of degree at most 2 in the remaining tree. Nodes in V_i are of level i. Nodes in V_i are removed from the tree.
3. Let $i \leftarrow i+1$. If $i \le k$, continue from step 2.
4. Remaining nodes are of level $k+1$.

Each node must output a single label in Σ_{out}, and based on their level, they must satisfy the following local constraints.

- No node of level 1 can be labeled E.
- All nodes of level $k+1$ must be labeled E.
- Any node of level $2 \le i \le k$ is labeled E iff it is adjacent to a lower level node labeled W, B, or E.
- Any node of level $1 \le i \le k$ that is labeled W (resp. B) has no neighbors of level i labeled B (resp. W) or D. In other words, W and B are colors, and nodes of the same color cannot be neighbors in the same level.
- Nodes of level k cannot be labeled D.

This problem can be expressed as a standard LCL by setting the checkability radius r to be $O(k)$, since in $O(k)$ rounds a node can determine its level and hence which constraints apply. In Section 1.2 we provided some intuition about these problems. We provide more intuition in the full version of the paper [1].

Better Node-Averaged Complexity

We now show that, for this class of LCL problems, we can obtain a better node-averaged complexity. The algorithm is similar to the one presented in [10] for the worst-case complexity, but it is modified to obtain a better node-averaged complexity (in Appendix B we show that this algorithm is tight up to a $\log n$ factor).

▶ **Theorem 24.** *The node-averaged complexity of the hierarchical $2\frac{1}{2}$-coloring problem with parameter k is $O(n^{1/(2^k-1)})$.*

Proof. At first, all nodes spend $O(1)$ rounds to compute their level. Nodes of level $k+1$ output E. Then, the algorithm proceeds in phases, for i in $1, \ldots, k$. In phase i, all nodes of level i get a label, and hence let us assume that all nodes of levels $1, \ldots, i-1$ already have a label, and let us focus on level-i nodes.

Consider a node v of level i. Node v proceeds as follows. If v has a neighbor from lower levels that is labeled W or B then v outputs E. Otherwise, v spends $t_i = c \cdot \gamma_i$ rounds to check the length of the path containing v induced by nodes of level i, for some constant c to be fixed later, and $\gamma_i = n^{2^{i-1}/(2^k-1)}$. If this length is strictly larger than t_i, then v outputs D. Otherwise, all nodes of the path are able to see the whole path, and hence they can output a consistent 2-coloring by using the labels W and B.

The above algorithm correctly solves the problem if we assume that no nodes in level k output D. In the full version of the paper [1] we show that indeed nodes of level k do not output D, hence showing the correctness of the algorithm, and then we prove a bound on the node-averaged complexity. In the following we sketch the main idea for the correctness of the algorithm.

Let S be the set of nodes of level i that do not directly output E at the beginning of phase i. It is possible to assign each node of level $j < i$ to exactly one node in S such that to each node in S are assigned $\Omega(n^{(2^{i-1}-1)/(2^k-1)})$ unique nodes of lower layers. Hence, the number of nodes that participate in phase i is at most $O(n^{1-(2^{i-1}-1)/(2^k-1)}) = O(n^{(2^k-2^{i-1})/(2^k-1)})$. This implies that in phase k the number of participating nodes is at most $O(n^{(2^k-2^{k-1})/(2^k-1)}) = O(n^{2^{k-1}/(2^k-1)}) = O(\gamma_k)$, where the hidden constant is inversely proportional to c. Hence, by picking c large enough, we get that in t_k rounds nodes of level k see the whole path and thus no node of level k outputs D, proving the correctness of the algorithm. ◀

B Lower Bounds in the Polynomial Regime

In this section we show that any LCL problem that requires polynomial time for worst-case complexity requires polynomial time also for node-averaged complexity. More precisely, we prove the following theorem.

▶ **Theorem 25.** *Let Π be an LCL problem with worst-case complexity $\Omega(n^{1/k})$ in the* LOCAL *model. The node-averaged complexity of Π in the* LOCAL *model is $\Omega(n^{1/(2^k-1)}/\log n)$.*

In order to prove this theorem, we proceed as follows (throughout this section we will use notions presented in Section 2.1). It is known by [10] that if an LCL problem Π has worst-case complexity $o(n^{1/k})$, then it can actually be solved in $O(n^{1/(k+1)})$ rounds. This statement is proved by showing that it is possible to use an algorithm (possibly randomized) running in $o(n^{1/k})$ rounds to construct a function $f_{\Pi,k+1}$ that can be used to map the label sets assigned to the edges connected to a compress path from lower layers into two label sets for the edges connecting the endpoints of the path to their parents, in such a way that, if the compress layers are at most $k+1$, then the algorithm sketched in Section 2.1 works. This implies, as shown in Section 2.1, the existence of a deterministic algorithm that solves Π and has worst-case complexity $O(n^{1/(k+1)})$. In this section we show that it is possible to construct a function $f_{\Pi,k+1}$ by starting from an algorithm $\mathcal{A}$ with node-averaged complexity $o(n^{1/(2^k-1)}/\log n)$. By Section 2.1, this implies that if there exists an algorithm with $o(n^{1/(2^k-1)}/\log n)$ node-averaged complexity, then there exists an algorithm with worst-case complexity $O(n^{1/(k+1)})$, implying that any LCL with worst-case complexity $\Omega(n^{1/k})$ has node-averaged complexity at least $\Omega(n^{1/(2^k-1)}/\log n)$. While we will use some ideas already presented in [10], handling an algorithm with only guarantees on its node-averaged complexity arises many (new) issues that we need to tackle.

Our statement will be proved even for the case in which algorithm $\mathcal{A}$ satisfies the weakest possible assumptions (i.e., the assumptions are so relaxed that they are satisfied by any deterministic algorithm, any randomized Las Vegas algorithm, and any randomized Monte Carlo algorithm). The assumptions are the following.

- We assume that $\mathcal{A}$ is a randomized algorithm that is only required to work when the unique IDs of nodes are assigned at random, among all possible valid assignments.
- We assume that $\mathcal{A}$ is an algorithm that fails with probability at most $1/n^c$ for any chosen constant $c \geq 1$.
- We assume that the bound on the node-averaged complexity of $\mathcal{A}$ holds with probability at least $1 - 1/n^c$ for any chosen constant $c \geq 1$.

However, in the following, we will assume that the bound on the node-averaged complexity of $\mathcal{A}$ holds always. In fact, observe that we can always convert an algorithm with node-averaged complexity T that holds with probability at least $1 - 1/n^c$ into an algorithm with node-averaged complexity $O(T)$ that holds always, since, even for $c = 1$, we can safely assume that when the bound does not hold (that happens with probability at most $1/n$), the runtime is anyways bounded by n (since everything can be solved in n rounds in the LOCAL model).

Proof Overview

We now sketch the high-level ideas for proving that it is possible to use an algorithm with $o(n^{1/(2^k-1)}/\log n)$ node-averaged complexity to construct a function $f_{\Pi,k+1}$. All the details are deferred to the full version of the paper [1], where, for completeness, we also prove a result already presented in [13, 10]: Whether a function $f_{\Pi,k+1}$ exists can be mechanically determined.

Recall that the input of the function $f_{\Pi,k+1}$ is a path, where to each node are connected some edges (that we call incoming edges), and for each of them is provided a set of labels. Additionally, to each endpoint of the path is connected an additional edge (called outgoing edge). We need to compute two sets of labels, L_1 and L_2, one for each outgoing edge. These sets must satisfy that, for any choice of labels $(\ell_1, \ell_2) \in L_1 \times L_2$, we can pick a label from the sets assigned to the incoming edges, and a label for each edge of the path, that results in a labeling that satisfies the constraints of the problem. Observe that there may exist multiple pairs of non-empty sets satisfying the requirements, but we need a specific type of solution: once such sets are propagated to higher layers, we still want to be able to perform the same operation in the next compress layer. A more general property that needs to be satisfied is that empty sets are never obtained, because this would prevent nodes from being able to pick a valid labeling in the top-down phase. In [13, 10] it is shown how to construct such a function, by starting from an algorithm with (possibly randomized) worst-case complexity $o(n^{1/k})$. On a high level, this is achieved as follows:

1. Replace each incoming edge of the path with a small tree, satisfying that the class of the tree corresponds to the label set of the edge.
2. Modify the path, by making it *much longer*, but by preserving its completability properties, that is, a choice $(\ell_1, \ell_2) \in L_1 \times L_2$ is good for the long version of the path if and only if it is good for the original version.
3. Ask the algorithm what it would output in the middle of such a long path. Crucially, the path is made so long that the algorithm, within its running time, cannot see the endpoints.
4. Compute what labels can be put on the outgoing edges of the long path, in a way that the labeling in the rest of the path can be completed in a valid way (and it is compatible with the label sets of the incoming edges), and such that the output in the middle of the path corresponds to the output of the algorithm. Since the output in the middle of the path is fixed, then the outputs on the outgoing edges can be chosen *independently*. Hence, in this way, we obtain an independent class for the path.

In [13, 10] it is argued why, by using such a function with the algorithm sketched in Section 2.1, empty label sets are never obtained. While we need to adapt such a proof to the case of node-averaged complexity, there are some additional challenges that we need to tackle. For example, one issue that we have when trying to adapt this approach to the case of node-averaged complexity is that an algorithm could run for a long time at *some nodes*, while still keeping a low node-averaged complexity. Hence, we do not have the guarantee that, if we make a path long, then the algorithm does not see the endpoints. In the full version of the paper we show that, in the instances that we need to handle in order to construct a valid function, we can prove that a stronger notion of node-averaged complexity must hold, namely that the expected running time of each node is bounded. Then, by considering $\Theta(\log n)$ sufficiently separated nodes of a long path, we can prove that, if we consider a node in the middle of the path and an endpoint, with high probability they are not able to communicate, and we hence obtain a result similar to the case of worst-case complexity. Moreover, the bound on the expected runtime will depend on the layer number, and this bound is what governs the final lower bound complexity.

Treasure Hunt with Volatile Pheromones

Evangelos Bampas
Université Paris-Saclay, CNRS, Laboratoire Interdisciplinaire des Sciences du Numérique, 91400 Orsay, France

Joffroy Beauquier
Université Paris-Saclay, CNRS, Laboratoire Interdisciplinaire des Sciences du Numérique, 91400 Orsay, France

Janna Burman
Université Paris-Saclay, CNRS, Laboratoire Interdisciplinaire des Sciences du Numérique, 91400 Orsay, France

William Guy–Obé
Université Paris-Saclay, CNRS, Laboratoire Interdisciplinaire des Sciences du Numérique, 91400 Orsay, France

Abstract

In the treasure hunt problem, a team of mobile agents need to locate a single treasure that is hidden in their environment. We consider the problem in the discrete setting of an oriented infinite rectangular grid, where agents are modeled as synchronous identical deterministic time-limited finite-state automata, originating at a rate of one agent per round from the origin. Agents perish τ rounds after their creation, where $\tau \geq 1$ is a parameter of the model. An algorithm solves the treasure hunt problem if every grid position at distance τ or less from the origin is visited by at least one agent. Agents may communicate only by leaving indistinguishable traces (pheromone) on the nodes of the grid, which can be sensed by agents in adjacent nodes and thus modify their behavior. The novelty of our approach is that, in contrast to existing literature that uses permanent pheromone markers, we assume that pheromone traces evaporate over μ rounds from the moment they were placed on a node, where $\mu \geq 1$ is another parameter of the model. We look for uniform algorithms that solve the problem without knowledge of the parameter values, and we investigate the implications of this very weak communication mechanism to the treasure hunt problem. We show that, if pheromone persists for at least two rounds ($\mu \geq 2$), then there exists a treasure hunt algorithm for all values of agent lifetime. We also develop a more sophisticated algorithm that works for all values of μ, hence also for the fastest possible pheromone evaporation of $\mu = 1$, but only if agent lifetime is at least 16.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Theory of computation → Design and analysis of algorithms

Keywords and phrases Mobile Agents, Exploration, Search, Treasure Hunt, Pheromone, Evaporation

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.8

Related Version *Full Version*: `https://hal.science/hal-04177364v1`

Acknowledgements We thank the DISC 2023 anonymous reviewers for their careful reading and comments.

1 Introduction

Treasure hunt is the fundamental problem of employing a team of searchers to locate a "treasure" that is hidden somewhere in their environment. It is one of the fundamental primitives in swarm robotics and a natural abstraction of foraging behavior of animals. Although various formulations of the problem exist at least since the 1960s, when Beck introduced the linear search problem [14], treasure hunt as a group search problem was first investigated from a distributed algorithms perspective by Feinerman et al. [41, 42, 43], under the name ANTS (Ants Nearby Treasure Search). In the ANTS problem, the search is performed by a team of randomized searchers, starting at the origin of an infinite 2-dimensional rectangular grid and having no means of communication once they start moving. Subsequent works considered stronger communication models, such as local communication by exchanging constant-size messages when two agents are located on the same node [40, 39, 25, 21, 54, 53], or communication by leaving permanent markers on grid nodes [56, 1, 2], that can be detected by other agents.

In this paper, we introduce a new model in which not only agents communicate indirectly, by dropping and sensing markers on nodes, but also these markers gradually evaporate and eventually disappear. This is directly inspired by the behavior of actual pheromone trails in nature. A common feature of the papers that we mentioned above is that the team of searchers is of constant size. However, with evaporating pheromones, we can no longer expect a constant-size team of constant-memory agents to explore all the grid nodes up to arbitrary distances.[1] Therefore, we propose a new model taking into account pheromone evaporation, in which a potentially infinite number of identical, synchronous, deterministic, time-limited finite-state automata are created at a rate of one agent per round at the origin of a 2-dimensional grid. Agents have a finite lifetime represented by the parameter τ, and the treasure is guaranteed to be within reach, i.e., at distance $\leq \tau$ from the origin. Pheromone evaporation is controlled by a parameter μ, which determines the number of rounds it takes for a pheromone marker to disappear from the system, assuming it is not refreshed in the meantime by a new pheromone drop on the same node. Agents can sense the presence or absence of pheromone in their neighboring nodes, and they can compare pheromone values, i.e., they know, for any pair of directions, which neighbor has the freshest pheromone. Agent memory cannot depend on the parameters τ, μ.

1.1 Related work

Searching is a well-studied family of problems in which a group consisting of one or multiple searchers (mobile agents) need to find a target placed at some unknown location. The search is typically concluded when the first searcher finds the target. Numerous books and research papers have been written on this subject, studying diverse models involving stationary or mobile targets, graphs or geometric terrains, different types of knowledge about the environment, one or many searchers, etc. [5, 6, 17, 23, 45, 47, 58].

Deterministic search on a line with a single robot was introduced in [14, 15]. In the original formulation of [14], a probability distribution of treasure placements is known to the agent. An optimal algorithm with competitive ratio 9, for an unknown probability distribution, is proposed in [15]. The problem is further generalized in [8, 35], by introducing a cost for turning, as well as a more general star topology. Further variants include searching for multiple targets [9], maximizing the searched area with a given time budget [10], and providing a hint to the searcher before it starts exploring [7].

[1] Indeed, intuitively, if they find themselves sufficiently far from each other, then they can no longer communicate because pheromone will evaporate before it can be sensed by another agent, whereas if they never find themselves more than a constant distance from each other, then their overall behavior can be described by a single finite automaton, which fails to explore a sufficiently large grid due to state repetition that forces it to explore at most a constant-width half-line (see, e.g., [39, Lemma 5]).

Various maintenance and patrolling problems have also been formulated as linear group search problems, under requirements and assumptions such as perpetual exploration [27, 50, 26] or distinct searcher speeds [29, 50, 13]. The closely related evacuation problem on the line, in which the search is concluded when *all* searchers reach the target, has also been studied in a series of papers [11, 22, 12, 20]. See also [30] for a survey of group search and evacuation in different domains.

Searching with advice (hints) is studied under various assumptions in several papers. The size of advice that must be provided to a lone deterministic searcher in a polygonal terrain with polygonal obstacles, in order to locate the treasure at a cost linear in the length of a shortest path from the initial position of the agent to the treasure, is investigated in [59]. An algorithm that enables a deterministic agent to find an inert treasure in the Euclidean plane, taking advantage of hints about the general direction of the treasure, is given in [18]. In [51, 57], they explore the tradeoff between advice size and search cost in graphs. In trees, [19] explores the impact of different kinds of initial knowledge given to a lone searcher on the time cost of treasure hunt, and [16] considers treasure hunt with faulty hints.

The speedup in search time obtained by multiple independent random walkers has been studied for various graph families, such as expanders and random graphs [37, 4, 38, 48, 28]. Multiple searchers following Lévy walk processes, a type of random walk in which jump lengths are drawn from a power-law distribution, and for which there is significant empirical evidence that it models the movement patterns of various animal species [31], are investigated in [24]. A further abstraction of multiple independent randomized searchers is studied in [46], where a group of non-communicating agents need to find an adversarially placed treasure, hidden in one of an infinite set of boxes indexed by the natural numbers. In this Bayesian search setting, searchers have random access to the boxes. A game-theoretic perspective to the Bayesian search framework of [46] is given in [52].

The ANTS (Ants Nearby Treasure Search) problem was introduced in [41, 42, 43] as a natural abstraction of foraging behavior of ants around their nest. They explore the tradeoff between searcher memory and the speedup obtained by using multiple probabilistic searchers vs using a single searcher. Searchers may not communicate once they leave the nest. A variant of the ANTS problem in the geometric plane, with searchers that are susceptible to crash faults, is investigated in [3]. A notion of selection complexity, which measures how likely a given ANTS algorithm is to arise in nature, is introduced in [55], where they study the tradeoff between selection complexity and speedup in search time.

In follow-up work [40, 39, 25, 21, 54, 53] to the original ANTS formulation, searchers are modeled as finite state machines and can communicate outside the nest, when they are sufficiently close to each other, by exchanging messages of constant size. Under these assumptions, it is shown in [40] that the optimal search time can still be achieved by probabilistic finite state machines, matching the lower bound of [42]. The minimum number of searchers that can solve the ANTS problem, when they are controlled by randomized/deterministic finite/push-down automata, is investigated in [39, 25, 21]. A probabilistic fault-tolerant constant-memory algorithm is presented in [54], for the synchronous case. An algorithm that tolerates obstacles is presented in [53].

A different communication mechanism is considered in [56, 1, 2], where it is assumed that agents may communicate only by leaving permanent markers (pheromones) on nodes, which can be sensed later by other agents. Note that, although these papers use the word "pheromone" to describe the traces that agents leave on nodes, these are assumed permanent and do not evaporate. The usual term in the mobile agent literature to describe this type of movable or immovable marker that agents may choose to leave on nodes, and which can be detected later by other agents, is "token" or "pebble" [34].

1.2 Our contributions

We study the treasure hunt problem in the model that we outlined above, and which is developed in detail in Section 2. Thematically, our work is closest to the literature descending from the original ANTS problem formulation, and in particular to these papers that use pheromone (or tokens) as a means of communication [56, 1, 2]. The novelty of our approach is that we use *evaporating* pheromones as an agent communication mechanism. Indeed, in our model, a pheromone trace disappears μ rounds after it was dropped, unless it is refreshed by a new pheromone drop on the same node. Tokens that may disappear instantly from the system have actually been considered before in the literature, but only in the context of faults [44, 33, 32, 36].

To our knowledge, evaporating pheromone markers have never been considered before as a communication mechanism, from an algorithmic point of view. We study the impact of this weak agent communication model on the treasure hunt problem.

Our first result is a treasure hunt algorithm that works for all $\tau \geq 1$, assuming that the pheromone markers persist for at least two rounds ($\mu \geq 2$). This algorithm is optimal in terms of search time, number of pheromone drops, and number of agents used. Intuitively, the algorithm dispatches agents to the North and to the South of the origin by means of pheromone patterns around the nest. An agent knows when to leave the vertical axis in order to explore a horizontal half-line by detecting pheromone markers that were dropped by previous agents when they left the vertical axis. Because of the North-South dispatching at the origin, successive agents on the same side of the origin are at distance 2 from each other, therefore it is crucial that $\mu \geq 2$ for an agent to be able to detect pheromone that was dropped by the previous agent.

Our second result is a more complex algorithm that works for all $\mu \geq 1$. This algorithm is also based on a North-South dispatching of agents at the origin. The challenge here is that, since pheromone may be detectable for only one round after being dropped, we can no longer use the same approach as in the first algorithm. We resolve this by introducing *differentiation* of agent roles as a result of observing different pheromone patterns as they walk along the vertical axis. Now, some agents become *signaling* agents that stop moving at key positions and start dropping pheromone according to a predetermined pattern, whereas other agents become *explorers* that are dispatched to different horizontal half-lines according to these signals. This algorithm works for all $\tau \geq 16$.

Both algorithms are deterministic and uniform, i.e., they do not assume knowledge of the values of the parameters τ, μ. They solve the problem for *all* parameter values in the specified ranges, and the required memory per agent is constant.

Some proofs have been omitted due to lack of space, and they can be found in the full version of the paper.

2 Model and problem setting

The environment in which the agents operate is an infinite two-dimensional rectangular grid graph, equipped with a Cartesian coordinate system. Each node of the grid is identified by a pair of integer coordinates $(x, y) \in \mathbb{Z}^2$. The node $(0, 0)$ is called *nest*, as newly created agents appear at $(0, 0)$. We assume that the grid is oriented, with the four outgoing edges from each node receiving globally consistent distinct local port labels from $\{N, E, S, W\}$. Each node stores a nonnegative integer that represents the amount of pheromone present on that node. This value is decremented by 1 at each round and a value of zero represents the absence of pheromone.

Given natural numbers a, b, we use the notation $a \dot{-} b$ for proper subtraction: $a \dot{-} b = \max(a - b, 0)$. Moreover, if x is a nonnegative integer, we use $\mathbb{B}_x$ for the set of nodes at distance at most x from the nest, and $\mathbb{L}_x$ for the set of nodes at distance exactly x from the nest. We have $|\mathbb{B}_x| = 2x^2 + 2x + 1$ and $|\mathbb{L}_x| = 4x$.

2.1 Agent model

Agents are modeled as identical copies of a deterministic finite-state machine (FSM). An agent can move from node to node along the edges of the grid graph, and it may decide to drop pheromone before or after each move (but not both on the origin and on the destination node). It computes its next move based on the relative pheromone values of the neighboring nodes. More precisely, the agent does not have access to the actual stored pheromone values, but it can detect the presence or absence of pheromone in any direction, as well as whether one direction has equal, less, or more pheromone than another.

▶ **Definition 1** (Agents). *An agent is a finite-state machine* $\mathcal{A} = (Q, q_0, \mathsf{In}, \mathsf{Out}, \delta)$ *where:*

- Q *is a finite set of states and* $q_0 \in Q$ *is the initial state.*
- In *is the input alphabet. A symbol of* In *encodes the presence or absence of pheromone in the four cardinal directions, as well as the result of the comparison of pheromone levels for any pair of directions. This is clearly a finite amount of information, hence* In *is a finite set.*
- $\mathsf{Out} = \{N, S, E, W, \perp\} \times \{\text{before}, \text{after}, \perp\}$ *is the output alphabet, where the first element of an output symbol is the local port label through which the agent will exit the current node (*$\perp$ *for no movement), and the second element indicates whether pheromone will be dropped before or after the move (*$\perp$ *for no pheromone drop).*
- $\delta : Q \times \mathsf{In} \to Q \times \mathsf{Out}$ *is the transition function.*

▶ Note 2. By definition, an agent does not perceive other agents that may be present on the same node or on neighboring nodes. Moreover, an agent does not perceive and therefore cannot compare the pheromone level of its current node to those of neighboring nodes.

2.2 Model parameters

Agents have limited life, which is a parameter of the model and is represented by a positive integer τ. An agent "dies" upon having performed τ state transitions, meaning that it essentially disappears from the system.[2] We will call this parameter *lifetime*.

We also assume that every node has a maximum amount of pheromone that it can store, which is a second parameter of the model and is represented by a positive integer μ, which we will call *pheromone duration*. Whenever any number of agents decide to drop pheromone on a node at the same time, the pheromone value of that node is updated to μ. If an agent drops pheromone on some node, the pheromone value of that node will decrease from μ to 0 over the following μ rounds (assuming it is not refreshed in the meantime).

2.3 Execution

Given a protocol $\mathcal{A}$ (FSM) and an assignment of values to the parameters (τ, μ), the execution of the system proceeds deterministically in synchronous rounds.

[2] Perhaps less fatally, we may assume that after τ transitions, an agent is so tired that it cannot continue executing the protocol before returning to the nest for a brief nap. It may re-emerge from the nest at a later round without retaining its state.

▶ **Definition 3** (Execution). *The* execution of an FSM $\mathcal{A}$ for parameter values (τ, μ) *is an infinite sequence of system configurations defined as follows: In the initial configuration, there are no agents and no pheromone present on the grid. In each round i ($i \geq 1$), the next configuration is obtained from the current configuration by synchronously executing the following steps in the given order:*

1. *A new agent (copy of $\mathcal{A}$) is created on node $(0, 0)$, in the initial state q_0.*
2. *All agents read their inputs.*
3. *At each node, the quantity of pheromone is decreased by 1, if not already zero (pheromone evaporation).*
4. *All agents compute their transition function based on the input from step 2 and change their state accordingly.*
5. *All agents that computed in step 4 a pheromone drop action "*before*" drop pheromone on their current nodes. For each node on which at least one agent drops pheromone, the pheromone quantity of that node is updated to μ.*
6. *All agent moves (as computed in step 4) are executed.*
7. *All agents that computed in step 4 a pheromone drop action "*after*" drop pheromone on their current nodes. For each node on which at least one agent drops pheromone, the pheromone quantity of that node is updated to μ.*
8. *If this is round $i \geq \tau$, the agent that was created at the beginning of round $i - \tau + 1$ "dies" as it has now performed τ state transitions.*

▶ Note 4. As agents are anonymous and deterministic, and because pheromone does not accumulate higher than μ on a single node, if two (or more) agents ever find themselves at the same node and in the same state, then they will effectively continue moving as one agent from that point on. In particular, agents do not appear simultaneously at the nest, but they are created at a rate of one agent per round.

▶ **Definition 5.** *For $i \geq 1$, we denote by A_i the agent that is created at the beginning of round i.*

2.4 The treasure hunt problem

In the *treasure hunt* problem, a treasure is placed at an unknown location in the grid and the goal is for at least one agent to visit that node. In that case, we say that the agent *locates* the treasure. Locating the treasure for any (unknown) treasure location up to distance d from the nest is trivially equivalent to exploring all nodes up to distance d from the nest. We recast, then, the treasure hunt problem as an exploration problem:

▶ **Definition 6** (Treasure hunt problem). *A given FSM $\mathcal{A}$* solves the treasure hunt problem for the pair of parameters (τ, μ) *if, with lifetime τ and pheromone duration μ, every node at distance τ or less is visited by at least one agent. In this case, we will also say that the FSM is* correct for (τ, μ).

We will seek a uniform algorithm that solves the problem without knowledge of the model parameters, i.e., a single FSM that is correct for arbitrarily large values of τ and μ (ideally, for all $\tau \geq \tau_0$ and $\mu \geq \mu_0$, for some constants τ_0, μ_0).

For a given FSM, we will consider the following measures of efficiency as functions of τ and μ:

- *Completion time*: the number of rounds until the treasure is located.
- *Pheromone utilization*: the total number of times that any agent decides to drop pheromone at its destination node until the treasure is located.

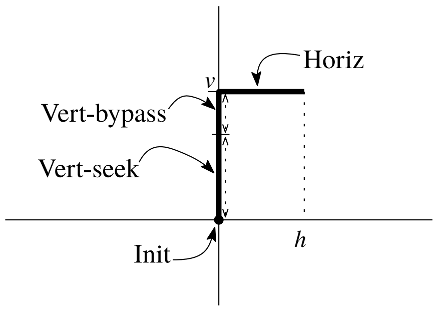


Figure 1 Illustration of the sequence of states for a typical agent with signature $[v; h]$ executing Algorithm 1.

- *Agent utilization*: the number of agents *effectively* used by the algorithm, i.e., the smallest r such that the algorithm remains correct even if the system stops creating new agents after round r.

3 A treasure hunt algorithm for $\tau \geq 1$ and $\mu \geq 2$

We propose a deterministic and uniform algorithm that solves the treasure hunt problem for all combinations of parameters (τ, μ) with $\tau \geq 1$ and $\mu \geq 2$. We give a compact representation of the algorithm as a hybrid state transition diagram in Appendix A, and the full pseudocode in Section 3.1.

Before we give an informal description of the algorithm, we define the notion of *agent signature*:

▶ **Definition 7** (Agent signature). *Let $v, h \in \mathbb{Z}$ with $|v| + |h| = \tau$. We say that an agent has* signature $[v; h]$ *if it starts (from the nest) by moving $|v|$ steps to the North (resp. South), up to node $(0, v)$, if $v \geq 0$ (resp. $v < 0$), followed by $|h|$ steps to the East (resp. West), up to node (h, v), if $h \geq 0$ (resp. $h < 0$).*

The algorithm creates agents of all possible signatures $[v; h]$, thus ensuring correctness by visiting all nodes at distance $\leq \tau$ from the nest. Each agent drops pheromone once upon leaving the nest on its first move, and once more if and when it leaves the vertical axis. Figure 1 shows the sequence of states of a typical agent executing the algorithm.

The first two agents use pheromone information to the East and to the West of the nest to take signatures $[0, -\tau]$ and $[0, \tau]$ (state Init, lines 2-5). Subsequently created agents use pheromone information to the North and to the South of the nest to alternate between the two vertical directions: If there is more pheromone to the North of the nest then they start moving South, otherwise they start moving North (state Init, lines 7-9).

A northbound agent (southbound agents behave symmetrically) starts moving to the North in state Vert-seek. In this state, it checks horizontally adjacent nodes for the presence of pheromone previously dropped by agents leaving the vertical axis. Once it finds such pheromone traces, it switches to state Vert-bypass and keeps moving to the North until it reaches the first node $(0, v)$ whose East and West neighbors do not *both* have pheromone.

At that point, if no horizontal neighbor has pheromone then it turns East, taking signature $[v, \tau - v]$, whereas if only the East neighbor has pheromone then it turns West, taking signature $[v, v - \tau]$ (state Vert-bypass). Once it leaves the vertical axis, an agent keeps moving horizontally until the end of its lifetime in state Horiz.

3.1 Pseudocode

We give the transition function executed by each agent during step 4 of each round (cf. Definition 3) in Algorithm 1. We denote by φ_x, for $x \in \{N, E, S, W\}$, the pheromone value of the neighboring node in direction x. These represent the input to the FSM. In accordance with Definition 1, the pheromone values are never used directly but only as part of comparisons to each other and to the value 0. The output of the FSM is composed of the pair of values $(\mathsf{dir}, \mathsf{drop})$ at the end of the transition function computation.

3.2 Correctness

▶ **Theorem 8.** *Algorithm 1 correctly solves the treasure hunt problem for all combinations of parameters (τ, μ) with $\tau \geq 1$ and $\mu \geq 2$.*

The complete proof of Theorem 8 is available in the full version. The proof is based on the following simple properties of Algorithm 1:

- Whenever an agent switches to state Horiz it moves horizontally (East or West) and drops pheromone. Subsequently, it keeps moving in the same direction in the same state without dropping pheromone until the end of its lifetime.
- Whenever an agent switches to state Vert-seek it moves vertically (North or South) and drops pheromone. Subsequently, it keeps moving in the same direction in the same state without dropping pheromone until one of the following happens: it reaches the end of its lifetime, or it switches to state Vert-bypass moving in the same direction as before, or it switches to state Horiz moving West.
- Whenever an agent switches to state Vert-bypass it moves vertically, and it drops pheromone only if it switches from state Init to Vert-bypass. Subsequently, it keeps moving in the same direction in the same state without dropping pheromone until one of the following happens: it reaches the end of its lifetime, or it switches to state Horiz.
- During its first transition, every agent switches to one of the states Horiz, Vert-seek, or Vert-bypass.

Based on these, we conclude that every agent has a signature as per Definition 7. The rest of the proof is devoted to showing that the first 4τ agents pick up distinct signatures, and thus they explore all nodes at distance τ or less from the nest. This is accomplished by a series of lemmas, whose proofs are omitted. We show first that agents A_1 and A_2 have signatures $[0; \tau]$ and $[0; -\tau]$, respectively, and that subsequent agents are alternately dispatched to the North and to the South half-planes. Then, the following two technical lemmas, whose proofs are omitted, describe completely the state transitions of agents on the vertical axis:

▶ **Lemma 9.** *For all odd i with $3 \leq i \leq 4\tau - 1$, and for all y with $1 \leq y \leq \lceil \frac{i-1}{4} \rceil$, A_i is at node $(0, y)$ at the beginning of round $i + y$ and:*

1. *It senses pheromone $\mu \dot{-} (i - 4y)$ to the East and $\mu \dot{-} (i - 2 - 4y)$ to the West.*
2. *If $y = 1$, it is in state* Vert-seek *if $i - 3 \geq \mu$, otherwise it is in state* Vert-bypass.
3. *If $y \geq 2$, it is in state* Vert-seek *if $i - 4y + 2 \geq \mu$, otherwise it is in state* Vert-bypass.

▶ **Lemma 10.** *For all even i with $4 \leq i \leq 4\tau$, and for all y with $1 \leq y \leq \lceil \frac{i-2}{4} \rceil$, A_i is at node $(0, -y)$ at the beginning of round $i + y$ and:*

1. *It senses pheromone $\mu \dot{-} (i - 4y - 1)$ to the East and $\mu \dot{-} (i - 3 - 4y)$ to the West.*
2. *If $y = 1$, it is in state* Vert-seek *if $i - 4 \geq \mu$, otherwise it is in state* Vert-bypass.
3. *If $y \geq 2$, it is in state* Vert-seek *if $i - 4y + 1 \geq \mu$, otherwise it is in state* Vert-bypass.

From Lemmas 9 and 10, we deduce the signatures of the first 4τ agents and conclude the proof of Theorem 8.

Algorithm 1 A treasure hunt algorithm for $\tau \geq 1$, $\mu \geq 2$.

```
Variables
state ∈ {Init, Vert-seek, Vert-bypass, Horiz}          ▷ Initial value: Init
dir ∈ {$N, E, S, W, \perp$}                                ▷ Initial value: $\perp$
drop ∈ {before, after, $\perp$}                           ▷ Initial value: $\perp$

Transition function
1:  if state = Init then
2:      if $\varphi_N = \varphi_W = \varphi_S = \varphi_E = 0$ then
3:          state ← Horiz; dir ← $E$; drop ← after
4:      else if $\varphi_E > \varphi_W$ then
5:          state ← Horiz; dir ← $W$; drop ← after
6:      else
7:          state ← Vert-bypass if $\varphi_W > 0$ else Vert-seek
8:          dir ← $S$ if $\varphi_N > \varphi_S$ else $N$
9:          drop ← after
10:     end if
11: else if state = Vert-seek then
12:     if $\varphi_W = \varphi_E = 0$ then
13:         state ← Vert-seek; drop ← $\perp$                   ▷ keep searching
14:     else
15:         INTERPRET-SIGNALS
16:     end if
17: else if state = Vert-bypass then
18:     if $\varphi_W = \varphi_E = 0$ then
19:         state ← Horiz; dir ← $E$; drop ← after
20:     else
21:         INTERPRET-SIGNALS
22:     end if
23: else if state = Horiz then
24:     drop ← $\perp$
25: end if

26: procedure INTERPRET-SIGNALS
27:     if $\varphi_W = 0$ and $\varphi_E > 0$ then
28:         state ← Horiz; dir ← $W$; drop ← after
29:     else if $\varphi_W > 0$ then
30:         state ← Vert-bypass; drop ← $\perp$
31:     end if
32: end procedure
```

3.3 Complexity

Recall the definitions of $\mathbb{B}_x$ (ball of radius x around the nest) and $\mathbb{L}_x$ (layer of nodes at distance x from the nest) from Section 2.

▶ **Theorem 11.** *If the treasure is located at distance at most d, where $1 \leq d \leq \tau$, then Algorithm 1 locates the treasure in time at most $5d - 1$.*

Proof. From Lemmas 9 and 10, it follows that agents $A_1, \dots, A_{4d}$ have all possible signatures with vertical component at most d (in absolute value). Moreover, since the distance of every agent from the nest strictly increases in each round, each agent A_i reaches distance d from the nest in round $i + d - 1$. It follows that, by the time agent A_{4d} reaches distance d from the nest, hence by round $5d - 1$, all nodes at distance d or less from the nest have been explored. ◀

▶ **Theorem 12.** *If the treasure is located at distance $d = \tau$, then any treasure hunt algorithm needs at least $5\tau - 1$ rounds to locate the treasure in the worst case.*

Proof. A given agent can explore at most one node at distance τ within its lifetime. Since $\mathbb{L}_\tau$ contains 4τ nodes, a correct algorithm must create at least 4τ agents, the last of which reaches distance τ in round $4\tau + \tau - 1 = 5\tau - 1$. It follows that, in the worst case, the treasure cannot be located before round $5\tau - 1$. ◀

▶ **Theorem 13.** *Let $\mathcal{A}$ be any treasure hunt algorithm that is correct for a pair of parameters (τ, μ). For every $d \le \tau$, $\mathcal{A}$ needs at least $\sqrt{5}d$ rounds to explore all nodes up to distance d.*

Proof. Fix a $d \le \tau$ and let T be the first round at the end of which $\mathcal{A}$ explores all nodes up to distance d. Clearly, $T \ge d$ because otherwise no agent can reach any node at distance d. We also assume that $T < \sqrt{5}d$, and we will show a contradiction.

Consider $\mathbb{B}_x$, for $x \le d$ to be determined below. Among the agents $A_1, \dots, A_T$, those with $i \ge T - x + 1$ have moved at most x times by the end of round T, therefore they are unable to explore any node outside of $\mathbb{B}_x$. For every $i \le T - x$, agent A_i moves at most $T - i + 1$ times by the end of round T, and it needs at least x moves before it can exit $\mathbb{B}_x$. Therefore, A_i explores at most $T - i + 1 - x$ nodes outside of $\mathbb{B}_x$. Summing over all agents with $i \le T - x$ and taking also into account $\mathbb{B}_x$ itself, we conclude that, by the end of round T, algorithm $\mathcal{A}$ can explore at most

$$|\mathbb{B}_x| + \sum_{i=1}^{T-x} \left(T - i + 1 - x\right) = 2x^2 + 2x + 1 + \frac{(T-x)(T-x+1)}{2}$$

nodes. The above expression is minimized for $x = \frac{T}{5} - \frac{3}{10} < d$, whence we obtain that $\mathcal{A}$ explores at most

$$\frac{2T^2}{5} + \frac{4T}{5} + \frac{31}{40}$$

nodes. By definition of T, at that round $\mathcal{A}$ has explored at least $\mathbb{B}_d$, therefore:

$$\frac{2T^2}{5} + \frac{4T}{5} + \frac{31}{40} \ge 2d^2 + 2d + 1$$

whence it follows that $T > \sqrt{5}d$, a contradiction. ◀

▶ **Theorem 14.** *Algorithm 1 effectively uses 4τ agents, and that is optimal.*

Proof (sketch). By Lemmas 9 and 10, agents $A_1, \dots, A_{4\tau}$ have all possible signatures with vertical component at most τ (in absolute value). Moreover, it can be shown that every agent A_i only senses pheromone left by some earlier agent A_j, $j < i$ (details omitted). It follows that, even if no agents are generated after round 4τ, the above agents $A_1, \dots, A_{4\tau}$ will still perform the same trajectories and explore all nodes up to distance τ. Therefore, the effective agent utilization of Algorithm 1 is 4τ. This is optimal because there exist 4τ nodes at distance τ, and an agent can only visit at most one node at distance τ during its lifetime. ◀

▶ **Theorem 15.** *The pheromone utilization of Algorithm 1 is at most $O(d)$, and this is asymptotically optimal.*

Proof. By Theorem 11, the treasure is located in time $O(d)$, and each of the $O(d)$ agents that are created until then drops pheromone at most 2 times: once when it leaves the nest, and once if and when it leaves the vertical axis. Hence, the pheromone utilization of Algorithm 1 is $O(d)$.

To prove optimality, consider a treasure hunt algorithm $\mathcal{A}$ that uses asymptotically less than d pheromone, i.e., its pheromone utilization is bounded by some function $f(d)$ such that $\lim_{d\to\infty} \frac{f(d)}{d} = 0$. Let N be the number of states of the FSM $\mathcal{A}$.

By our assumption on $f(d)$, for every $\varepsilon > 0$ there exists a d_ε such that for all $d > d_\varepsilon$, $f(d) < \varepsilon \cdot d$. Let us fix, then, a $d_0 > N + 1$ such that $f(d_0) < \frac{d_0}{N+1}$. Moreover, it is well known and has been observed several times in the literature (see, e.g., [39, Lemma 5]) that a deterministic FSM that moves in a grid and does not interact with its environment can explore at most a constant-width band, infinite in one direction. Let W be the constant that bounds the number of nodes of any particular layer that are visited by such an agent.

Now, consider the execution of $\mathcal{A}$ in a system with parameters (τ, μ), where $\tau \geq WNd_0+1$ and the treasure is located at distance d_0. The number of layers on which at least one agent drops pheromone is clearly bounded by the pheromone utilization of $\mathcal{A}$, and hence by $f(d_0) < \frac{d_0}{N+1}$. It follows that there exists at least one layer $d_1 \leq d_0 - (N + 1)$, such that no agent drops pheromone on any of the layers $d_1, d_1 + 1, \ldots, d_1 + N$. Therefore, any agent that arrives at layer d_0 is already repeating a sequence of states during which it drops no pheromone.

Consider, now, the execution of $\mathcal{A}$ in the same system but with the treasure placed at distance $d^\star = WNd_0 + 1$. As agents do not perceive the presence of treasure, they will behave as in the previous case. In particular, even though there is an infinite number of agents coming out of layer d_0, their trajectories are contained in at most most $4d_0 \cdot N$ distinct bands of constant width W, infinite in one direction. This is because the trajectory of an agent that is coming out of d_0 is completely determined by the node from which it exits layer d_0 and the state in which it leaves the layer.

It follows that algorithm $\mathcal{A}$ explores at most $4WNd_0$ nodes of layer $d^\star$, but layer $d^\star$ contains $4d^\star > 4WNd_0$ nodes. Therefore, the adversary can place the treasure at a node that will not be explored by $\mathcal{A}$. ◀

4 A treasure hunt algorithm for $\mu \geq 1$ and $\tau \geq 16$

Similarly to Algorithm 1, the algorithm that we present in this section creates agents of all possible signatures $[v; h]$, as per Definition 7.

The main difficulty here is that the dropped pheromone can evaporate in one round only, in the case of $\mu = 1$. To explore a grid up to an unknown distance τ, where τ is also the lifetime of an agent, every node at distance τ has to be visited by at least one agent (Definition 6). This agent cannot stop even for one round and has to follow a shortest path to the node at distance τ. At the same time, agents have to be sent alternatively exploring each half of the grid (north and south, in our case), and so the shortest time interval between two following agents (moving to the positions to explore) is two rounds. This makes it difficult to solve the problem with pheromone evaporating in one round. It disappears too fast to provide any information to the next arriving agent.

In order to overcome this challenge, we use in Algorithm 2 two types of agents: *signaling* agents, that stop moving at key positions and start dropping pheromone according to some predetermined pattern, and *explorer* agents, that read these patterns on their way to the extreme grid positions without stopping even for a single round. Since signaling consumes rounds from the lifetime of signaling agents, these agents must stop at a sufficient distance away from the extreme positions, to still have enough lifetime to signal the required pattern. This distance is expressed by the parameter s of our algorithm. This also has an impact on the minimum agent lifetime that is required for the algorithm to operate correctly, as the furthest signaling agents must have enough lifetime to reach their signaling positions and complete the required pattern. Algorithm 2 works for all values of μ, but only for $\tau \geq 16$.

Binary word notations. Let us define some finite binary word notations that we will use in order to present the algorithm. The empty word is denoted by ϵ and the length of a word w by $|w|$. For any word w and integer $j \in \{1, \ldots, |w|\}$, $w[j]$ denotes the j^{th} most significant bit of w. Let $\mathsf{shiftleft}(w)$ return a word obtained by removing the most significant bit ($w[1]$) from w.

We now present Algorithm 2 by refering to the pseudocode that we give in this section. All proofs are omitted. Algorithm 2 uses a constant number of special states, as follows: Given a binary word w of length at most 9, $\mathsf{Pattern}(w)$ is the first of a sequence of $|w|$ states, during which the agent stays on the same node and drops (or not) pheromone according to the bit pattern w. $\mathsf{Forward}(s)$-$\mathsf{Explore}(E)$ (resp. $\mathsf{Forward}(s)$-$\mathsf{Explore}(W)$) is the first of a sequence of states during which the agent moves s steps forward (north or south, in the same direction as it was moving before entering this state) and then turns east (resp. west) and keeps moving in that direction until the end of its lifetime.

In the main part of the algorithm, agents leave the nest alternatively moving either north or south, on the vertical axis, until arriving s steps away from a non-explored yet line where they either stop for signaling (moving to state $\mathsf{Pattern}(w)$) or continue moving to reach this non-explored yet line to turn there either east (state $\mathsf{Forward}(s)$-$\mathsf{Explore}(E)$) or west (state $\mathsf{Forward}(s)$-$\mathsf{Explore}(W)$) for exploring each half of the line, s steps away.

Such a signaling, for exploring each next line at distance h, is achieved by using three agents. One is placed s lines before, and at one cell east from the vertical axis, i.e. at $(1, h-s)$ if heading north (resp. $(1, -(h-s))$, if heading south). The second one is also s lines before, but at one cell west from the vertical axis, i.e. at $(-1, h-s)$ (resp. $(-1, -(h-s))$). The third agent, is at $(0, h-s+1)$ (resp. $(0, -(h-s+1))$) . A newly arrived agent (at $(0, h-s)$ (resp. $(0, -(h-s))$)) senses the pheromone dropped by these three agents and performs actions according to the parsing of the sensed pattern. This part of the algorithm is controlled mainly by the INTERPRET-SIGNALS-PHASE2() procedure (see Alg. 2).

Operating in this way, with agents "jumping" each time s steps vertically, for exploring a line there, leaves at least the s first horizontal lines of each half of the grid unexplored. Hence, we need a special procedure for exploring these lines. For that, up to horizontal lines at distance s, the signaling agents stay longer for guiding some of the incoming agents to explore these lines (some other agents are still guided to "jump" for exploring the lines s steps further). This part of the algorithm is controlled mainly by the INTERPRET-SIGNALS-PHASE1() procedure (see Alg. 2).

Algorithm 2 A treasure hunt algorithm for $\tau \geq 16, \mu \geq 1$ and $s = 6$.

```
Variables
state ∈ {Init, Vert-seek, Vert-bypass, Horiz, Pattern($w$), Forward($k$)-Explore($E$)
    Forward($k$)-Explore($W$)}, $w \in \{0,1\}^9, k \in [0,s]$        ▷ Initial value: Init
dir ∈ $\{N, E, S, W, \perp\}$                                        ▷ Initial value: $\perp$
drop ∈ {before, after, $\perp$}                                      ▷ Initial value: $\perp$
moves ∈ $[0, s+1]$                                                   ▷ Initial value: 0

Transition function
1:  if state = Init then
2:      if $\varphi_N = \varphi_W = \varphi_S = \varphi_E = 0$ then
3:          state ← Pattern(11001); dir ← $E$; drop ← after              ▷ Start signaling E
4:      else if $\varphi_N = \varphi_W = \varphi_S = 0$ and $\varphi_E > 0$ then
5:          state ← Pattern(111); dir ← $W$; drop ← after                ▷ Start signaling W
6:      else if $\varphi_N = \varphi_S = 0$ and $\varphi_W = \varphi_E > 0$ then
7:          state ← Pattern(01); dir ← $N$; drop ← after                 ▷ Start signaling N
8:      else if $\varphi_S = 0$ and $\varphi_N = \varphi_W = \varphi_E > 0$ then
9:          state ← Horiz; dir ← $E$; drop ← $\perp$                     ▷ Explore E
10:     else if $\varphi_S = 0$ and $\varphi_N = \varphi_E < \varphi_W$ then
11:         state ← Horiz; dir ← $W$; drop ← $\perp$                     ▷ Explore W
12:     else if $\varphi_S = 0$ and $\varphi_N = \varphi_W > \varphi_E$ then
13:         state ← Vert-bypass; dir ← $N$; drop ← after        ▷ Go signaling E on line $(0,1)$
14:     else if $\varphi_S = 0$ and $\varphi_N = \varphi_E > \varphi_W$ then
15:         state ← Vert-bypass; dir ← $S$; drop ← after       ▷ Go signaling E on line $(0,-1)$
16:     else if $\varphi_S > \varphi_N$ and $\varphi_S > \varphi_E$ and $\varphi_S > \varphi_W$ then
17:         state ← Vert-seek; dir ← $N$; drop ← after          ▷ Go signaling W on line $(0,1)$
18:     else if $\varphi_N > \varphi_S$ and $\varphi_N > \varphi_E$ and $\varphi_N > \varphi_W$ then
19:         state ← Vert-seek; dir ← $S$; drop ← after         ▷ Go signaling W on line $(0,-1)$
20:     end if
21: else if state = Vert-seek then
22:     if $\varphi_{\text{dir}} = \varphi_E = \varphi_W = 0$ then
23:         drop ← $\perp$                                              ▷ keep searching
24:     else if moves < $s+1$ then
25:         state ← Vert-bypass; INTERPRET-SIGNALS-PHASE1
26:     else
27:         state ← Vert-bypass; INTERPRET-SIGNALS-PHASE2
28:     end if
29: else if state = Vert-bypass then
30:     if moves < $s+1$ then
31:         INTERPRET-SIGNALS-PHASE1
32:     else
33:         INTERPRET-SIGNALS-PHASE2
34:     end if
35: else if state = Horiz then
36:     drop ← $\perp$
37: else if state = Forward($k$)-Explore($d$) then
38:     if $k > 1$ then
39:         state ← Forward($k-1$)-Explore($d$)
40:     else
41:         state ← Horiz; dir ← $d$
42:     end if
43:     drop ← $\perp$
```

► Algorithm 2 (continued)

```
else if state = Pattern($w$) then
    dir $\leftarrow \perp$
    if $|w| > 1 \wedge w[1] = 1$ then                    ▷ $w[1]$ returns the first bit of the binary word $w$
        drop $\leftarrow$ after
    else
        drop $\leftarrow \perp$
    end if
    if $w \neq \epsilon$ then
        state $\leftarrow$ Pattern(shiftleft($w$))              ▷ shiftleft($w$) removes the first bit of $w$
    end if
end if
if dir $\neq \perp \wedge$ moves $< s + 1$ then
    moves $\leftarrow$ moves $+ 1$
end if
procedure INTERPRET-SIGNALS-PHASE1
    if $\varphi_{\mathsf{dir}} = \varphi_E = \varphi_W = 0$ then
        state $\leftarrow$ Pattern(1 01 01 01); dir $\leftarrow E$; drop $\leftarrow \perp$            ▷ Start signaling E
    else if $\varphi_{\mathsf{dir}} = \varphi_W = 0$ and $\varphi_E > 0$ then
        state $\leftarrow$ Pattern(1 01 00 01 01); dir $\leftarrow W$; drop $\leftarrow \perp$         ▷ Start signaling W
    else if $\varphi_{\mathsf{dir}} = 0$ and $\varphi_E = \varphi_W > 0$ then
        state $\leftarrow$ Pattern(1 01 00 01 01); drop $\leftarrow \perp$                  ▷ Start signaling N or S
    else if $\varphi_{\mathsf{dir}} = \varphi_E = \varphi_W > 0$ then
        state $\leftarrow$ Horiz; dir $\leftarrow E$; drop $\leftarrow \perp$                          ▷ Explore E
    else if $\varphi_{\mathsf{dir}} = \varphi_E > \varphi_W$ then
        state $\leftarrow$ Horiz; dir $\leftarrow W$; drop $\leftarrow \perp$                          ▷ Explore W
    else if $\varphi_{\mathsf{dir}} = \varphi_E < \varphi_W$ then
        state $\leftarrow$ Forward($s$)-Explore($E$); drop $\leftarrow \perp$      ▷ Move forward s steps and explore E
    else if $\varphi_{\mathsf{dir}} = \varphi_W > \varphi_E$ then
        state $\leftarrow$ Forward($s$)-Explore($W$); drop $\leftarrow \perp$      ▷ Move forward s steps and explore W
    else if $\varphi_{\mathsf{dir}} > \varphi_E$ and $\varphi_{\mathsf{dir}} > \varphi_W$ then
        drop $\leftarrow \perp$                                       ▷ Continue to bypass pheromone traces
    end if
end procedure
procedure INTERPRET-SIGNALS-PHASE2
    if $\varphi_{\mathsf{dir}} = \varphi_E = \varphi_W = 0$ then
        state $\leftarrow$ Pattern(1 01 01); dir $\leftarrow E$; drop $\leftarrow \perp$              ▷ Start signaling E
    else if $\varphi_{\mathsf{dir}} = \varphi_W = 0$ and $\varphi_E > 0$ then
        state $\leftarrow$ Pattern(1 01 01); dir $\leftarrow W$; drop $\leftarrow \perp$              ▷ Start signaling W
    else if $\varphi_{\mathsf{dir}} = 0$ and $\varphi_E = \varphi_W > 0$ then
        state $\leftarrow$ Pattern(1 01 01); drop $\leftarrow \perp$                       ▷ Start signaling N or S
    else if $\varphi_{\mathsf{dir}} = \varphi_E = \varphi_W > 0$ then
        state $\leftarrow$ Forward($s$)-Explore($E$); drop $\leftarrow \perp$      ▷ Move forward s steps and explore E
    else if $\varphi_{\mathsf{dir}} = \varphi_W > \varphi_E$ then
        state $\leftarrow$ Forward($s$)-Explore($W$); drop $\leftarrow \perp$      ▷ Move forward s steps and explore W
    else if $\varphi_{\mathsf{dir}} > \varphi_E$ and $\varphi_{\mathsf{dir}} > \varphi_W$ then
        drop $\leftarrow \perp$                                       ▷ Continue to bypass pheromone traces
    end if
end procedure
```

▶ Remark 16. The technical analysis shows that the algorithm works with $s = 6$. We give a short intuition for this value. As briefly explained above, the point is that signaling patterns cannot be established too far from the nest, because agents do not have enough remaining lifetime to complete the pattern. As such, the exploration of horizontal lines that are far from the nest must be signaled by patterns that are set up closer to the nest. In fact, s depends on the longest such signaling pattern, dropped by a signaling agent (at a distance further than s from the nest). This in turn establishes the closest position of such agent to the grid extremity (at distance τ), where it can complete the signaling before it dies. In our algorithm, to explore lines after distance s, only 6 rounds are used by a signaling agent, which explains why $s = 6$. We actually need to encode 6 actions (3 for signaling agents and 3 for exploring). This requires 12 rounds of signaling due to the N/S dispatching at the nest, but we can get away with s being only 6 because the agents arrive at different times. Still, signaling agents have to stay alive there only for 6 rounds each.

Regarding the minimal τ which is 16, it is due to the transition from operation in the s first lines to the next ones. During this transition, signaling agents should have enough remaining lifetime to reach line s and to signal the required pattern (in these lines, the signaling pattern of each agent requires 10 rounds; there are $3 + 5$ actions to signal here). So 10 rounds for signaling and 6 rounds to reach the line at distance 6 gives $\tau \geq 16$ rounds.

Let us detail now the operation of the algorithm during the first rounds intended to explore the x-axis (this differs from the exploration of other lines). Agents start at the nest in state Init. Each of the first three agents are placed respectively east, west and north to the nest and start signaling according to the predetermined pattern (lines 3, 5 and 7, Alg. 2). This signaling instructs the 4th agent (A_4) to explore the east half of the x-axis (line 9) and the 5th agent (A_5), to explore the remaining (west) half of the x-axis (line 11). The next four agents are instructed to move to lines $(0, 1)$ and $(0, -1)$ (lines 13 - 19), two agents on each line, to stop on the East and West from the vertical axis (cells $(1, 1), (-1, 1), (1, -1), (-1, -1)$). This is for instructing to explore lines $(0, 1), (0, -1), (0, s + 1)$ and $(0, -s - 1)$ (as explained in the previous paragraph).

Notice that starting from round 8, every even round, an agent in Vert-seek leaves the nest to the North, and every odd round, an agent in Vert-seek leaves the nest to the South. This alternation allows to explore both the north and the south halves of the grid, without knowing its size.

States Vert-seek and Vert-bypass are used in a similar way as in the previous algorithm, to overcome the difficulty caused by the pheromone traces left from previous drops in case of $\mu > 1$. An agent has to bypass (in state Vert-bypass) these traces (lines 74 and 89) until arriving to a line with either no pheromone or with "fresh" pheromones, just dropped in the previous round (treated in all other lines of the INTERPRET-SIGNALS-PHASE1() and INTERPRET-SIGNALS-PHASE2() procedures). Starting with the 8th agent, agents leave the nest in state Vert-seek and move vertically in this state until sensing some dropped pheromone, moving then to Vert-bypass (lines 22 - 27).

▶ **Theorem 17.** *Algorithm 2 solves the treasure hunt problem for $\mu \geq 1, \tau \geq 16$ and $s = 6$ in $11\tau - 6s + 2$ rounds, using $10\tau - 6s + 3$ agents and $28\tau + O(s) + 8$ pheromone drops.*

5 Concluding remarks

We have presented the first algorithms for the treasure hunt problem under the weak communication mechanism of evaporating pheromone markers. In Algorithm 1, the assumption that pheromone lasts for at least two rounds ($\mu \geq 2$) leads to a fairly simple algorithm design with very few states. By contrast, Algorithm 2 is significantly more complicated, as it needs to be able to handle both an extremely fast evaporation rate ($\mu = 1$) and larger values of μ.

Algorithm 2 covers all values of the evaporation parameter $\mu \geq 1$, but it requires a lifetime of $\tau \geq 16$. It would be interesting to determine the smallest τ_0 such that there exists a treasure hunt algorithm that works for all $\mu \geq 1$ and for all $\tau \geq \tau_0$. With ad-hoc arguments, it can be seen that $\tau_0 > 2$. However, it is far from obvious how to generalize these arguments to larger values of τ_0. On the other hand, there may be room to improve the upper bound of $\tau_0 \leq 16$, with some fine-tuning of the signaling patterns.

Another interesting direction for future work is improving on the complexities of Algorithm 2, or studying tradeoffs between completion time, pheromone utilization, and agent utilization. Since both of our algorithms use only a constant number of pheromone drops per agent, one idea would be to increase the frequency of pheromone drops. It seems that this would not help to reduce agent utilization or the completion time. Indeed, the limiting factor in Algorithm 2 seems to be not the amount of pheromone that is dropped or that might be dropped, but indeed the number of grid positions that are available in order to set up an efficient pattern, i.e., a pattern that resides in the neighborhood of the main axis so that it can be immediately sensed by agents.

The assumption of detecting pheromones only in adjacent nodes to the agent, although natural, could be relaxed. However, if the sensing range is increased even to 2 while maintaining the principle that the agent can pinpoint exactly the position of the pheromone and compare pheromone levels between all nodes in its 2-neighborhood, then Algorithm 1 resolves the problem for all values of the parameters. Indeed, the only reason why Algorithm 1 fails for $\mu = 1$ is that, due to the North-South dispatching at the nest, agents are dispatched into the same half-plane every two rounds, and therefore any pheromone dropped by an agent evaporates before the next agent can sense it. Consequently, in order to study a meaningful problem with an increased sensing range, some loss of information would have to be introduced at distance 2 or more.

Our algorithms are quite far from modeling natural ant foraging patterns. Indeed, depending on species, ants in nature tend to employ a wide range of communication methods, including multiple types of pheromone of various degrees of volatility, repellent pheromones, contact, or sounds [49]. However, our proposed solutions are more appropriate for artificial agent systems, where the parameter τ might correspond to agents with limited energy, and evaporating markers could be useful to prevent area pollution. The appropriate parameter values will depend on the specific application.

As a general remark, we believe that the communication model of evaporating pheromone markers is inherently interesting and we would like to study other agent coordination problems in this model. Orthogonally, one may consider less predictable evaporation mechanisms, such as evaporation governed by a random process, or controlled by an adversary.

References

1 Yehuda Afek, Roman Kecher, and Moshe Sulamy. Optimal pheromone utilization. In *2nd Workshop on Biological Distributed Algorithms (BDA)*, 2014.

2 Yehuda Afek, Roman Kecher, and Moshe Sulamy. Optimal and resilient pheromone utilization in ant foraging. *CoRR*, abs/1507.00772, 2015. arXiv:1507.00772.

3 Abhinav Aggarwal and Jared Saia. ANTS on a plane. In Andrea Werneck Richa and Christian Scheideler, editors, *Structural Information and Communication Complexity - 27th International Colloquium, SIROCCO 2020, Paderborn, Germany, June 29 - July 1, 2020, Proceedings*, volume 12156 of *Lecture Notes in Computer Science*, pages 47–62. Springer, 2020. doi:10.1007/978-3-030-54921-3_3.

4 Noga Alon, Chen Avin, Michal Koucký, Gady Kozma, Zvi Lotker, and Mark R. Tuttle. Many random walks are faster than one. *Combinatorics, Probability and Computing*, 20(4):481–502, 2011. doi:10.1017/S0963548311000125.

5 Steve Alpern, Robbert Fokkink, Leszek Gąsieniec, Roy Lindelauf, and V.S. Subrahmanian, editors. *Search Theory: A Game Theoretic Perspective*. Springer New York, NY, 2013. doi:10.1007/978-1-4614-6825-7.

6 Steve Alpern and Shmuel Gal. *The Theory of Search Games and Rendezvous*. International Series in Operations Research & Management Science. Springer New York, NY, 2003. doi:10.1007/b100809.

7 Spyros Angelopoulos. Online search with a hint. In James R. Lee, editor, *12th Innovations in Theoretical Computer Science Conference, ITCS 2021, January 6-8, 2021, Virtual Conference*, volume 185 of *LIPIcs*, pages 51:1–51:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ITCS.2021.51.

8 Spyros Angelopoulos, Diogo Arsénio, and Christoph Dürr. Infinite linear programming and online searching with turn cost. *Theoretical Computer Science*, 670:11–22, 2017. doi:10.1016/j.tcs.2017.01.013.

9 Spyros Angelopoulos, Alejandro López-Ortiz, and Konstantinos Panagiotou. Multi-target ray searching problems. *Theoretical Computer Science*, 540:2–12, 2014. doi:10.1016/j.tcs.2014.03.028.

10 Spyros Angelopoulos and Malachi Voss. Online search with maximum clearance. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 3642–3650. AAAI Press, 2021. URL: https://ojs.aaai.org/index.php/AAAI/article/view/16480.

11 Ricardo A. Baeza-Yates and René Schott. Parallel searching in the plane. *Computational Geometry*, 5:143–154, 1995. doi:10.1016/0925-7721(95)00003-R.

12 Evangelos Bampas, Jurek Czyzowicz, Leszek Gasieniec, David Ilcinkas, Ralf Klasing, Tomasz Kociumaka, and Dominik Pajak. Linear search by a pair of distinct-speed robots. *Algorithmica*, 81(1):317–342, 2019. doi:10.1007/s00453-018-0447-0.

13 Evangelos Bampas, Jurek Czyzowicz, David Ilcinkas, and Ralf Klasing. Beachcombing on strips and islands. *Theoretical Computer Science*, 806:236–255, 2020. doi:10.1016/j.tcs.2019.04.001.

14 Anatole Beck. On the linear search problem. *Israel Journal of Mathematics*, 2:221–228, 1964. doi:10.1007/BF02759737.

15 Anatole Beck and Donald J. Newman. Yet more on the linear search problem. *Israel Journal of Mathematics*, 8:419–429, 1970. doi:10.1007/BF02798690.

16 Lucas Boczkowski, Uriel Feige, Amos Korman, and Yoav Rodeh. Navigating in trees with permanently noisy advice. *ACM Transactions on Algorithms*, 17(2):15:1–15:27, 2021. doi:10.1145/3448305.

17 Anthony Bonato and Richard J. Nowakowski. *The Game of Cops and Robbers on Graphs*. American Mathematical Society, 2011.

18 Sébastien Bouchard, Yoann Dieudonné, Andrzej Pelc, and Franck Petit. Deterministic treasure hunt in the plane with angular hints. *Algorithmica*, 82(11):3250–3281, 2020. doi:10.1007/s00453-020-00724-4.

19 Sébastien Bouchard, Arnaud Labourel, and Andrzej Pelc. Impact of knowledge on the cost of treasure hunt in trees. *Networks*, 80(1):51–62, 2022. doi:10.1002/net.22075.

20 Sebastian Brandt, Klaus-Tycho Foerster, Benjamin Richner, and Roger Wattenhofer. Wireless evacuation on m rays with k searchers. *Theoretical Computer Science*, 811:56–69, 2020. doi:10.1016/j.tcs.2018.10.032.

21 Sebastian Brandt, Jara Uitto, and Roger Wattenhofer. A tight lower bound for semi-synchronous collaborative grid exploration. *Distributed Computing*, 33(6):471–484, 2020. doi:10.1007/s00446-020-00369-0.

22 Marek Chrobak, Leszek Gasieniec, Thomas Gorry, and Russell Martin. Group search on the line. In Giuseppe F. Italiano, Tiziana Margaria-Steffen, Jaroslav Pokorný, Jean-Jacques Quisquater, and Roger Wattenhofer, editors, *SOFSEM 2015: Theory and Practice of Computer Science - 41st International Conference on Current Trends in Theory and Practice of Computer Science, Pec pod Sněžkou, Czech Republic, January 24-29, 2015. Proceedings*, volume 8939 of *Lecture Notes in Computer Science*, pages 164–176. Springer, 2015. doi:10.1007/978-3-662-46078-8_14.

23 Timothy H. Chung, Geoffrey A. Hollinger, and Volkan Isler. Search and pursuit-evasion in mobile robotics - A survey. *Autonomous Robots*, 31(4):299–316, 2011. doi:10.1007/s10514-011-9241-4.

24 Andrea E. F. Clementi, Francesco D'Amore, George Giakkoupis, and Emanuele Natale. Search via parallel Lévy walks on $\mathbb{Z}^2$. In Avery Miller, Keren Censor-Hillel, and Janne H. Korhonen, editors, *PODC '21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021*, pages 81–91. ACM, 2021. doi:10.1145/3465084.3467921.

25 Lihi Cohen, Yuval Emek, Oren Louidor, and Jara Uitto. Exploring an infinite space with finite memory scouts. In Philip N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 207–224. SIAM, 2017. doi:10.1137/1.9781611974782.14.

26 Jared Ray Coleman and Oscar Morales-Ponce. The snow plow problem: Perpetual maintenance by mobile agents on the line. In *24th International Conference on Distributed Computing and Networking, ICDCN 2023, Kharagpur, India, January 4-7, 2023*, pages 110–114. ACM, 2023. doi:10.1145/3571306.3571396.

27 Andrew Collins, Jurek Czyzowicz, Leszek Gasieniec, Adrian Kosowski, Evangelos Kranakis, Danny Krizanc, Russell Martin, and Oscar Morales-Ponce. Optimal patrolling of fragmented boundaries. In Guy E. Blelloch and Berthold Vöcking, editors, *25th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '13, Montreal, QC, Canada - July 23 - 25, 2013*, pages 241–250. ACM, 2013. doi:10.1145/2486159.2486176.

28 Colin Cooper, Alan M. Frieze, and Tomasz Radzik. Multiple random walks in random regular graphs. *SIAM Journal on Discrete Mathematics*, 23(4):1738–1761, 2009. doi:10.1137/080729542.

29 Jurek Czyzowicz, Leszek Gasieniec, Konstantinos Georgiou, Evangelos Kranakis, and Fraser MacQuarrie. The beachcombers' problem: Walking and searching with mobile robots. *Theoretical Computer Science*, 608:201–218, 2015. doi:10.1016/j.tcs.2015.09.011.

30 Jurek Czyzowicz, Kostantinos Georgiou, and Evangelos Kranakis. Group search and evacuation. In Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro, editors, *Distributed Computing by Mobile Entities, Current Research in Moving and Computing*, volume 11340 of *Lecture Notes in Computer Science*, pages 335–370. Springer, 2019. doi:10.1007/978-3-030-11072-7_14.

31 Francesco D'Amore. *On the collective behaviors of bio-inspired distributed systems. (Sur les comportements collectifs de systèmes distribués bio-inspirés)*. PhD thesis, Côte d'Azur University, Nice, France, 2022. URL: `https://tel.archives-ouvertes.fr/tel-03906167`.

32 Shantanu Das. Mobile agent rendezvous in a ring using faulty tokens. In Shrisha Rao, Mainak Chatterjee, Prasad Jayanti, C. Siva Ram Murthy, and Sanjoy Kumar Saha, editors, *Distributed Computing and Networking, 9th International Conference, ICDCN 2008, Kolkata, India, January 5-8, 2008*, volume 4904 of *Lecture Notes in Computer Science*, pages 292–297. Springer, 2008. `doi:10.1007/978-3-540-77444-0_29`.

33 Shantanu Das, Matús Mihalák, Rastislav Srámek, Elias Vicari, and Peter Widmayer. Rendezvous of mobile agents when tokens fail anytime. In Theodore P. Baker, Alain Bui, and Sébastien Tixeuil, editors, *Principles of Distributed Systems, 12th International Conference, OPODIS 2008, Luxor, Egypt, December 15-18, 2008. Proceedings*, volume 5401 of *Lecture Notes in Computer Science*, pages 463–480. Springer, 2008. `doi:10.1007/978-3-540-92221-6_29`.

34 Shantanu Das and Nicola Santoro. Moving and computing models: Agents. In Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro, editors, *Distributed Computing by Mobile Entities, Current Research in Moving and Computing*, volume 11340 of *Lecture Notes in Computer Science*, pages 15–34. Springer, 2019. `doi:10.1007/978-3-030-11072-7_2`.

35 Erik D. Demaine, Sándor P. Fekete, and Shmuel Gal. Online searching with turn cost. *Theoretical Computer Science*, 361(2-3):342–355, 2006. `doi:10.1016/j.tcs.2006.05.018`.

36 Yoann Dieudonné and Andrzej Pelc. Deterministic network exploration by a single agent with Byzantine tokens. *Information Processing Letters*, 112(12):467–470, 2012. `doi:10.1016/j.ipl.2012.03.017`.

37 Klim Efremenko and Omer Reingold. How well do random walks parallelize? In Irit Dinur, Klaus Jansen, Joseph Naor, and José D. P. Rolim, editors, *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, 12th International Workshop, APPROX 2009, and 13th International Workshop, RANDOM 2009, Berkeley, CA, USA, August 21-23, 2009. Proceedings*, volume 5687 of *Lecture Notes in Computer Science*, pages 476–489. Springer, 2009. `doi:10.1007/978-3-642-03685-9_36`.

38 Robert Elsässer and Thomas Sauerwald. Tight bounds for the cover time of multiple random walks. *Theoretical Computer Science*, 412(24):2623–2641, 2011. `doi:10.1016/j.tcs.2010.08.010`.

39 Yuval Emek, Tobias Langner, David Stolz, Jara Uitto, and Roger Wattenhofer. How many ants does it take to find the food? *Theoretical Computer Science*, 608:255–267, 2015. `doi:10.1016/j.tcs.2015.05.054`.

40 Yuval Emek, Tobias Langner, Jara Uitto, and Roger Wattenhofer. Solving the ANTS problem with asynchronous finite state machines. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II*, volume 8573 of *Lecture Notes in Computer Science*, pages 471–482. Springer, 2014. `doi:10.1007/978-3-662-43951-7_40`.

41 Ofer Feinerman and Amos Korman. Memory lower bounds for randomized collaborative search and implications for biology. In Marcos K. Aguilera, editor, *Distributed Computing - 26th International Symposium, DISC 2012, Salvador, Brazil, October 16-18, 2012. Proceedings*, volume 7611 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2012. `doi:10.1007/978-3-642-33651-5_5`.

42 Ofer Feinerman and Amos Korman. The ANTS problem. *Distributed Computing*, 30(3):149–168, 2017. `doi:10.1007/s00446-016-0285-8`.

43 Ofer Feinerman, Amos Korman, Zvi Lotker, and Jean-Sébastien Sereni. Collaborative search on the plane without communication. In Darek Kowalski and Alessandro Panconesi, editors, *ACM Symposium on Principles of Distributed Computing, PODC '12, Funchal, Madeira, Portugal, July 16-18, 2012*, pages 77–86. ACM, 2012. `doi:10.1145/2332432.2332444`.

44 Paola Flocchini, Evangelos Kranakis, Danny Krizanc, Flaminia L. Luccio, Nicola Santoro, and Cindy Sawchuk. Mobile agents rendezvous when tokens fail. In Rastislav Kralovic and Ondrej Sýkora, editors, *Structural Information and Communication Complexity, 11th International Colloquium , SIROCCO 2004, Smolenice Castle, Slovakia, June 21-23, 2004, Proceedings*, volume 3104 of *Lecture Notes in Computer Science*, pages 161–172. Springer, 2004. doi:10.1007/978-3-540-27796-5_15.

45 Fedor V. Fomin and Dimitrios M. Thilikos. An annotated bibliography on guaranteed graph searching. *Theoretical Computer Science*, 399(3):236–245, 2008. doi:10.1016/j.tcs.2008.02.040.

46 Pierre Fraigniaud, Amos Korman, and Yoav Rodeh. Parallel Bayesian search with no coordination. *Journal of the ACM*, 66(3):17:1–17:28, 2019. doi:10.1145/3304111.

47 Subir Kumar Ghosh and Rolf Klein. Online algorithms for searching and exploration in the plane. *Computer Science Review*, 4(4):189–201, 2010. doi:10.1016/j.cosrev.2010.05.001.

48 Andrej Ivaskovic, Adrian Kosowski, Dominik Pajak, and Thomas Sauerwald. Multiple random walks on paths and grids. In Heribert Vollmer and Brigitte Vallée, editors, *34th Symposium on Theoretical Aspects of Computer Science, STACS 2017, March 8-11, 2017, Hannover, Germany*, volume 66 of *LIPIcs*, pages 44:1–44:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.STACS.2017.44.

49 Duncan E. Jackson and Francis L.W. Ratnieks. Communication in ants. *Current Biology*, 16(15):R570–R574, 2006. doi:10.1016/j.cub.2006.07.015.

50 Akitoshi Kawamura and Yusuke Kobayashi. Fence patrolling by mobile agents with distinct speeds. *Distributed Computing*, 28(2):147–154, 2015. doi:10.1007/s00446-014-0226-3.

51 Dennis Komm, Rastislav Královic, Richard Královic, and Jasmin Smula. Treasure hunt with advice. In Christian Scheideler, editor, *Structural Information and Communication Complexity - 22nd International Colloquium, SIROCCO 2015, Montserrat, Spain, July 14-16, 2015, Post-Proceedings*, volume 9439 of *Lecture Notes in Computer Science*, pages 328–341. Springer, 2015. doi:10.1007/978-3-319-25258-2_23.

52 Amos Korman and Yoav Rodeh. Multi-round cooperative search games with multiple players. *Journal of Computer and System Sciences*, 113:125–149, 2020. doi:10.1016/j.jcss.2020.05.003.

53 Tobias Langner, Barbara Keller, Jara Uitto, and Roger Wattenhofer. Overcoming obstacles with ants. In Emmanuelle Anceaume, Christian Cachin, and Maria Gradinariu Potop-Butucaru, editors, *19th International Conference on Principles of Distributed Systems, OPODIS 2015, December 14-17, 2015, Rennes, France*, volume 46 of *LIPIcs*, pages 9:1–9:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015. doi:10.4230/LIPIcs.OPODIS.2015.9.

54 Tobias Langner, Jara Uitto, David Stolz, and Roger Wattenhofer. Fault-tolerant ANTS. In Fabian Kuhn, editor, *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings*, volume 8784 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2014. doi:10.1007/978-3-662-45174-8_3.

55 Christoph Lenzen, Nancy A. Lynch, Calvin Newport, and Tsvetomira Radeva. Searching without communicating: tradeoffs between performance and selection complexity. *Distributed Computing*, 30(3):169–191, 2017. doi:10.1007/s00446-016-0283-x.

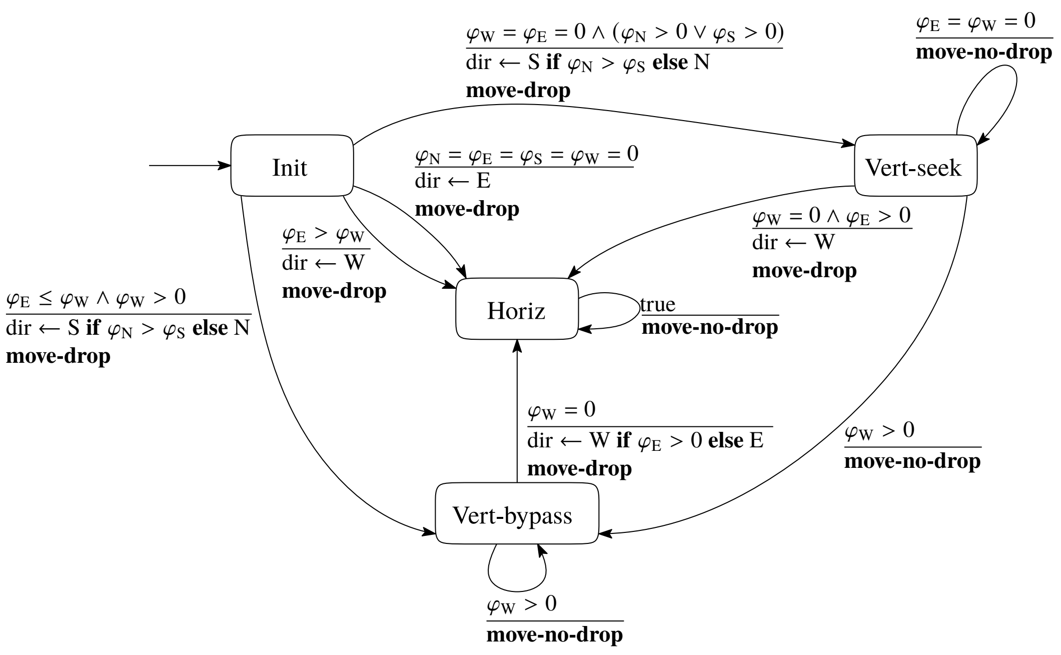
56 Christoph Lenzen and Tsvetomira Radeva. The power of pheromones in ant foraging. In *1st Workshop on Biological Distributed Algorithms (BDA)*, 2013.

57 Avery Miller and Andrzej Pelc. Tradeoffs between cost and information for rendezvous and treasure hunt. *Journal of Parallel and Distributed Computing*, 83:159–167, 2015. doi:10.1016/j.jpdc.2015.06.004.

58 Paul J. Nahin. *Chases and Escapes: The Mathematics of Pursuit and Evasion*. Princeton University Press, 2007.

59 Andrzej Pelc and Ram Narayan Yadav. Advice complexity of treasure hunt in geometric terrains. *Information and Computation*, 281:104705, 2021. doi:10.1016/j.ic.2021.104705.

A State transition diagram of Algorithm 1



Init
Vert-seek
Horiz
Vert-bypass
$\varphi_W = \varphi_E = 0 \wedge (\varphi_N > 0 \vee \varphi_S > 0)$
dir ← S **if** $\varphi_N > \varphi_S$ **else** N
move-drop
$\varphi_E = \varphi_W = 0$
move-no-drop
$\varphi_N = \varphi_E = \varphi_S = \varphi_W = 0$
dir ← E
move-drop
$\varphi_E > \varphi_W$
dir ← W
move-drop
$\varphi_W = 0 \wedge \varphi_E > 0$
dir ← W
move-drop
$\varphi_E \le \varphi_W \wedge \varphi_W > 0$
dir ← S **if** $\varphi_N > \varphi_S$ **else** N
move-drop
true
move-no-drop
$\varphi_W = 0$
dir ← W **if** $\varphi_E > 0$ **else** E
move-drop
$\varphi_W > 0$
move-no-drop
$\varphi_W > 0$
move-no-drop

Figure 2 A hybrid state transition diagram representing Algorithm 1. On each transition, the guard condition is given above the horizontal line. The actions that are executed if the transition is triggered are given below the horizontal line. The values φ_x, for $x \in \{N, E, S, W\}$, represent the pheromone values in neighboring nodes at the beginning of the round. dir $\in \{N, E, S, W\}$ is a variable whose value persists between transitions. The statement **move-drop** instructs the agent to move in the direction indicated by the variable dir, dropping pheromone on the destination node. The statement **move-no-drop** instructs the agent to move in the direction indicated by the variable dir, without dropping pheromone on the destination node. Exactly one guarded transition is enabled from each state.

The FIDS Theorems: Tensions Between Multinode and Multicore Performance in Transactional Systems

Naama Ben-David
VMware Research, Palo Alto, California, USA

Gal Sela[1]
Technion, Haifa, Israel

Adriana Szekeres
VMware Research, Bellevue, Washington, USA

Abstract

Traditionally, distributed and parallel transactional systems have been studied in isolation, as they targeted different applications and experienced different bottlenecks. However, modern high-bandwidth networks have made the study of systems that are both distributed (i.e., employ multiple nodes) and parallel (i.e., employ multiple cores per node) necessary to truly make use of the available hardware.

In this paper, we study the performance of these combined systems and show that there are inherent tradeoffs between a system's ability to have fast and robust distributed communication and its ability to scale to multiple cores. More precisely, we formalize the notions of a *fast deciding* path of communication to commit transactions quickly in good executions, and *seamless fault tolerance* that allows systems to remain robust to server failures. We then show that there is an inherent tension between these two natural distributed properties and well-known multicore scalability properties in transactional systems. Finally, we show positive results; it is possible to construct a parallel distributed transactional system if any one of the properties we study is removed.

2012 ACM Subject Classification Computing methodologies → Concurrent computing methodologies; Computing methodologies → Distributed algorithms

Keywords and phrases transactions, distributed systems, parallel systems, impossibility results

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.9

Related Version *Full Version*: `https://arxiv.org/abs/2308.03919` [9]

1 Introduction

Transactional systems offer a clean abstraction for programmers to write concurrent code without worrying about synchronization issues. This has made them extremely popular and well studied in the last couple of decades [5, 21, 23, 41, 47, 53, 54, 48].

Many transactional systems in practice are *distributed* across multiple machines [14, 55, 30], allowing them to have many benefits that elude single-machine designs. For example, distributed solutions can scale to much larger data sets, handle much larger workloads, service clients that are physically far apart, and tolerate server failures. It is therefore unsurprising that distributed transactional systems have garnered a lot of attention in the literature, with many designs aimed at optimizing their performance in various ways: minimizing network round trips to commit transactions [29, 47, 54, 38, 15, 36], increasing robustness and availability when server failures occur [29, 54, 47], and scaling to heavier workloads [55, 19].

[1] Part of this work was done while this author was at VMware Research.

Due to increased bandwidth on modern networks, new considerations must be taken into account to keep improving the performance of distributed transactional systems. In particular, while traditional network communication costs formed the main bottleneck for many applications, sequential processing within each node is now no longer enough to handle the throughput that modern networks can deliver (through e.g., high-bandwidth links, multicore NICs, RDMA, kernel bypassing). Thus, to keep up with the capabilities of modern hardware, distributed transactional systems must make use of the parallelism available on each server that they use. That is, they must be designed while optimizing *both* network communication *and* multicore scalability.

Two main approaches have been employed by transactional storage systems to take advantage of the multicore architecture of their servers [46]: *shared-nothing* or *shared-memory.* The shared-nothing approach, where each core can access a distinct partition of the database and only communicates with other cores through message passing, has a significant drawback: cores responsible for hot data items become a throughput bottleneck while other cores are underutilized. To be able to adapt to workloads that stress a few hot data items, the shared memory approach, where each core can access any part of the memory, can be used. However, shared memory must be designed with care, as synchronization overheads can hinder scalability. Fortunately, decades of work has studied how to scale transactional systems in a multicore shared-memory setup [7, 6, 5, 10, 12, 40, 41, 8]. Thus, there is a lot of knowledge to draw from when designing distributed transactional systems that also employ parallelism within each server via the shared-memory approach.

In this paper, we study such systems, which we call *parallel distributed transactional systems (PDTSs).* Our main contribution is to show that there is an inherent tension between properties known to improve performance in distributed settings and those known to improve performance in parallel settings. To show this result, we first formalize a model that combines both shared memory and message passing systems. While such a model has been formulated in the past [1], it has not been formulated in the context of transactional systems.

We then describe and formalize three properties of distributed transactional systems that improve their performance. These properties have all appeared in various forms intuitively in the literature [54, 29, 47], but have never been formalized until now. We believe that each of them may be of independent interest, as they capture notions that apply to many existing systems. In particular, we first present *distributed disjoint-access parallelism*, a property inspired by its counterpart for multicore systems, but which captures scalability across different distributed shards of data. Then, we describe a property that intuitively requires a *fast path* for transactions: transactions must terminate quickly in executions in which they do not encounter asynchrony, failures, or conflicts. While many fast-path properties have been formulated in the literature for consensus algorithms, transactions are more complex since different transactions may require a different number of network round trips, or message delays, in order to even know what data they should access. We capture this variability in a property we name *fast decision*, intuitively requiring that once the data set of a transaction is known, it must reach a decision within one network round trip. Finally, we present a property called *seamless fault tolerance*, which requires an algorithm to be able to tolerate some failures without affecting the latency of ongoing transactions. This has been the goal of many recent works which focus on robustness and high availability [47, 37, 38, 29, 54].

Equipped with these properties, we then show the inherent tension that exists between them and the well-known multicore properties of disjoint-access parallelism and invisible reads, both of which intuitively improve cache coherence and have been shown to increase scalability in transactional systems [42, 22]. More precisely, we present the **FIDS** theorem for *sharded* PDTSs: a PDTS that guarantees a minimal progress condition and shards data across multiple nodes cannot simultaneously provide **F**ast decision, **I**nvisible reads, distributed **D**isjoint-access parallelism, and **S**erializability. An important implication of this result is that serializable shared-memory sharded PDTSs that want to provide multicore scalability cannot simply use a two-phase atomic commitment protocol (such as the popular two-phase commit). Furthermore, we turn our attention to *replicated* PDTSs. We discover that a similar tension exists for PDTSs that utilize *client-driven* replication. With client-driven replication replicas do not need to communicate with each other to process transactions. It is commonly used in conjunction with a leaderless replication algorithm to save two message delays [29, 54, 47, 38], as well as in RDMA-based PDTSs which try to bypass the replicas' CPUs [16, 44]. We present a *robust* version of the FIDS theorem, which we call the **R-FIDS** theorem: a PDTS (that may or may not shard its data) and utilizes client-driven replication cannot simultaneously provide **R**obustness to failures in the form of seamless fault tolerance, **F**ast decision, **I**nvisible reads, **D**isjoint-access parallelism, and **S**erializability.

Interestingly, similar impossibility proofs appear in the literature, often showing properties of parallel transactional systems that cannot be simultaneously achieved [41, 7, 12]. Indeed, some works have specifically considered disjoint-access parallelism and invisible reads, and shown that they cannot be achieved simultaneously with strong progress conditions [7, 41]. However, several systems achieve both disjoint-access parallelism and invisible reads with weak progress conditions such as the one we require [53, 48, 16]. To the best of our knowledge, the two versions of the FIDS theorem are the first to relate multicore scalability properties to multinode scalability ones.

Finally, we show that the FIDS theorems are minimal in the sense that giving up any one of these properties does allow for implementations that satisfy the rest.

In summary, our contributions are as follows.

- We present a transactional model that combines the distributed and parallel settings.
- We formalize three distributed performance properties that have appeared in intuitive forms in the literature.
- We present the FIDS and R-FIDS theorems for parallel distributed transactional systems, showing that there are inherent tensions between multicore and multinode scalability properties.
- We show that giving up any one of the properties in the theorems does allow designing implementations that satisfy the rest.

The rest of the paper is organized as follows. Section 2 presents the model and some preliminary notions. In Section 3, we define the properties of distributed transactional systems that we focus on. We present our impossibility results in Section 4, and then in Section 5, we show that it is possible to build a PDTS that sacrifices any one of the properties. Finally, we discuss related works in Section 6 and future research directions in Section 7.

2 Model and Preliminaries

Communication. We consider a *message-passing* model among n *nodes* (server hosts) and any number of *client processes*, as illustrated in Figure 1. Each node has P *node processes*. Messages are sent either between two nodes or between clients and nodes. We consider *partial synchrony* [18]; messages can be arbitrarily delayed until an a priori unknown *global stabilization time (GST)*, after which all messages reach their target within a known delay Δ. An execution is said to be *synchronous* if GST is at the beginning of the execution. Furthermore, node processes within a single node communicate with each other via *shared*

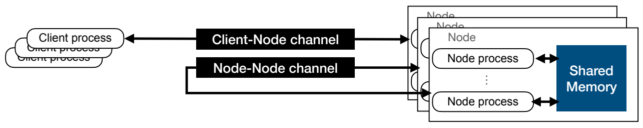


Figure 1 Communication mediums between the different types of processes considered in our model.

memory. That is, they access *shared base objects* through *primitive operations*, which are atomic operations, such as read, write, read-modify-write (compare-and-swap, test-and-set, fetch-and-increment, etc.), defined in the usual way. A primitive operation is said to be *non-trivial* if it may modify the object. Two primitive operations *contend* if they access the same object and at least one of them is non-trivial. The order of accesses of processes to memory is governed by a *fair scheduler* which ensures that all processes take steps.

Transactions. We consider a database composed of a set of *data items*, Σ, which can be accessed by *read* and *write* operations. Each node N_i holds some subset $\Sigma_i \subseteq \Sigma$, which may overlap with the subsets held on other nodes. A *transaction* T is a program that executes read and write operations on a subset of the data items, called its *data set*, $D_T \subseteq \Sigma$. A transaction T's *write set*, $W_T \subseteq D_T$, and *read set*, $R_T \subseteq D_T$, are the sets of data items that it writes and reads, respectively. Two transactions are said to *conflict* if their data sets intersect at an item that is in the write set of at least one of them.

Transaction Interface. An *application* may execute a transaction T by calling an *invokeTxn(T)* procedure. The invokeTxn(T) procedure returns with a *commit* or *abort* value indicating whether it committed or aborted, as well as the full read and write sets of T, with the order of execution of the operations (relative to each other), and with the read and written values. We say that a transaction is *decided* when invokeTxn(T) returns.

Failure Model. Nodes can fail by crashing; if a node crashes then *all* processes on the node crash as well. We do not consider failures where individual processes crash and we assume clients do not fail. We denote by *failure-free* execution an execution without node crashes.

Handlers and Implementations. An *implementation* of a PDTS provides data representation for transactions and data items, and algorithms for two types of handlers: the *coordinator handler* and the *message handler*. Each handler is associated with a transaction and is executed by a single process. Each process executes at most one handler at any given time, and is otherwise *idle*. The coordinator handler of a transaction T is the first handler associated with T and is triggered by an invokeTxn(T) call on some client process.

The execution of a handler involves a sequence of *handler steps*, which are of one of three types: (1) an *invocation* or *response* step, which is the first or last step of the handler respectively, (2) a primitive operation on a base object in shared memory, including its return value, and (3) sending or receiving a message, denoted send(T, m) or receive(T, m). Each handler step is associated with the corresponding transaction and the process that runs it. The return value in a response step of a transaction's coordinator handler is the return value of invokeTxn described above, and a message handler has no return value.

Executions. An *execution* of a PDTS implementation is a sequence of handler steps and *node crash steps*. Each node crash step is associated with a node. After a node crash step associated with node N_i in execution E, no process on node N_i takes any steps in E. An execution can interleave handler steps associated with different transactions and processes. An *extension* E' of E is an execution that has E as its prefix.

We say that a transaction T's *interval* in an execution E begins at the invocation step of T's coordinator handler, and ends when there are no sends associated with T that have not been received whose target node has not crashed, and all handlers associated with T have reached their response step. Note that the end of a transaction's interval must therefore be a response step of some handler associated with T, but might not be the response step of T's coordinator handler (which may terminate earlier than some other handlers of T). We say that two transactions are *concurrent* in E if their intervals overlap. We say that two transactions, T_1 and T_2, *contend on node* N_i in E if they are concurrent, and there is at least one primitive operation step on node N_i in E associated with T_1 that contends with a primitive operation step in E associated with T_2. We say that T_1 and T_2 *contend* in E if there is some node N_i such that they contend on node N_i in E.

The *projection* of an execution E on a process p, denoted $E|p$, is the subexecution of E that includes exactly all of the steps associated with p in E. Two executions E and E' are *indistinguishable* to a process p if the projections of E and E' on p are identical (i.e., if $E|p = E'|p$).

It is also useful to discuss *knowledge* of properties during an execution. The notion of knowledge has been extensively used in other works [24, 20]. Formally, a process p *knows* a property P in an execution E of a PDTS implementation I, if there is no execution E' of I that is indistinguishable to p from E in which P is not true.

We adopt two concepts introduced by Lamport [35, 31] to aid reasoning about distributed systems: *depth of a step*, and the *happened-before* relation. The *depth of a step* s associated with transaction T in execution E is 0 if s is the invocation of T's coordinator handler. Otherwise, it equals the maximum of (i) the depths of all steps that are before s in E within the same handler as s, and (ii) if s is a receive(T, m) step of a message sent in a send(T, m) step, s', then 1 plus the depth of s'. *Happened-before* is the smallest relation on the set of steps of an execution E satisfying the following three conditions: 1) if a and b are steps of the same handler and a comes before b in E, then a *happened-before* b; 2) if a is a send(T, m) step and b is a receive(T, m) step, then a *happened-before* b; 3) if a *happened-before* b and b *happened-before* c, then a *happened-before* c.

Serializability. Intuitively, a transactional system is *serializable* if transactions appear to have executed in some serial order [40]. The formal definition appears in the full version of this paper [9].

Weak Progress. A transactional system must guarantee at least *weak progress*: every transaction is eventually *decided*, and every transaction that did not execute concurrently with any other transaction eventually *commits*.

2.1 Multicore Scalability Properties

To scale to many processes on each server node, transactional systems should reduce memory contention between different transactions. This topic has been extensively studied in the literature on parallel transactional systems [7, 6, 5, 11, 13, 23, 26, 41]. Here, we focus on two well-known properties, disjoint-access parallelism and invisible reads, that are known to reduce contention and improve scalability in parallel systems. We later show how they interact with distributed scalability properties.

2.1.1 Disjoint-Access Parallelism

Originally introduced to describe the degree of parallelism of implementations of shared memory primitives [26], and later adapted to transactional memory, *disjoint-access parallelism* intuitively means that transactions that are disjoint at a high level, e.g., whose data sets do not intersect, do not contend on shared memory accesses [7, 41]. While this property may sound intuitive, it can in fact be difficult to achieve, as it forbids the use of global locks or other global synchronization mechanisms. Multiple versions of disjoint-access parallelism exist in the literature, differing in which transactions are considered to be disjoint at a high level. In this paper, we use the following definition.

▶ **Definition 2.1** (Disjoint-access parallelism (DAP)). *An implementation of a PDTS satisfies* disjoint-access parallelism (DAP) *if two transactions whose data sets do not intersect cannot contend.*

2.1.2 Invisible Reads

The second property we consider, *invisible reads*, intuitively requires that transactions' read operations not execute any shared memory writes. This property greatly benefits workloads with read hotspots, by dramatically reducing cache coherence traffic. Two variants of this property are common in the literature. The first, which we call *weak invisible reads*, only requires invisible reads at the granularity of transactions. That is, if a transaction is read-only (i.e., its write set is empty), then it may not make any changes to the shared memory. This simple property has been often used in the literature [7, 41].

▶ **Definition 2.2** (Weak invisible reads). *An implementation of a PDTS satisfies* weak invisible reads *if, in all its executions, every transaction with an empty write set does not execute any non-trivial primitives.*

However, this property is quite weak, as it says nothing about the number of shared memory writes a transaction may execute once it has even a single item in its write set. When developing systems that decrease coherence traffic, this is often not enough. Indeed, papers that refer to invisible reads in the systems literature [47, 48] require that no read operation in the transaction be the cause of shared memory modifications. Note that an algorithm that locally stores the read set for validation (which is the case in the above referenced systems) can still satisfy invisible reads, since the writes are not to shared memory. Attiya et al. [6] formalize this stronger notion of invisible reads by requiring that we be able take an execution E and replace any transaction T in E with a transaction that has the same write set but an empty read set, and arrive at an execution that is indistinguishable from E. Intuitively, this captures the requirement that reads should not update shared metadata (e.g., through "read locks"). We adopt Attiya et al.'s definition of invisible reads here, adapted to fit our model.

▶ **Definition 2.3** (Invisible reads (adapted from [6])). *An implementation I of a PDTS satisfies the* invisible reads *property if it satisfies weak invisible reads and, additionally, for any execution E of I that includes a transaction T with write set W and read set R, there exists an execution E' of I identical to E except that it has no steps of T and it includes steps of a transaction T', which has the same interval as T (i.e., T's first and last steps in E are replaced by T''s ones in E'), and writes the same values to W in the same order as in T, but has an empty read set.*

Note that the invisible reads property complements the DAP property for enhanced multicore scalability. A system that has both allows all transactions that do not conflict, not just the disjoint-access ones, to proceed independently, with no contention (as we will show in Lemma 4.4). Interestingly, previous works discovered some inherent tradeoffs of such systems [7, 41], in conjunction with strong progress guarantees. In this paper, we study these properties under a very weak notion of progress, but with added requirements on distributed scalability (see Section 3).

3 Multinode Performance Properties

To overcome the limitations of a single machine (e.g., limited resources, lack of fault tolerance), distributed transactional systems shard or replicate the data items on multiple nodes, and, thus, must incorporate distributed algorithms that coordinate among multiple nodes. The performance of these distributed algorithms largely depends on the number of communication rounds required to execute a transaction. Ideally, at least in the absence of conflicts, transactions can be executed in few rounds of communication, even if some nodes experience failures. In this section we propose formal definitions for a few multinode performance properties.

3.1 Distributed Disjoint-Access Parallelism

We start by proposing an extension of DAP to distributed algorithms, which we term *distributed-DAP*, or DDAP. In addition to requiring DAP, DDAP proscribes transactions from contending on a node unless they access common elements that reside at that node:

▶ **Definition 3.1** (Distributed disjoint-access parallelism (DDAP)). *An implementation of a PDTS satisfies* distributed disjoint-access parallelism (DDAP) *if for any two transactions T and T', and any node N_i, if T and T''s data sets do not intersect on node N_i (i.e., $D_T \cap D_{T'} \cap \Sigma_i = \emptyset$), then they do not contend on node N_i.*

While the main goal of sharding is to distribute the workload across nodes, DDAP links sharding to increased parallelism – DDAP systems can offer more node parallelism than DAP systems through sharding.

3.2 Fast Decision

Distributed transactional systems must integrate agreement protocols (such as *atomic commitment* and *consensus*) to ensure consistency across all nodes involved in transaction processing. Fast variants of such protocols can reach agreement in two message delays in "good" executions [35]. Ideally, we would like distributed transactional systems to preserve this best-case lower bound, and decide transactions in two message delays; reducing the number of message delays required to process transactions not only can significantly reduce the latency as perceived by the application (processing delay within a machine is usually smaller than the delay on the network), but can also reduce the *contention footprint* [21] of the transactions (intuitively, this is the duration of time in which a transaction might interfere with other transactions in the system).

Requiring transactions to be decided in just two message delays, however, is too restrictive in many scenarios. The latency of a distributed transactional system depends on how many message delays are required for a transaction to "learn" its data set (data items and their values); the data set needs to be returned to the application when the transaction commits, and is also used to determine whether the transaction can commit. For example, for interactive transactions or disaggregated storage, the values must be made available to the application (which runs in a client process) before the transaction can continue to execute. Thus, since the data items are remote, each read operation results in two message delays, one to request the data from the remote node and one for the remote node to reply. For non-interactive transactions or systems where transaction execution can be offloaded to the node processes, the latency for learning the data set can be improved; since the client does not need to immediately know the return value of read operations, the values of data items can be learned through a chain of messages that continue transaction processing at the nodes containing the remote data. More precisely, the client first determines a node, n_1, that contains the first data item the transaction needs to read; the client sends a message to n_1 containing the transaction; n_1 processes the transaction, preforming the read locally, until it determines that the transaction needs to perform a remote read from another node, n_2; n_1 sends a message to n_2 containing the transaction and its state so far; n_2 continues processing the transaction, performing the read locally, and so on. RPC chains [45] already provides an implementation of this mechanism, saving one message delay per remote read operation. At the lowest extreme, non-sharded transactional systems can learn a transaction's entire data set in a single message delay.

We introduce the *fast decision* property to describe distributed transactional systems that can decide each transaction in "good" executions within only two message delays in addition to the message delays it requires to "learn" the transaction's entire data set. As explained above, the number of message delays required to learn a transaction's data set depends on several design choices. We note that often, deciding a transaction's outcome within two message delays after learning its data set is not plausible if the execution has suboptimal conditions, for example, if there are transactional conflicts that need to be resolved, or if not all nodes reply to messages within some timeout. This is true even for just consensus, where the two-message-delay decisions can happen only in favorable executions, on a *fast path* [3, 34]. We therefore define the fast decision property to only be required in such favorable executions.

To formalize fast decisions, we must be able to discuss several intuitive concepts more formally. In particular, we begin by defining the *depth of a transaction*, to allow us to formally discuss the number of message delays that the transactional system requires to decide a transaction.

▶ **Definition 3.2** (Depth of a transaction). *The* depth of a decided transaction T *in execution* E *of a PDTS implementation,* $d_E(T)$*, is the depth of the response step of* T*'s coordinator handler in* E*.*

In many cases, we need to refer to the depth of a transaction T in an execution in which T is still ongoing, and its coordinator handler has not reached its response step yet. While we could simply refer to the depth of the deepest step of T in the execution, this would not be appropriate: it is possible that a transaction in fact took steps along one "causal path" that led to a large depth, but when the response step to T's coordinator handler happens, its depth is actually shorter. In such a case, we really only care about the depth along the "causal paths" that lead to the response step, since these are the ones affecting the latency to the application. To capture this notion, we define the *partial depth* of a transaction T in a prefix of an execution in which T is decided as follows.

▶ **Definition 3.3** (Partial depth of a transaction). *Let* T *be a decided transaction in execution* E *of a PDTS implementation. The* partial depth *of* T *in a prefix* P *of* E *in which* T *is not decided,* $d_E(T, P)$*, is the maximum step depth across all steps associated with* T *in* P*, which happened-before the response step of* T*'s coordinator handler in* E *(or* 0 *if there are no such steps).*

We next formalize another useful concept that we need for the discussion of fast decisions; namely, what it means to *learn* the data set of a transaction. For that purpose, we introduce the following two definitions:

▶ **Definition 3.4** (Decided data item). *A data item d is* decided *to be in a transaction T's read or write set in execution E of a PDTS implementation if, in all extensions of E, the read or write set respectively in the return value of invokeTxn(T) contains d.*

▶ **Definition 3.5** (Decided value). *A data item d's* value *is* decided *for T in execution E of a PDTS implementation if, in all extensions of E, the read set in the return value of invokeTxn(T) contains d and with the same value.*

Note that a data item's value can be decided for a transaction only if that data item is part of its read set; the definition does not apply for data items in the write set. In the definition of the fast decision property and in the proofs, we refer in most places to knowing the decided values and not the data items in the write set as well. This is because knowing the read set and its values implies that a transaction's write set is decided in case it commits; this is the property that matters in many of the arguments we use in the paper.

Finally, we are ready to discuss the *fast decision* property. Intuitively, the formal definition of the property considers *favorable* executions, which are synchronous, failure-free and have each transaction run solo. For those executions, the property requires two things to hold: first, a transaction is not allowed to spend more than two message delays without learning some new value for its data set, and second, once its entire data set is known, it must be decided within 2 more message delays (Corollary 3.7). This captures "speed" in both learning the data set and deciding the transaction outcome. As discussed above, 2 message delays is an upper bound on the minimal amount of time needed to perform a read operation (and bring its value to the necessary process). Note that this is a tight bound for systems processing interactive transactions, and as such, fast decision also means optimal latency for these systems.

▶ **Definition 3.6** (Fast decision). *A PDTS implementation I is* fast deciding *if, for every failure-free synchronous execution E of I and every decided transaction T in E that did not execute concurrently with any other transaction, for any prefix P of E such that $d_E(T,P) < d_E(T) - 2$, there exists a prefix of E of partial depth $d_E(T,P)+2$ in which the number of values known by some process to be decided is bigger than in P.*

Formalizing the allowed depth of a transaction in terms of prefixes of an execution in which the transaction is already decided (so we know its depth in that execution) helps capture the two requirements we want: (1) for any prefix of the execution, if we advance from it by two message delays, we must have improved our knowledge of the values of the read set, and (2) once the read set and its values are completely known (regardless of the depth of the prefix in which this occurs), we must be at most 2 message delays from deciding that transaction. Corollary 3.7 helps make this intuition concrete.

▶ **Corollary 3.7.** *For every failure-free synchronous execution E of a fast-deciding PDTS implementation I and every decided transaction T in E that did not execute concurrently with any other transaction, let P be the shortest prefix of E in which the value of each item in T's read set is known by some process to be decided. Then*

$$d_E(T) \leq d_E(T,P) + 2.$$

(Intuitively, T must be decided within at most 2 message delays from when T's read set including its values are known to be decided.)

Proof. Assume by contradiction that $d_E(T) > d_E(T, P) + 2$. Then by the fast decision property of I, there exists a prefix of E in which the number of data items whose value is known by some process to be decided is bigger than in P. But this is impossible, since the values of T's entire read set are known to be decided in P. ◀

Several fast-deciding distributed transactional systems have been recently proposed for general interactive transactions [29, 54, 47]; our fast decision property captures what they informally refer to as "one round-trip commitment". These systems use an *optimistic concurrency control* and start with an execution phase that constructs their data sets with two message delays per read operation. The agreement phase consists of validation checks that require a single round-trip latency (integrates atomic commitment and a fast consensus path in one single round trip). The write phase happens asynchronously, after the response of the transaction has been emitted to the application.

3.3 Seamless Fault Tolerance

High availability is critical for transactional storage systems, as many of their applications expect their data to be always accessible. In other words, the system must mask server failures and network slowdowns. To achieve this, many systems in practice are designed to be fault tolerant; the system can continue to operate despite the failures of some of its nodes.

However, oftentimes, while the system can continue to function when failures occur, it experiences periods of unavailability, or its performance degrades by multiple orders of magnitude while recovering [3, 51]. This is the case in systems that must manually reconfigure upon failures [50], and those that rely on a leader [3, 39, 51, 33, 32].

These slow failure-recovery mechanisms, while providing some form of guaranteed availability, may not be sufficient for systems in which high availability is truly critical; suffering from long periods of severe slowdowns potentially from a single server failure may not be acceptable in some applications.

To address this issue, some works in recent years have focused on designing algorithms that experience minimal slowdowns, or no slowdowns at all, upon failures. One approach has been to minimize the impact of leader failures by making the leader-change mechanism lightweight and switching leaders even when failures do not occur [52]. Another approach aims to eliminate the leader completely; such algorithms are called *leaderless* algorithms [4, 47, 54, 37]. All of these approaches aim to tolerate the failure of some nodes without impacting the latency of ongoing transactions.

In this paper, we formalize this goal of tolerating failures without impacting latency into a property that we call *s-seamless fault tolerance*, where $s \leq f$. In essence, s-seamless fault tolerance requires that if only up to s failures occur in an execution, no slowdown is experienced. To capture this formally, we require that for any execution E with up to $s - 1$ crashes, it be possible to find an equivalent execution E' with one more crash event, which may happen at any time after the crashes in E, where the depth of all transactions are the same in E and E'. We express this in an inductive definition.

▶ **Definition 3.8** (s-seamless fault tolerance). *Any implementation of a PDTS satisfies* 0-*seamless fault tolerance. An implementation I of a PDTS satisfies* s-seamless fault tolerance *if it satisfies* $(s-1)$*-seamless fault tolerance, and for any execution E of I with $s-1$ node crashes, for any prefix E_P of E that contains the $s-1$ node crashes, and any node crash event c of a node that has not crashed in E_P, there exists an execution E' of I whose prefix is $E_P \cdot c$, such that (1) stripping each of E and E' of all steps other than invocation and response steps of coordinator handlers results in the same sequence of invocation and response steps (intuitively, the executions are equivalent), and (2) the depth of each decided transaction is the same in both executions (intuitively, E' seamlessly tolerates the node crashes).*

While s-seamless fault tolerance offers the extremely desirable robustness property, it also requires that: a) no single node can be on the critical path of all transactions, and b) no single node can be solely responsible for processing a transactional task. This can be a double-edged sword; on the one hand, this eliminates the possibility of a leader bottleneck, which implies better scalability. On the other hand, it disallows certain optimizations, like reading from a single replica.

4 Impossibility Results

Having specified some key properties which make distributed transactional systems fast and scalable, we now turn to the main result of our paper: unfortunately, there is a tension between these multinode performance properties and the single-node multicore performance properties discussed in Section 2. More specifically, we present the **FIDS** theorems, which formalize the impossibility of achieving all of these properties simultaneously in two different parallel distributed settings.

4.1 The FIDS Theorems

The first **FIDS** theorem states that no PDTS with weak progress which *shards data* can guarantee **F**ast decision, **I**nvisible reads, distributed **D**isjoint-access parallelism, and **S**erializability simultaneously. This is in contrast to known systems that achieve just the multinode properties [47, 54, 38] or just the multicore properties [53, 48, 16]. Thus, the FIDS theorem truly shows tensions that arise when a transactional system is both parallel and distributed. This version of the FIDS theorem considers only systems that shard data, that is, systems in which each node only stores part of the database items. Interestingly, the impossibility holds in this setting even without requiring any fault tolerance, and in particular, without seamless fault tolerance. We note that the FIDS theorem applies also to systems that replicate data in addition to sharding it; adding replication on top of a sharded system only makes it more complex. Formally:

▶ **Theorem 4.1** (The FIDS theorem for sharded transactional systems). *There is no implementation of a PDTS which shards data across multiple nodes that guarantees weak progress, and simultaneously provides fast decision, invisible reads, distributed disjoint-access parallelism, and serializability.*

For systems that maintain multiple copies of the data, but do not necessarily shard it, we show a different version of the result. Note that in such systems, distribution comes from replication; several nodes, each with a copy of the entire database, are used to ensure fault tolerance. For this setting, we present the Robust-FIDS, or **R-FIDS**, theorem: a PDTS with weak progress that utilizes client-driven replication and satisfies **R**obustness to at least one failure through the seamless fault tolerance property, in addition to satisfying **F**ast decision, **I**nvisible reads, **D**isjoint-access parallelism, and **S**erializability, is also impossible to implement. Formally:

▶ **Theorem 4.2** (The R-FIDS theorem for replicated transactional systems). *There is no implementation of a PDTS that utilizes client-driven replication that guarantees weak progress, and simultaneously provides* 1*-seamless fault tolerance, fast decision, invisible reads, disjoint-access parallelism, and serializability.*

In the reminder of this section we present an overview of the proof technique for the two versions of the FIDS theorem; the detailed proof for each of them and the supporting lemmas we introduce here can be found in Appendix A.

4.2 Proof Overview

Both proofs have a similar structure; we consider example transactions that form a dependency cycle, and show an execution in which all of them commit, thereby violating serializability. To argue that all transactions in our execution commit, we build the execution by merging executions in which each transaction ran solo (and therefore had to commit by weak progress), and showing that the resulting concurrent execution is indistinguishable to each transaction from its solo run. Starting with solo executions also gives us another property that we can exploit; we define the solo executions to be synchronous and failure-free, and therefore they must be fast deciding as well.

The key challenge in the proofs is how to construct a concurrent execution E_{concur} that remains indistinguishable to all processes from the solo execution that they were a part of. To do so, we divide the concurrent execution into two phases; first, we let the solo executions run, in any interleaving, until right before the point in each execution at which some process learns the values of its transaction's read set. When this point is reached in each solo execution, we carefully interleave the remaining steps in a second phase of the concurrent execution. A key feature is that by the fast decision property, which each solo execution satisfies, once some process learns the read set including its values, there are at most two message delays left in each solo execution before the transaction is decided. This bounds the amount of communication we need to worry about in the second phase of the concurrent execution.

To show that E_{concur} is indistinguishable from the solo runs, we look at each of the two phases separately. The idea is to show that no process makes *any* shared memory modifications in the first phase, and then show that we can interleave messages and message handlers in a way that allows each transaction to be oblivious to the other transactions for at least one more intuitive "round trip", which is all we need to reach decision according to the fast decision property.

To show that a transaction performs no shared memory modifications in the first phase of the concurrent execution we construct, we rely on the way we choose the transactions, their data sets, and when in the execution their data sets are decided; in both proofs, the transactions we choose may have empty or non-empty write sets, depending on the results of their reads. The following lemma shows that as long as a transaction's write set is not *known* to be non-empty, the transaction cannot cause any modifications in a system that provides weak invisible reads.

▶ **Lemma 4.3.** *Let I be an implementation of a PDTS that provides weak invisible reads, and let T be a transaction in an execution E of I, such that no process in E knows the following proposition: T's write set is non-empty in all extensions of this execution in which T is decided. Then T cannot cause any base object modifications in E.*

This lemma, combined with the way we choose the transactions in our proofs, immediately implies that phase 1 of E_{concur} is indistinguishable to all processes from the solo executions they are a part of.

The proofs differ somewhat in how they show that E_{concur} is indistinguishable from the solo runs in the second phase. We argue about restricted shared memory modifications through the use of the DAP and invisible reads properties in the following key lemma, which intuitively shows that transactions that do not conflict do not (visibly) contend.

▶ **Lemma 4.4.** *Let I be an implementation of a PDTS that provides both DAP and invisible reads, and let T be a transaction in an execution E of I, such that its final write set is W. Then T does not cause any base object modifications visible to any concurrent transaction in E whose data set does not overlap with W.*

To make phase 2 of E_{concur} also indistinguishable from the solo executions, we schedule the remaining messages carefully. In particular, we schedule messages sent by reading transactions to each node before those sent by writing transactions, and again rely on DAP and invisible reads to argue that the reading transactions' handlers will not cause changes visible to those who write afterwards. However, here the two proofs diverge.

4.2.1 Sharded Systems

We first discuss the proof structure for showing that serializable sharded transactional systems that provide weak progress cannot simultaneously achieve fast decision, invisible reads and DDAP. That is, sharding the data across multiple nodes while achieving these properties is impossible even if we do not tolerate any failures (Theorem 4.1).

The proof uses two nodes and two transactions, each reading from a data item on one node and, if it sees the initial value, writing on the other node. The read set of one transaction is the same as the (potential) write set of the other transaction. We need to argue that the reading transaction on some node cannot cause modifications on that node that are visible to the writing transaction. However, since the write set of each transaction overlaps with the data set of the other, we cannot apply Lemma 4.4. Instead, we rely on DDAP, and show that with this property, the reading transaction indeed cannot be visible to the writing one on each node. We show a lemma very similar to Lemma 4.4 but which applies to transactions whose write set on a specific node does not overlap the data set of another transaction on that node.

▶ **Lemma 4.5.** *In any implementation of a PDTS that provides both DDAP and invisible reads, a transaction whose write set is W does not cause any modifications on shared based objects on a node N visible to any concurrent transaction whose data set does not overlap with W on N.*

The proof of this lemma is very similar to the proof of Lemma 4.4. The only required adjustments are using *DDAP* instead of DAP, and referring to T''s data set and T's write set and modification *on a certain node N*.

Note that while the proof of the FIDS theorem relies on sharding, it does not need fault tolerance. In particular, it does not make use of the seamless fault tolerance property. However, the result does apply to systems in which the data is both sharded and replicated, as those systems are even more complex than ones in which no replication is used.

4.2.2 Replicated but Unsharded

So far, we have considered a PDTS in which node failures cannot be tolerated; if one of the nodes crashes, we lose all data items stored on that node, and cannot execute any transactions that access those data items. However, in reality, server failures are common, and therefore many practical systems use replication to avoid system failures. Of course, the impossibility result of Theorem 4.1 holds for a PDTS even for the more difficult case in which failures are possible and each node's data is replicated on several backups.

However, we now turn our attention to PDTSs in which the entire database is stored on each node. This setting makes it plausible that a client could get away with accessing only one node to see the state of the data items of its transaction. However, we show that the impossibility of Theorem 4.1 still holds in this setting for a system in which failures are tolerated without affecting transaction latency (i.e., systems that satisfy seamless fault tolerance) (Theorem 4.2).

As explained in Section 4.2, the use of seamless fault tolerance requires us to explicitly argue about the length of the executions in which transactions decide. To do so, we need the following lemma, which gives a lower bound for the depth at which a transaction's read set and values can be decided.

▶ **Lemma 4.6.** *There is no execution E of any serializable PDTS implementation that tolerates at least 1 failure in which there is a transaction T and prefix P such that $d_E(T, P) < 2$ and some process knows the decided value of some read of T in P.*

Once we have this lemma, the proof of the R-FIDS theorem is then similar to the proof of the FIDS theorem. We build a cycle of dependencies between transactions where each neighboring pair in the cycle overlaps on a single data item that one of them reads and the other writes. The key is that because of invisible reads, each read can happen before the write on the same data item without leaving a trace. However, to construct this cycle in the replicated case, we need at least 3 replicas, 3 transactions and 3 data items. This is because we can no longer separate the read and write of a single transaction on each node. Furthermore, we make use of Lemma 4.6, as well as the budgeted depth of a transaction in a fast-deciding execution, to explicitly argue about the amount of communication possible after a transaction learns its write set.

More specifically, we choose three transactions, where the write set of one equals the read set of the next. We divide them into pairs, where within each pair, the write set of one does not overlap with the data set of the other. We can then directly use Lemma 4.4 to argue that the second one to be scheduled of this pair will not see changes made by the first. We exploit fault tolerance to have the third transaction's messages never reach that node. However, here, we must be careful, since we defined the solo executions to be failure-free to guarantee fast decisions. We therefore rely on seamless fault tolerance; we show indistinguishability of the concurrent execution not from the original solo executions, but from executions of the same depth that we know exist due to seamless fault tolerance.

Interestingly, when we convert a solo execution S to an execution F of the same depth (but with a node failure) via the seamless fault tolerance property, we may lose its fast decision property. That is, while the new execution must have the same depth as the original ones, that does not guarantee that it will also be fast deciding, as the fast decision property does not solely refer to the length of the execution. In particular, it could be the case that in F, the data set of a transaction including its values is learned earlier, but then the transaction takes more than 2 message delays to be decided. This would be problematic for our proof, in which the indistinguishability relies heavily on fast decision once the data set including its values are known. To show that this cannot happen in the executions we consider, we rely on Lemma 4.6 that bounds the depth at which any transaction in a fault tolerant system can learn the decided values of its reads.

5 Possibility Results

Any subset of the properties outlined in Theorem 4.2 *is* possible to achieve simultaneously in a single system. Due to lack of space, we show this in the full version of this paper [9], where we present four distributed transactional system algorithms, each sacrificing one of the desired properties. Recall from our model description that the presented protocols work under the assumption that the client does not fail and nodes do not recover, and as such are not intended to be used "as is" in practice. We first present a "base" algorithm which achieves all the desired properties (i.e., fast decision, invisible reads, DDAP, and

1-seamless fault tolerance), but is not serializable. We obtain each of the four transactional systems algorithms, by tweaking the base algorithm to sacrifice one desired property and gain serializability.

6 Related Work

Disjoint-access parallelism was first introduced in [26] in the context of shared memory objects. It was later adapted to the context of transactions. Over the years, it has been extensively studied as a desirable property for scalable multicore systems [53, 49, 47, 7, 5, 41, 23]. Several versions of DAP have been considered, differing in what is considered a conflict between operations (or transactions). A common variant of DAP considers two transactions to conflict if they are connected in the conflict graph of the execution (where vertices are transactions and there is an edge between two transactions if their data sets intersect) [7, 41]. In this paper, we consider a stricter version, which only defines transactions as conflicting if they are neighbors in the conflict graph. This version has also appeared frequently in the literature [41, 12].

Invisible reads have also been extensively considered in the literature [43, 7, 47, 49, 53, 25]. Many papers consider invisible reads on the granularity of data item accesses; any read operation on a data item should not cause changes to shared memory [49, 47, 25]. Others, often those that study invisible reads from a more theoretical lens, consider only the invisibility of *read-only transactions* [7, 41].

Some impossibility tradeoffs for transactional systems, similar to the one we show in this paper, are known in the literature. Attiya et al. [7] show that it is impossible to achieve weak invisible reads, disjoint-access parallelism, and wait-freedom in a parallel transactional system. Peluso et al. [41] show an impossibility of a similar setting, with disjoint-access parallelism, weak invisible reads, and wait-freedom, but consider any correctness criterion that provides real-time ordering. Bushkov et al. [12] show that it is impossible to achieve disjoint-access parallelism and obstruction-freedom, even when aiming for consistency that is weaker than serializability. In this paper, none of our algorithms provide the obstruction-freedom considered in [12]; we use locks, and our algorithms can therefore indefinitely prevent progress if process failures can occur while holding locks.

Fast paths for *fast decision* have been considered extensively in the replication and consensus literature [28, 17, 34, 2, 3]. In most of these works, the conditions for remaining on the fast path include experiencing no failures. That is, they do not provide seamless fault tolerance. However, some algorithms, like Fast Paxos [34], can handle some failures without leaving the fast path. In the context of transactional systems, the fast path is often considered for conflict-free executions rather than those without failures [47, 54, 38, 29], as we do in this paper. Seamless fault tolerance captures the idea that (few) failures should not cause an execution to leave the fast path. Systems often have a general fault tolerance f that is higher than the number of failures they can tolerate in a seamless manner [47, 54, 38, 29].

Seamless fault tolerance as presented in this paper is also related to *leaderlessness* [37, 4], as any leader-based algorithm would slow down upon a leader failure. However, the leaderless requirement alone is less strict than our seamless fault tolerance; Antoniadis et al. [4] defined a leaderless algorithm as any algorithm that can terminate despite an adaptive adversary that can choose which process to temporarily remove from an execution at any point in time. This does not put any requirement on the speed at which the execution must terminate.

Parallel distributed transactional systems have been recently studied in the systems literature. Meerkat [47] provides serializability and weak progress, and three of the desirable properties we outline in this paper. It does not, however, provide invisible reads in any form (not even the weaker version). Eve [27] considers replication for multicore systems. It briefly outlines how PDTS transactions are possible using its replication system, but it is not their main focus.

7 Discussion

This paper is inspired by recent trends in network capabilities, which motivate the study of distributed transactional systems that also take advantage of the parallelism available on each of their servers. We formalize three performance properties of distributed transactional systems that have appeared intuitively in various papers in the literature, and show that these properties have inherent tensions with multicore scalability properties. In particular, in this paper we formalized the notions of distributed disjoint-access parallelism, a fast decision path for transactions, and robustness in the form of seamless fault tolerance. Combined with the well-known multicore scalability properties of disjoint-access parallelism and invisible reads, we show the *FIDS theorem*, and its fault tolerant version, the *R-FIDS theorem*, which show that serializable transactional systems cannot satisfy all these properties at once. Finally, we show that removing any one of these properties allows for feasible implementations.

We note that our possibility results can be seen as "proofs of concept" rather than practical implementations. It would be interesting to design practical algorithms that give up just one of the properties we discuss. We believe that each property has its own merit for certain applications and workloads, and it would be interesting to determine which property would be the best to abandon for which types of applications.

In this work, we focused on studying parallel distributed transactional systems under a minimal progress guarantee. It would also be interesting to explore PDTSs under stronger progress conditions, or consistency conditions other than serializability. It would be equally interesting to see if the tension still exists between weaker variants of the properties we considered.

References

1 Marcos K Aguilera, Naama Ben-David, Irina Calciu, Rachid Guerraoui, Erez Petrank, and Sam Toueg. Passing messages while sharing memory. In *Proceedings of the 2018 ACM symposium on principles of distributed computing*, pages 51–60, 2018.

2 Marcos K Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra Marathe, and Igor Zablotchi. The impact of RDMA on agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 409–418, 2019.

3 Marcos K Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J Marathe, Athanasios Xygkis, and Igor Zablotchi. Microsecond consensus for microsecond applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 599–616, 2020.

4 Karolos Antoniadis, Antoine Desjardins, Vincent Gramoli, Rachid Guerraoui, and Igor Zablotchi. Leaderless consensus. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 392–402. IEEE, 2021.

5 Hagit Attiya and Panagiota Fatourou. Disjoint-access parallelism in software transactional memory. In *Transactional Memory. Foundations, Algorithms, Tools, and Applications*, pages 72–97. Springer, 2015.

6 Hagit Attiya and Eshcar Hillel. The cost of privatization in software transactional memory. *IEEE Transactions on Computers*, 62(12):2531–2543, 2012.

7 Hagit Attiya, Eshcar Hillel, and Alessia Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. *Theory of Computing Systems*, 49(4):698–719, 2011.

8 Hillel Avni and Nir Shavit. Maintaining consistent transactional states without a global clock. In *Colloquium on Structural Information & Communication Complexity*, 2008.

9 Naama Ben-David, Gal Sela, and Adriana Szekeres. The FIDS Theorems: Tensions between Multinode and Multicore Performance in Transactional Systems, 2023. `arXiv:2308.03919`.

10 Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., USA, 1986.

11 Silas Boyd-Wickizer. *Optimizing communication bottlenecks in multiprocessor operating system kernels*. PhD thesis, Massachusetts Institute of Technology, 2014.

12 Victor Bushkov, Dmytro Dziuma, Panagiota Fatourou, and Rachid Guerraoui. The pcl theorem: Transactions cannot be parallel, consistent and live. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 178–187, 2014.

13 Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 1–17, New York, NY, USA, 2013. Association for Computing Machinery.

14 James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*, 2012.

15 James Cowling and Barbara H. Liskov. Granola: Low-Overhead Distributed Transaction Coordination. In *Proceedings of 2012 USENIX Annual Technical Conference (USENIX ATC'12)*, pages 223–236, 2012.

16 Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 54–70, New York, NY, USA, 2015. Association for Computing Machinery. `doi:10.1145/2815400.2815425`.

17 Partha Dutta, Rachid Guerraoui, and Leslie Lamport. How fast can eventual synchrony lead to consensus? In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 22–27. IEEE, 2005.

18 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

19 Mostafa Elhemali, Niall Gallagher, Nick Gordon, Joseph Idziorek, Richard Krog, Colin Lazier, Erben Mo, Akhilesh Mritunjai, Somasundaram Perianayagam, Tim Rath, Swami Sivasubramanian, James Christopher Sorenson III, Sroaj Sosothikul, Doug Terry, and Akshat Vig. Amazon DynamoDB: A scalable, predictably performant, and fully managed NoSQL database service. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022.

20 Ronald Fagin, Joseph Y Halpern, Yoram Moses, and Moshe Vardi. *Reasoning about knowledge*. MIT press, 2004.

21 Jose M. Faleiro, Alexander Thomson, and Daniel J. Abadi. Lazy evaluation of transactions in database systems. In *SIGMOD '14*, 2014.

22 Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. Time-based software transactional memory. *IEEE Transactions on Parallel and Distributed Systems*, 21(12):1793–1807, 2010.

23 Rachid Guerraoui and Michal Kapalka. On obstruction-free transactions. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 304–313, 2008.

24 Joseph Y Halpern and Yoram Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM (JACM)*, 37(3):549–587, 1990.

25 Maurice Herlihy, Victor Luchangco, Mark Moir, and William N Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, 2003.

26 Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '94, pages 151–160, New York, NY, USA, 1994. Association for Computing Machinery. doi:10.1145/197917.198079.

27 Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. All about eve: Execute-verify replication for multi-core servers. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 237–250, USA, 2012. USENIX Association.

28 Idit Keidar and Sergio Rajsbaum. On the cost of fault-tolerant consensus when there are no faults: preliminary version. *ACM SIGACT News*, 32(2):45–63, 2001.

29 Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-Data Center Consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 113–126, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2465351.2465363.

30 Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010. doi:10.1145/1773912.1773922.

31 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978. doi:10.1145/359545.359563.

32 Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

33 Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, 2001.

34 Leslie Lamport. Fast paxos. *Distributed Comput.*, 19(2):79–103, 2006. doi:10.1007/s00446-006-0005-x.

35 Leslie Lamport. Lower bounds for asynchronous consensus. *Distrib. Comput.*, 19(2):104–125, October 2006. doi:10.1007/s00446-006-0155-x.

36 Jialin Li, Ellis Michael, and Dan R. K. Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*, SOSP '17, 2017.

37 Iulian Moraru, David G Andersen, and Michael Kaminsky. Egalitarian paxos. In *ACM Symposium on Operating Systems Principles*, 2012.

38 Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. Consolidating concurrency control and consensus for commits under conflicts. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 517–532, USA, 2016. USENIX Association.

39 Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (Usenix ATC 14)*, pages 305–319, 2014.

40 Christos H Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM (JACM)*, 26(4):631–653, 1979.

41 Sebastiano Peluso, Roberto Palmieri, Paolo Romano, Binoy Ravindran, and Francesco Quaglia. Disjoint-access parallelism: Impossibility, possibility, and cost of transactional memory implementations. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 217–226, 2015.

42 Amitabha Roy, Steven Hand, and Tim Harris. Exploring the limits of disjoint access parallelism. In *Proceedings of the 1st USENIX Workshop on Hot Topics in Parallelism, Berkeley, CA*, 2009.

43 William N Scherer III and Michael L Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 240–248, 2005.

44 Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojević, Dushyanth Narayanan, and Miguel Castro. Fast general distributed transactions with opacity. In *SIGMOD'19*, 2019.

45 Yee Jiun Song, Marcos K. Aguilera, Ramakrishna Kotla, and Dahlia Malkhi. Rpc chains: Efficient client-server communication in geodistributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*, 2009.

46 Michael Stonebraker. The case for shared nothing. In *IEEE Database Eng. Bull.*, 1985.

47 Adriana Szekeres, Michael Whittaker, Jialin Li, Naveen Kr Sharma, Arvind Krishnamurthy, Dan RK Ports, and Irene Zhang. Meerkat: multicore-scalable replicated transactions following the zero-coordination principle. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–14, 2020.

48 Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 18–32, New York, NY, USA, 2013. Association for Computing Machinery. `doi:10.1145/2517349.2522713`.

49 Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32, 2013.

50 Robbert Van Renesse and Fred B Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, 2004.

51 Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. Apus: Fast and scalable paxos on rdma. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 94–107, 2017.

52 Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.

53 Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. TicToc: Time Traveling Optimistic Concurrency Control. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD'16)*, 2016.

54 Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 263–278, New York, NY, USA, 2015. Association for Computing Machinery. `doi:10.1145/2815400.2815404`.

55 Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. Foundationdb: A distributed unbundled transactional key value store. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD'21)*, 2021.

Appendix

A Full proofs of the FIDS theorems (and the supporting lemmas)

▶ **Lemma 4.3.** *Let I be an implementation of a PDTS that provides weak invisible reads, and let T be a transaction in an execution E of I, such that no process in E knows the following proposition: T's write set is non-empty in all extensions of this execution in which T is decided. Then T cannot cause any base object modifications in E.*

Proof. Let I be an implementation of a PDTS that satisfies weak invisible reads. Assume by contradiction that there is a transaction T in execution E of I, such that no process in E knows that T's final write set is not empty in all extensions in which T is decided, and a process p runs a handler associated with T that performs some base object modification.

Since p does not know that T's final write set is not empty in all extensions of the current execution in which T is decided, there exists an execution indistinguishable to p from E that has an extension in which T's final write set is empty. Let that extension be $E_{readOnly}$. Since T's final write set in $E_{readOnly}$ is empty, then by weak invisible reads, T cannot cause base object modifications in $E_{readOnly}$. Contradiction. ◀

▶ **Lemma 4.4.** *Let I be an implementation of a PDTS that provides both DAP and invisible reads, and let T be a transaction in an execution E of I, such that its final write set is W. Then T does not cause any base object modifications visible to any concurrent transaction in E whose data set does not overlap with W.*

Proof. Let I be an implementation of a PDTS that satisfies DAP and invisible reads. Let T be a transaction whose final write set in an execution E of I is W. Assume by contradiction that there exists some transaction T' concurrent with T in E whose data set does not overlap with W, but which sees a modification made by T in E. That is, there is some base object operation step s of T' whose return value is affected by T's modification.

By invisible reads, there exists an execution E' of I identical to E except that it includes a transaction T_{noRead} in place of T with the same interval, where T_{noRead} has W as its write set and an empty read set. By DAP, T_{noRead} does not modify in E' any base object accessed by any concurrent transaction whose data set does not overlap with W. In particular, T_{noRead} cannot make any modifications visible to T' in E'. Note that step s must exist in E', since by definition, E' is identical to E except in steps associated with T and T_{noRead}. However, in E, s's return value is affected by T's modification, and in E', this modification does not exist. Therefore, E' cannot be an execution of I. Contradiction. ◀

▶ **Lemma 4.5.** *In any implementation of a PDTS that provides both DDAP and invisible reads, a transaction whose write set is W does not cause any modifications on shared based objects on a node N visible to any concurrent transaction whose data set does not overlap with W on N.*

Proof. Let I be an implementation of a PDTS that satisfies DDAP and invisible reads. Let T be a transaction whose final write set on a node N in an execution E of I is W. Assume by contradiction that there exists some transaction T' concurrent with T in E whose data set on N does not overlap with W, but which sees a modification made by T on a base object on N in E. That is, there is some base-object operation step s of T' whose return value is affected by T's modification.

By invisible reads, there exists an execution E' of I identical to E except that it includes a transaction T_{noRead} in place of T with the same interval, where T_{noRead} has W as its write set on N and an empty read set. By DDAP, T_{noRead} does not modify in E' any base object on N accessed by any concurrent transaction whose data set does not overlap with W on N. In particular, T_{noRead} cannot make any modifications visible to T' on N in E'. Note that step s must exist in E', since by definition, E' is identical to E except in steps associated with T and T_{noRead}. However, in E, s's return value is affected by T's modification, and in E', this modification does not exist. Therefore, E' cannot be an execution of I. Contradiction. ◀

▶ **Lemma 4.6.** *There is no execution E of any serializable PDTS implementation that tolerates at least 1 failure in which there is a transaction T and prefix P such that $d_E(T, P) < 2$ and some process knows the decided value of some read of T in P.*

Proof. Assume by contradiction that there is some implementation I of a serializable PDTS that tolerates at least 1 failure, an execution E of I, and a prefix P of E such that $d_E(T, P) \leq 1$ and some process knows the decided value of some data item d of T in P. Without loss of generality, let process p on node N be the process that knows d's decided value, let that value be v and let T's invoking client be C. Note that since C does not have access to the data, and any step of any process not on C's node must be of depth at least 1, p cannot be on C's node, and cannot have received any message from any process other than C within depth less than 2. Therefore p can only know the value of d on node N, but not any other nodes. Consider the following executions.

E_{N-fail}. E_{N-fail} and E are identical up to right before T's invocation. In E_{N-fail}, node N fails at this point. Then, a transaction T' is invoked by a client $C' \neq C$. T' writes a value $v' \neq v$ to d and commits. After T' commits, T is invoked in E_{N-fail}. Clearly, by serializability, T's read of d in E_{N-fail} returns v' or a more updated value, but not v.

E_{N-slow}. E_{N-slow} is identical to E_{N-fail} except that node N does not fail in E_{N-slow}. Instead, all messages to and from N are arbitrarily delayed in E_{N-slow} starting at the same point at which N fails in E_{N-fail}. Clearly, E_{N-slow} is indistinguishable from E_{N-fail} to all processes not on N.

E'. E' is identical to E_{N-slow} except that node N receives messages from client C. Clearly, E' and E_{N-slow} are indistinguishable to all processes not on N. So, T's read of d must return the same value as in E_{N-slow}, namely v' or a more updated one, but not v. However, note that E' is also indistinguishable to processes on N from E in any prefix of E of partial depth < 2 for T, since no process in N received any messages other than those it received in E, and since clients do not receive any messages not related to their own transactions, so C must have sent the same message(s) to N in E' as it did in E. Therefore, there is a prefix P' of E' indistinguishable to p from P, in which v is not the decided value of d, contradicting p's knowledge of d's decided value in P. ◀

▶ **Theorem 4.1** (The FIDS theorem for sharded transactional systems)**.** *There is no implementation of a PDTS which shards data across multiple nodes that guarantees weak progress, and simultaneously provides fast decision, invisible reads, distributed disjoint-access parallelism, and serializability.*

Proof. Assume by contradiction that there exists an implementation I of a PDTS with all the properties in the theorem statement. Consider a database with 2 data items, X_1, X_2, partitioned on 2 nodes, N_1, N_2 respectively. Consider two transactions, T_1, T_2, with the following data sets: T_1's read set is $\{X_1\}$. Its write set is $\{X_2\}$ if its read returns the initial value of X_1, in which case it writes a value different from X_2's initial value. Otherwise, its write set is empty. For T_2, its read set is $\{X_2\}$, and its write set is $\{X_1\}$ if its read returns the initial value of X_2, and empty otherwise. If its write set is non-empty, it writes a value different from X_1's initial value. Let T_1 be executed by a client C_1 and T_2 be executed by a different client C_2. Consider the following executions.

Solo Executions. We define two executions S_1, S_2, corresponding to T_1, T_2 respectively running in isolation, without the other transaction present in the execution. Both executions are synchronous and failure-free. By weak progress, T_i commits in S_i, and by serializability, T_i returns the initial value of its read item and therefore its write set is not empty.

Concurrent Execution. We define an execution, E_{concur}, where T_1 and T_2 execute concurrently. On each node, each transaction is executed on different processes. Recall that this can happen since this is a parallel system, and the executing processes for a transaction are arbitrarily chosen among the idle processes of each node. In E_{concur}, for each transaction T_i, we let each process that executes it run until right before it knows the decided read set and read set value of T_i. Let the prefix of E_{concur} that includes all these steps be P_1. We then let each process that handles T_i run until when the next step of its handler has depth $\geq d_{S_i}(T_i) - 2$. Next, we let all messages sent on behalf of T_1 to N_1 and not yet received reach N_1 and be handled before any message sent on behalf of T_2 to N_1. For node N_2, we let the reverse happen; messages sent on behalf of T_2 reach it and are handled before messages sent on behalf of T_1. Finally, we resume all processes, and pause node processes that handle T_i when the next step of their handler has depth $\geq d_{S_i}(T_i)$. As for the client of each transaction, we let any messages sent to it arrive in the same order as they did in their corresponding solo executions (we will show that it receives the same messages in E_{concur}).

We now claim that execution E_{concur} is indistinguishable to C_i from S_i, and indistinguishable to each node process running T_i from the prefix of S_i containing all this process's steps of depth $< d_{S_i}(T_i)$. To do so, we consider the execution in two phases; the phase before the two transactions achieve knowledge of their data sets including their values (up to the end of P_1), and the phase afterwards.

Phase 1 of E_{concur}. Note that for any prefix P of E_{concur} in which T_i's read set's value is not known to be decided by some process, T_i's known decided write set in P is empty. Consider the longest prefix $P_{undecided_i}$ of P_1 in which the decided write set of T_i is still empty. Note that for every process p, its knowledge of T_i's write set in $P_{undecided_i}$ is the same as it is in P. Therefore, by Lemma 4.3, in any such prefix P, T_i may not make any modifications to shared base objects visible to *any* concurrent transaction. Therefore, in phase 1 there are no modifications visible to either transaction that were not visible in the solo execution as well. Thus, by the end of phase 1, E_{concur}'s prefix P_1 is indistinguishable to both transactions from their respective solo executions. Therefore, both transactions read the initial values of their respective read sets, and both have a non-empty write set in E_{concur}.

Phase 2 of E_{concur}. To show that E_{concur} remains indistinguishable from the solo executions to their respective transactions in phase 2, we rely on the order of messages that are received by the two nodes.

First, we note that by Lemma 4.5, T_i does not make base object modifications visible to $T_{(i \bmod 2)+1}$ on node N_i, since T_i's final write set is $\{X_{(i \bmod 2)+1}\}$, which does not intersect $T_{(i \bmod 2)+1}$'s final data set on node N_i.

Next, note that in each solo execution S_i, the first process that knows the decided value of T_i must be on node N_i, since that is where the data for the read of T_i is stored. Furthermore, by construction of E_{concur}, any messages sent on behalf of T_i to N_i immediately after both transactions gain knowledge of their write sets arrives before any such message sent on behalf of $T_{(i \bmod 2)+1}$, and its handler is completely executed. Thus, by the above claim, on both nodes, all handlers of both transactions for messages sent at depth $d_{E_{concur}}(T_i, P_1)$ execute to completion in a way that is indistinguishable to T_i from the solo execution S_i.

Finally, note that since S_i is synchronous and failure and conflict free, and I satisfies the fast decision property, by Corollary 3.7, the depth of T_i in S_i is at most 2 more than the partial depth of the first prefix in which T_i's data set including its values became known. In particular, since S_i is indistinguishable to processes executing T_i from E_{concur} up to that point, this means that $d_{S_i}(T_i) \leq d_{E_{concur}}(T_i, P_1) + 2$. Thus, once messages from within the handlers that were activated by messages sent in E_{concur} at depth $d_{E_{concur}}(T_i, P_1)$ are received, T_i must be decided in E_{concur} as well, since E_{concur} is indistinguishable from S_i to all processes running T_i up to this point. Therefore, both transactions commit successfully in E_{concur} in a manner indistinguishable from their respective solo executions.

However, this yields a circular dependency between the two transactions; T_2 must occur before T_1, since it returns the initial value of X_2, before T_1 writes to it. Similarly, T_1 must occur before T_2, since it returns the initial value of X_1. This therefore contradicts serializability. ◀

▶ **Theorem 4.2** (The R-FIDS theorem for replicated transactional systems). *There is no implementation of a PDTS that utilizes client-driven replication that guarantees weak progress, and simultaneously provides* 1*-seamless fault tolerance, fast decision, invisible reads, disjoint-access parallelism, and serializability.*

Proof. Assume by contradiction that there exists an implementation I of a parallel replicated transactional system with all the properties stated in the theorem.

Consider a transactional system with 3 nodes N_1, N_2, N_3. (For less than 3 nodes, there is no PDTS that tolerates $f \geq 1$ failures in the partial-synchrony model [18].) Further consider 3 transactions T_1, T_2, T_3, 3 client processes C_1, C_2, C_3, and 3 data items X_1, X_2, X_3 each of which is replicated on all 3 nodes. The data sets of the transactions are as follows: T_i's read set includes $X_{(i \bmod 3)+1}$, and if the result of T_i's read of $X_{(i \bmod 3)+1}$ is the initial value of $X_{(i \bmod 3)+1}$, its write set includes X_i. Otherwise, its write set is empty. Each transaction T_i, if its write set is non-empty, writes a value that is different from X_i's initial value.

Transactions read and write sets			
T	T_1	T_2	T_3
R_T	$\{X_2\}$	$\{X_3\}$	$\{X_1\}$
W_T	$\{X_1\}$ if R(X_2)=$\perp$, else $\{\}$	$\{X_2\}$ if R(X_3)=$\perp$, else $\{\}$	$\{X_3\}$ if R(X_1)=$\perp$, else $\{\}$

Consider the following executions. For each $i = 1, 2, 3$, in any of the following executions, if it includes T_i then its coordinator handler is executed by C_i.

Solo Executions. Let E_1, E_2, E_3 be failure-free synchronous executions of I, where transaction T_i runs solo in E_i. Since E_i contains a single transaction and I satisfies weak progress, transaction T_i commits in E_i. Since E_i is synchronous, has no failures and contains only T_i, and I satisfies fast decision, T_i is fast deciding in E_i.

▷ Claim. T_i must have a depth of at most 4 in E_i.

To see this, note that by the definition of fast decision, if transaction T_i in E_i has depth at least 3, the empty prefix of E_i must have an extension C_i of partial depth $d_{E_i}(T_i, C_i) \leq 2$ in which the value of the read set's item is known by some process to be decided, and therefore the write set is known by that process to be decided as well. By Corollary 3.7, the depth of T_i in E_i must be at most 2 more than the depth of C_i, and therefore is at most 4.

	X_1	X_2	X_3
N_1		1	2
N_2	2		1
N_3	1	2	

Figure 2 Visual representation of execution E_{concur} in the proof of Theorem 4.2. The numbers in the table represent the order of writing on each node; on node N_1, X_2 is written first, followed by X_3, and so on.

Since I satisfies 1-seamless fault tolerance, there exist executions E'_1, E'_2, E'_3 of I, where the first event in E'_i is a crash of N_i, T_i runs solo and the depth of T_i in E'_i is the same as its depth in E_i. We assume that in each E'_i, a different set of processes runs the handlers. Lastly, since I is serializable, each E'_i is serializable, thus T_i's read in E'_i returns the initial value of $X_{(i \bmod 3)+1}$, and therefore modifies X_i as part of its write set.

Since T_i's write set is only determined from the outcome of T_i's read, and may be empty until that read's value is decided, by Lemma 4.3, no base object modifications visible to other transactions are executed by T_i in E'_i until after T_i's read set values are known to some process. Let the shortest prefix at which some process gains knowledge of T_i's read set values in E'_i be P_i. By Lemma 4.6, $d_{E'_i}(T_i, P_i) \geq 2$.

Concurrent Execution. We define an execution E_{concur} with all 3 transactions. In E_{concur}, all messages between processes on node N_i and any process that executes handlers associated with T_i are arbitrarily delayed. For each $i = 1, 2, 3$, let processes that execute T_i in E'_i run in E_{concur} identically to E'_i, in the same order of steps, until the end of P_i.

Note that up to this point, E_{concur} is indistinguishable to all executing processes from the solo executions, since none of them has made any shared memory modifications visible to the others. Therefore, the prefix $P_{knowledge}$ of E_{concur} up to this point is an execution of I.

We continue E_{concur} as follows: Let all messages sent on behalf of T_i at depth $d_{E'_i}(T_i, P_i)$ be sent in E_{concur}, and be received and handled in the following order: on node N_1, messages for T_2 are received first, and their handlers are run to completion, followed by messages for T_3. On node N_2, T_3's messages are handled first, followed by messages of T_1. Finally, on node N_3, messages of T_1 are handled first followed by messages of T_2. coordinator handlers receive messages in the same order they received them in their corresponding solo executions.

Recall that transaction T_i reads data item $X_{(i \bmod 3)+1}$ and, if it reads the initial value, writes data item X_i. Thus, the service order defined above for execution E_{concur} (see the order in which the nodes process their writes in Figure 2) means that on each node, the second serviced transaction writes to data item X after the first transaction reads X, but it is never the case that a transaction reads a data item after it was written by another transaction on the same node. Since the data set of the second transaction to execute handlers after prefix $P_{knowledge}$ on each node does not overlap with the write set of the first one, and since I provides invisible reads and DAP, then by Lemma 4.4, the first transaction does not make base object modifications visible to the second transaction. In other words, on each node, a process executing the second transaction cannot observe any changes on shared memory. Thus, E_{concur} is still indistinguishable from E'_i to any node process that executes T_i up to the end of the handlers of messages sent at depth $d_{E_{concur}}(T_i, P_i)$. Note, however, that since $d_{E'_i}(T_i) \leq 4$ and $d_{E_{concur}}(T_i, P_i) \geq 2$, this means that E_{concur} remains indistinguishable from E'_i to these processes until T_i is decided.

Therefore, for all three transactions T_i commit in E_{concur}, reading the initial value of $X_{(i \bmod 3)+1}$ and writing a non-initial value in X_i. However, this yields a circular dependency between the transactions (transaction T_1 must happen before T_2, which must happen before T_3, which must happen before T_1), which contradicts serializability. ◀

Communication Lower Bounds for Cryptographic Broadcast Protocols

Erica Blum
University of Maryland, College Park, MD, USA

Elette Boyle
Reichman University, Herzliya, Israel
NTT Research, Sunnyvale, CA, USA

Ran Cohen
Reichman University, Herzliya, Israel

Chen-Da Liu-Zhang
Hochschule Luzern, Switzerland
Web3 Foundation, Zug, Switzerland

Abstract

Broadcast protocols enable a set of n parties to agree on the input of a designated sender, even in the face of malicious parties who collude to attack the protocol. In the honest-majority setting, a fruitful line of work harnessed randomization and cryptography to achieve low-communication broadcast protocols with sub-quadratic total communication and with "balanced" sub-linear communication cost per party. However, comparatively little is known in the *dishonest-majority* setting. Here, the most communication-efficient constructions are based on the protocol of Dolev and Strong (SICOMP '83), and sub-quadratic broadcast has not been achieved even using randomization and cryptography. On the other hand, the only nontrivial $\omega(n)$ communication lower bounds are restricted to *deterministic* protocols, or against *strong adaptive* adversaries that can perform "after the fact" removal of messages.

We provide communication lower bounds in this space, which hold against arbitrary cryptography and setup assumptions, as well as a simple protocol showing near tightness of our first bound.

- **Static adversary.** We demonstrate a tradeoff between *resiliency* and *communication* for randomized protocols secure against $n - o(n)$ *static* corruptions. For example, $\Omega(n \cdot \mathsf{polylog}(n))$ messages are needed when the number of honest parties is $n/\mathsf{polylog}(n)$; $\Omega(n\sqrt{n})$ messages are needed for $O(\sqrt{n})$ honest parties; and $\Omega(n^2)$ messages are needed for $O(1)$ honest parties.
  Complementarily, we demonstrate broadcast with $O(n \cdot \mathsf{polylog}(n))$ total communication and balanced $\mathsf{polylog}(n)$ per-party cost, facing any constant fraction of static corruptions.
- **Weakly adaptive adversary.** Our second bound considers $n/2 + k$ corruptions and a *weakly adaptive* adversary that cannot remove messages "after the fact." We show that any broadcast protocol within this setting can be attacked to force an arbitrary party to send messages to k other parties.
  Our bound implies limitations on the feasibility of *balanced* low-communication protocols: For example, ruling out broadcast facing 51% corruptions, in which all non-sender parties have sublinear communication locality.

2012 ACM Subject Classification Theory of computation → Communication complexity

Keywords and phrases broadcast, communication complexity, lower bounds, dishonest majority

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.10

Funding *Erica Blum*: Part of the work was done while the author was an intern at NTT Research.
Elette Boyle: Supported in part by AFOSR Award FA9550-21-1-0046 and ERC Project HSS (852952).

Ran Cohen: Supported in part by NSF grant no. 2055568 and by the Algorand Centres of Excellence programme managed by Algorand Foundation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of Algorand Foundation.
Chen-Da Liu-Zhang: Part of this work was done while the author was at NTT Research.

1 Introduction

In a *broadcast* protocol (a.k.a. Byzantine generals [42, 37]) a designated party (the sender) distributes its input message in a way that all honest parties agree on a common output, equal to the sender's message if the sender is honest. Broadcast is amongst the most widely studied problems in the context of distributed computing, and forms a fundamental building block in virtually any distributed system requiring reliability in the presence of faults. The focus of this work is on synchronous protocols that proceed in a round-by-round manner.

Understanding the required communication complexity of broadcast is the subject of a rich line of research. Most centrally, this is measured as the total number of bits communicated within the protocol, as a function of the number of parties n and corrupted parties t. Other metrics have also been studied, such as *message complexity* (number of actual messages sent), *communication locality* (defined as the *maximal degree* of a party in the induced communication graph of the protocol execution [9]), and *per-party* communication requirements (measuring how communication is split across parties).

The classical lower bound of Dolev and Reischuk [21] showed that $\Omega(n+t^2)$ messages are necessary for *deterministic* protocols (a cubic message complexity is also sufficient facing any linear number of corruptions [21, 22, 41][1]). This result came as part of several seminal impossibility results for deterministic protocols presented in the '80s, concerning feasibility [28], resiliency [37, 27], round complexity [26, 22], and connectivity [20, 27]. Those lower bounds came hand in hand with feasibility results initiated by Ben-Or [4] and Rabin [44], as well as Dolev and Strong [22], showing that randomization and cryptography are invaluable tools in achieving strong properties within broadcast protocols.

As opposed to bounds on feasibility, resiliency, and round complexity, the impossibility of [21] held for over 20 years, in both the honest- and dishonest-majority settings. Recently, there has been progress in the honest-majority setting, with several works demonstrating how randomization and cryptography can be used to bypass the classical communication complexity bound and achieve sub-quadratic communication: with information-theoretic security [36, 34, 35, 10] or with computational security under cryptographic assumptions [14, 1, 17, 6, 8]; some of these protocols even achieve poly-logarithmic locality and "balanced" sub-linear communication cost per party. While the security of some of these constructions holds against a *static* adversary that specifies corruptions before the protocol's execution begins, some of these protocols are even secure against a *weakly adaptive* adversary; that is, an adversary that cannot retract messages sent by a party upon corrupting that party. Abraham et al. [1] showed that this relaxation is inherent for sub-quadratic broadcast, even for randomized protocols, by demonstrating an $\Omega(t^2)$ communication lower bound in the presence of a *strongly rushing* adversary; that is, an adversary that can "drop" messages by corrupting the sender after the message is sent but before it is delivered – this ability is known as *after-the-fact removal.*

[1] We note that [21, 22, 41] rely on cryptography, so they are not deterministic per se; however, these protocols make a black-box use of the cryptographic primitives and are deterministic otherwise.

Focusing on the *dishonest-majority* setting, however, comparatively little is known about communication complexity. Here, the most communication-efficient broadcast constructions are based on the protocol of Dolev and Strong [22], and broadcast with $o(nt)$ messages has not been achieved even using randomization and cryptography. The state-of-the-art protocols, for a constant fraction $t = \Theta(n)$ of corruptions, are due to Chan et al. [13] in the weakly adaptive setting under a trusted setup assumption, and to Tsimos et al. [45] in the static setting under a weaker setup assumption; however, both works require $\Omega(nt)$ communication, namely $\tilde{O}(n^2)$.[2] On the other hand, the only nontrivial $\omega(n)$ communication lower bounds are those discussed above, restricted to deterministic protocols, or against strong adaptive adversaries.

1.1 Our Contributions

In this work, we explore the achievable communication complexity of broadcast in the dishonest-majority setting. We provide new communication lower bounds in this space, which hold against arbitrary cryptographic and setup assumptions, as well as a simple protocol showing near tightness of our first bound. Our results consider a *synchronous* communication model: lower bounds in this model immediately translate into lower bounds in the *asynchronous* and *partially synchronous* models, whereas protocols in the latter models can only tolerate $t < n/3$ corruptions [23] implying that synchrony is inherently needed for our protocol construction.

Static adversary. We begin with the setting of static corruptions. We demonstrate a simple modification to the protocol of Chan et al. [13], incorporating techniques from Tsimos et al. [45], which obtains a new protocol with essentially optimal $\tilde{O}(n)$ communication. The resulting protocol relies on the same assumptions as [13]: namely, cryptographic verifiable random functions (VRFs)[3] and a trusted public-key infrastructure (PKI) setup, where the keys for each party are honestly generated and distributed. Further, the protocol is resilient against any constant fraction of static corruptions as in [45], and achieves balanced $\tilde{O}(1)$ cost per party.

▶ **Proposition 1** (sub-quadratic broadcast facing a constant fraction of static corruptions). *Let $0 < \epsilon < 1$ be any constant. Assuming a trusted-PKI for VRFs and signatures, it is possible to compute broadcast with $\tilde{O}(n)$ total communication ($\tilde{O}(1)$ per party) facing a static adversary corrupting $(1 - \epsilon) \cdot n$ parties.*

Perhaps more interestingly, in the regime of $n - o(n)$ static corruptions, we demonstrate a feasibility tradeoff between *resiliency* and *communication* that nearly tightly complements the above upper bound. We show that resilience in the face of only $\epsilon(n) \cdot n$ honest parties, for $\epsilon(n) \in o(1)$, demands message complexity scaling as $\Omega(n/\epsilon(n))$. Note that a lower bound on message complexity is stronger than for communication complexity, directly implying the latter. Our lower bound holds for randomized protocols, given any cryptographic assumption and any setup information that is generated by an external trusted party and given to the parties before the beginning of the protocol, including the assumptions of the above upper bound.

[2] As standard in relevant literature, in this work $\tilde{O}$ notation hides polynomial factors in $\log(n)$ as well as in the cryptographic security parameter κ.
[3] A verifiable random function [40] is a pseudorandom function that provides a non-interactively verifiable proof for the correctness of its output.

▶ **Theorem 2** (communication lower bound for static corruptions)**.** *Let $\epsilon(n) \in o(1)$. If there exists a broadcast protocol that is secure against $(1-\epsilon(n)) \cdot n$ static corruptions, then the message complexity of the protocol is $\Omega(n \cdot \frac{1}{\epsilon(n)})$.*

For example, for $n - n/\log^d(n)$ corruptions with a constant $d \geq 1$ (i.e., $\epsilon(n) = \log^{-d}(n)$), the message complexity must be $\Omega(n \cdot \log^d(n))$. For $n - \sqrt{n}$ corruptions (i.e., $\epsilon(n) = 1/\sqrt{n}$), the message complexity must be $\Omega(n \cdot \sqrt{n})$. And for $n - c$ corruptions with a constant $c > 1$ (i.e., $\epsilon(n) = c/n$), the message complexity must be $\Omega(n^2)$, in particular meaning that sub-quadratic communication is impossible in this regime.

As noted, Theorem 2 holds for any cryptographic and setup assumptions. This captures, in particular, PKI-style setup (such as the VRF-based PKI of Chan et al. [13]) in which the trusted party samples a private/public key-pair for each party and gives each party its private key together with the vector of all public keys. It additionally extends to even stronger, more involved setup assumptions for generating *correlated randomness* beyond a product distribution, e.g., setup for threshold signatures where parties' secret values are nontrivially correlated across one another.

Weakly adaptive adversary. The lower bound of Theorem 2 carries over directly to the setting of weakly adaptive adversaries. Shifting back to the regime of a constant fraction of corruptions, one may naturally ask whether a protocol such as that from Proposition 1 can also exist within this regime.

Unfortunately, given a few minutes thought one sees that a balanced protocol with polylogarithmic per-party communication as demonstrated by Proposition 1 cannot translate to the weakly adaptive setting. The reason is that if the *sender* party speaks to at most t other parties, then the adaptive adversary can simply corrupt each receiving party and drop the message, thus blocking any information of the sender's input from reaching honest parties.

However, this attack applies only to the unique sender party. Indeed, non-sender parties contribute no input to the protocol to be blocked; and, without the ability to perform "after-the-fact" message removal, a weakly adaptive adversary cannot prevent communication from being *received* by a party without a very large number of corruptions.

We therefore consider the locality of *non-sender* parties, and ask whether sublinear locality is achievable. Our third result answers this question in the negative. That is, we show an efficient adversary who can force any party of its choosing to communicate with a large number of neighbors. Note that this in particular lower bounds the per-party communication complexity of non-sender parties.

▶ **Theorem 3** (non-sender locality facing adaptive corruptions)**.** *Let $0 < k < (n-1)/2$ and let π be an n-party broadcast protocol secure against $t = n/2 + k$ adaptive corruptions. Then, for any non-sender party P_{i^*} there exists a PPT adversary that can force the locality of P_{i^*} to be larger than k, except for negligible probability.*

For example, for $k \in \Theta(n)$, e.g., a constant fraction $t = 0.51 \cdot n$ of corruptions, the locality of P_{i^*} must be $\Theta(n)$, thus forming a separation from Proposition 1 for the locality of non-sender parties. Similarly to Theorem 2, this bound holds in the presence of any correlated-randomness setup and for any cryptographic assumptions.

We remark that our bound further indicates a design requirement for any protocol attempting to achieve sub-quadratic $o(n^2)$ communication complexity within this setting. To obtain $o(n^2)$ communication, it must of course be that nearly all parties have sublinear communication locality. Our result shows that any such protocol must include instructions causing a party to send out messages to a linear number of other parties upon determining that it is under attack.

Summary. For completeness, Table 1 summarizes our results alongside prior work.

Table 1 Communication requirements of dishonest-majority (synchronous) broadcast. We consider the standard, property-based definition of broadcast (see Definition 4). For each type of adversary (strongly adaptive, weakly adaptive, and static), we consider the state-of-the-art protocols and lower bounds in terms of setup, number of corruptions, total communication and (non-sender) locality. For setup we distinguish bare PKI, where each party locally generates its key pair, as opposed to trusted PKI, where all keys are generated by a trusted dealer. Reference [21] is only for deterministic protocols.

	setup	corruptions	total com.	locality	ref.
strongly adaptive	bare pki	$t < n$	$O(n^3)$	n	[22]
	any	$t = \Theta(n)$	$\Omega(n^2)$	$\Omega(n)$	[1]
weakly adaptive	trusted pki	$t = \Theta(n)$	$\tilde{O}(n^2)$	$O(n)$	[13]
	any	$t = \Theta(n)$		$\Omega(n)$	Thm. 3
static	any (deterministic)	$t = \Theta(n)$	$\Omega(n^2)$	$\Omega(n)$	[21]
	bare pki	$t = \Theta(n)$	$\tilde{O}(n^2)$	$\tilde{O}(1)$	[45]
	trusted pki	$t = \Theta(n)$	$\tilde{O}(n)$	$\tilde{O}(1)$	Prop. 1
	any	$t = (1 - \epsilon(n)) \cdot n$, $\epsilon(n) \in o(1)$	$\Omega(n \cdot \frac{1}{\epsilon(n)})$		Thm. 2
		e.g., $t = n - \frac{n}{\mathsf{polylog}(n)}$	$\Omega(n \cdot \mathsf{polylog}(n))$		
		e.g., $t = n - \sqrt{n}$	$\Omega(n \cdot \sqrt{n})$		
		e.g., $t = n - O(1)$	$\Omega(n^2)$		

1.2 Technical Overview

The proof of Proposition 1 follows almost immediately from [13] and [45]. We therefore focus on our lower bounds.

Communication lower bound for static corruptions. The high-level idea of the attack underlying Theorem 2 is to split all parties except for the sender P_s into two equal-size subsets, $\mathcal{A}$ and $\mathcal{B}$, randomly choose a set $\mathcal{S}$ of size $\epsilon(n) - 1$ parties in $\mathcal{A}$ and a party $\mathsf{P}_{i^*} \in \mathcal{B}$, and corrupt all parties but $\mathcal{S} \cup \{\mathsf{P}_{i^*}\}$. The adversary proceeds by running two independent executions of the protocol. In the first, the sender runs an execution on input 0 towards $\mathcal{A}$, and all corrupted parties in $(\{\mathsf{P}_s\} \cup \mathcal{A}) \setminus \mathcal{S}$ ignore all messages from parties in $\mathcal{B}$ (pretending they all crashed). In the second, the sender runs an execution on input 1 towards $\mathcal{B}$, and all corrupted parties in $(\{\mathsf{P}_s\} \cup \mathcal{B}) \setminus \{\mathsf{P}_{i^*}\}$ ignore all messages from parties in $\mathcal{A}$.

As long as the honest parties in $\mathcal{S}$ and the honest party P_{i^*} do not communicate, the adversary will make them output different values. This holds because, conditioned on no communication between $\mathcal{S}$ and P_{i^*}, the view of honest parties in $\mathcal{S}$ is indistinguishable from a setting where the adversary crashes all parties in $\mathcal{B}$ and an honest sender has input 0; in this case, all parties in $\mathcal{A}$ (and in particular in $\mathcal{S}$) must output 0. Similarly, conditioned on no communication between $\mathcal{S}$ and P_{i^*}, the view of P_{i^*} is indistinguishable from a setting where the adversary crashes all parties in $\mathcal{A}$ and an honest sender has input 1; in this case, all parties in $\mathcal{B}$ (and in particular P_{i^*}) must output 1.

The challenge now is to argue that the honest parties in $\mathcal{S}$ and the honest party P_{i^*} do not communicate with noticeable probability. Note that this does not follow trivially from the overall low communication complexity, as the communication patterns unfold as a function of the adversarial behavior, which in particular depends on the choice of $\mathcal{S}$ and P_{i^*}. The argument instead follows from a series a delicate steps that compare the view of parties in this execution with other adversarial strategies.

The underlying trick is to analyze the event of communication between $\mathcal{S}$ and P_{i^*} by splitting into two sub-cases: when $\mathcal{S}$ speaks to P_{i^*} *before* receiving any message from P_{i^*}, and when P_{i^*} speaks to $\mathcal{S}$ *before* receiving any message from $\mathcal{S}$. (Note, these events are not disjoint.) The important observation is that before any communication is received by the other side, then each side's view in the attack is identically distributed as in a hypothetical execution in which the corresponding set $\mathcal{A}$ or $\mathcal{B}$ crashes from the start. Since these simple crash adversarial strategies are indeed independent of $\mathcal{S}$ and P_{i^*}, then we can easily analyze and upper bound the probability of $\mathcal{S}$ and P_{i^*} communicating within their hypothetical executions. To finalize the argument, we carry this analysis over to show that with noticeable probability P_{i^*} does not communicate with $\mathcal{S}$ in an actual execution with the original adversary.

Locality lower bound for weakly adaptive corruptions. We proceed to consider the setting of a constant fraction $n/2+k$ of *weakly adaptive* corruptions. As mentioned above, in the adaptive setting it is easy to see that the sender must communicate with many parties, since otherwise the adversary may crash every party that the sender communicates with; therefore, the challenging part is to focus on non-sender parties. Further, when considering strong adaptive adversaries that can perform after-the-fact message removal by corrupting the sender, Abraham et al. [1] showed that every honest party must communicate with a linear number of parties. In our setting, we do not consider such capabilities of the adversary. In particular, once the adversary learns that an honest party has chosen to send a message, this message cannot be removed or changed.

Unlike our previous lower bound which assumed $n-o(n)$ corruptions, here we consider a constant fraction of corruptions, so we cannot prevent sets of honest parties from communicating with each other. Our approach, instead, is to keep the targeted party P_{i^*} confused about the output of other honest parties.

More concretely, our adversarial strategy splits all parties but the sender and P_{i^*} into disjoint equal-size sets $\mathcal{S}_0$ and $\mathcal{S}_1$ of parties, samples a random bit b and corrupts the sender party and the parties in $\mathcal{S}_{1-b}$. The adversary communicates with $\mathcal{S}_0$ as if the sender's input is 0 and all parties in $\mathcal{S}_1$ have crashed, and at the same time plays towards $\mathcal{S}_1$ as if the sender's input is 1 and all parties in $\mathcal{S}_0$ have crashed. Although the adversary cannot prevent honest parties from $\mathcal{S}_b$ from sending messages to the targeted party P_{i^*}, it can corrupt every party that *receives* a message from P_{i^*}. The effect of this attack is that, although P_{i^*} can tell that the sender is cheating, P_{i^*} cannot know whether parties in $\mathcal{S}_0$ or parties in $\mathcal{S}_1$ are honest. And, moreover, P_{i^*} cannot know whether the remaining honest parties *know* that the sender is cheating or if they believe that the sender is honest and other parties crashed – in which case they must output a bit (either 0 if $\mathcal{S}_0$ are honest or 1 if $\mathcal{S}_1$ are honest). To overcome this attack, P_{i^*} must communicate with sufficiently many parties such that the adversary's corruption budget will run out, i.e., with output locality at least k.

1.3 Further Related Work

Since the classical results from the '80s, a significant line of work has been devoted to understanding the complexity of broadcast protocols.[4]

[4] In this work we consider broadcast protocols that achieve the usual properties of *termination*, *agreement*, and *validity*. We note that stronger notions of broadcast have been considered in the literature, e.g., in the adaptive setting, the works of [31, 30, 16] study *corruption fairness* ensuring that once any receiver learns the sender's input, the adversary cannot corrupt the sender and change its message). As our main technical contributions are lower bounds, focusing on weaker requirements yields stronger results.

Communication complexity. In the honest-majority regime, we know of several protocols, deterministic [5, 15, 41] or randomized [39, 29], that match the known lower bounds [21, 1] for strongly adaptive adversaries. When considering static, or weakly adaptive security, a fruitful line of works achieved sub-quadratic communication, with information-theoretic security [36, 34, 35, 10] or with computational security [14, 1, 17, 6, 8] .

In the dishonest-majority regime, the most communication-efficient broadcast constructions are based on the protocol of Dolev and Strong [22]. This protocol is secure facing any number of strongly adaptive corruptions and the communication complexity is $O(n^3)$. When considering weakly adaptive corruptions, Chan et al. [13] used cryptography and trusted setup to dynamically elect a small, polylog-size committee in each round and improved the communication to $\tilde{O}(n^2)$. In the static-corruption setting, Tsimos et al. [45] achieved $\tilde{O}(n^2)$ communication by running the protocol of [22] over a "gossiping network" [19, 33]. This work further achieved *amortized* sub-quadratic communication facing weakly adaptive corruptions when all parties broadcast in parallel.

A line of works focused on achieving *balanced* protocols, where all parties incur the same work in terms of communication complexity [35, 8, 2]. The work of [8] also showed lower bounds on the necessary setup and cryptographic assumptions to achieve balanced protocols when extending almost-everywhere agreement to full agreement.[5] Message dissemination protocols [18, 38] have also been proven useful for constructing balanced protocols.

The work in [32] showed that without trusted setup assumptions, at least one party must send $\Omega(n^{1/3})$ messages, in the *static filtering* model, where each party must decide which set of parties it will accept messages from in each round before the rounds begins. We remark that our lower bounds hold also given trusted setup, and in the dynamic-filtering model (in which sub-quadratic upper bounds have been achieved).

Connectivity. Obtaining communication-efficient protocols inherently relies on using a strict subgraph of the communication network. Early works [20, 27] showed that *deterministic* broadcast is possible in an incomplete graph only if the graph is $(t+1)$-connected. The influential work of King et al. [36] laid a path not only for randomized Byzantine agreement with sub-quadratic communication, but also for protocols that run over a partial graph [34, 35, 10, 8]. The graphs induced by those protocols yield expander graphs, and the work of [7] showed that in the strongly adaptive setting and facing a linear number of corruptions, no protocol for all-to-all broadcast in the plain model (without PKI setup) can maintain a non-expanding communication graph against all adversarial strategies. Further, feasibility of broadcast with a non-expander communication graph, admitting a sub-linear cut, was demonstrated in weaker settings [7].

Outline of Paper

Preliminaries can be found in Section 2. In Section 3, we present the message-complexity lower bound for static corruption,s and in Section 4, we present the locality lower bound for weakly adaptive corruptions. The statically secure broadcast protocol with sub-quadratic communication and poly-logarithmic locality can be found in the full version of the paper.

[5] *Almost-everywhere agreement* [24] is a relaxed problem in which all but an $o(1)$ fraction of the parties must reach agreement. For this relaxation, King et al. [36] showed an efficient protocol, with poly-logarithmic locality, communication, and rounds. This protocol serves as a stepping stone to several sub-quadratic Byzantine agreement protocols, by extending almost-everywhere agreement to *full agreement* [36, 34, 35, 10, 8].

2 Preliminaries

In this section, we present the security model and preliminary definitions.

Notations. We use calligraphic letters to denote sets or distributions (e.g., $\mathcal{S}$), uppercase for random variables (e.g., R), lowercase for values (e.g., r), and sans-serif (e.g., A) for algorithms (i.e., Turing machines). For $n \in \mathbb{N}$, let $[n] = \{1, \ldots, n\}$. Let poly denote the set all positive polynomials and let PPT denote a probabilistic (interactive) Turing machines that runs in *strictly* polynomial time. We denote by κ the security parameter. A function $\nu \colon \mathbb{N} \mapsto [0, 1]$ is *negligible*, denoted $\nu(\kappa) = \mathsf{negl}(\kappa)$, if $\nu(\kappa) < 1/p(\kappa)$ for every $p \in \mathsf{poly}$ and sufficiently large κ. Moreover, we say that $\nu \colon \mathbb{N} \mapsto [0, 1]$ is *noticeable* if $\nu(\kappa) \geq 1/p(\kappa)$ for some $p \in \mathsf{poly}$ and sufficiently large κ. When using the $\tilde{O}(n)$ notation, polynomial factors in $\log(n)$ and the security parameter κ are omitted.

Protocols. All protocols considered in this paper are PPT (probabilistic polynomial time): the running time of every party is polynomial in the (common) security parameter, given as a unary string. For simplicity, we consider Boolean-input Boolean-output protocols, where apart from the common security parameter, a designated sender P_s has a single input bit, and each of the honest parties outputs a single bit. We note that our protocols can be used for broadcasting longer strings, with an additional dependency of the communication complexity on the input-string length.

As our main results are the lower bounds, we consider protocols in the *correlated randomness* model; that is, prior to the beginning of the protocol π a trusted dealer samples values $(r_1, \ldots, r_n) \leftarrow \mathcal{D}_\pi$ from an efficiently sampleable known distribution $\mathcal{D}_\pi$ and gives the value r_i to party P_i. This model captures, for example, a trusted PKI setup for digital signatures and verifiable random functions (VRFs), where the dealer samples a public/private keys for each party and hands to each P_i its secret key and a vector of all public keys; this is the setup needed for our upper bound result. The model further captures more involved distributions, such as setup for *threshold signatures*, information-theoretic PKI [43], pairwise correlations for *oblivious transfer* [3] , and more.

We define the *view* of a party P_i as its setup information r_i, its random coins, possibly its input (in case P_i is the sender), and its set of all messages received during the protocol.

Communication model. The communication model that we consider is *synchronous*, meaning that protocols proceed in rounds. In each round every party can send a message to every other party over an authenticated channel, where the adversary can see the content of all transmitted messages, but cannot drop/inject messages. We emphasize that our lower bounds hold also in the private-channel setting which can be established over authenticated channels using public-key encryption and a PKI setup; our protocol construction only requires authenticated channels. It is guaranteed that every message sent in a round will arrive at its destination by the end of that round. The adversary is *rushing* in the sense that it can use the messages received by corrupted parties from honest parties in a given round to determine the corrupted parties' messages for that round.

Adversary model. The adversary runs in probabilistic polynomial time and may corrupt a subset of the parties and instruct them to behave in an arbitrary (malicious) manner. Some of our results (the lower bound in Section 3 and the feasibility result) consider a static adversary that chooses which parties to corrupt *before* the beginning of the protocol, i.e., *before* the setup information is revealed to the parties. Note that this strengthens the lower bound, but provides a weaker feasibility result. Our second lower bound (Section 4) considers an adaptive adversary that can choose which parties to corrupt during the course of the protocol, based on information it dynamically learns. We consider the *atomic-multisend model* (also referred to as a weakly adaptive adversary), meaning that once a party P_i starts sending messages in a given round, it cannot be corrupted until it completes sending all messages for that round, and every message sent by P_i is delivered to its destination. This is weaker than the standard model for adaptive corruptions [25, 12, 11] (also referred to as a strongly rushing adversary), which enables the adversary to corrupt a party at any point during the protocol and drop/change messages that were not delivered yet. Again, we note that the weaker model we consider yields a stronger lower bound. Further, in the stronger model, a result by Abraham et al. [1] rules out sub-quadratic protocols with linear resiliency, even in the honest-majority setting.

Broadcast. We consider the standard, property-based definition of broadcast.

▶ **Definition 4** (Broadcast protocol). *An n-party protocol π, where a distinguished sender P_s holds an initial input message $x \in \{0, 1\}$, is a* broadcast protocol *secure against t corruptions, if the following conditions are satisfied for any PPT adversary that corrupts up to t parties:*

- ***Termination:*** *There exists an a-priori-known round R such that the protocol is guaranteed to complete within R rounds (i.e., every so-far honest party produces an output value).*
- ***Agreement:*** *For every pair of parties P_i and P_j that are honest at the end of the protocol, if party P_i outputs y_i and party P_j outputs y_j, then $y_i = y_j$ with all but negligible probability in κ.*
- ***Validity:*** *If the sender is honest at the end of the protocol, then for every party P_i that is honest at the end of the protocol, if P_i outputs y_i then $y_i = x$ with all but negligible probability in κ.*

The communication locality [9, 7] of a protocol corresponds to the maximal degree of any honest party in the communication graph induced by the protocol execution. While defining the incoming communication edges to a party can be subtle (as adversarial parties may "spam" honest parties; see e.g., a discussion in [7]), out-edges of honest parties are clearly identifiable from the protocol execution. In this paper, we will focus on this simpler notion of output-locality, and use the terminology *locality* of the protocol to simply refer to this value. Our results provide a lower bound on output locality of given protocols, which in turn directly lower bounds standard locality as in [9, 7].

▶ **Definition 5** ((Output) Locality). *An n-party t-secure broadcast protocol π with setup distribution $\mathcal{D}_\pi$ has locality ℓ, if for every PPT adversary Adv corrupting up to t parties and every sender input x it holds that*

$$\Pr\left[\mathsf{OutEdges}(\pi, \mathsf{Adv}, \mathcal{D}_\pi, \kappa, x) > \ell\right] \leq \mathsf{negl}(\kappa),$$

where $\mathsf{OutEdges}(\pi, \mathsf{Adv}, \mathcal{D}_\pi, \kappa, x)$ is the random variable of the maximum number of parties any honest party sends messages to, defined by running the protocol π with the adversary Adv and setup distribution $\mathcal{D}_\pi$, security parameter κ and sender input x. The probability is taken over the random coins of the honest parties, the random coins of Adv, and the sampling coins from the setup distribution $\mathcal{D}_\pi$.

3 Message-Complexity Lower Bound for Static Corruptions

We begin with the proof of Theorem 2. The high-level idea of the lower bound is that if a protocol has $o(n^2)$ messages, then, with noticeable probability, a randomly chosen pair of parties do not communicate even under certain attacks.

▶ **Theorem 6** (Theorem 2, restated). *Let $\epsilon(n) \in o(1)$. If there exists a broadcast protocol that is secure against $(1-\epsilon(n)) \cdot n$ static corruptions, then the message complexity of the protocol is $\Omega(n \cdot \frac{1}{\epsilon(n)})$.*

Proof. Let $\psi(n) = \frac{1}{12\epsilon(n)}$ and let π be a broadcast protocol with message complexity $\mathsf{MC} = n \cdot \psi(n)$ that is secure against $(1-\epsilon(n)) \cdot n$ static corruptions. (In fact, we will prove a stronger statement than claimed, where the message complexity of the protocol must be greater than $n \cdot \frac{1}{12\epsilon(n)}$.) Without loss of generality, we assume that the setup information sampled before the beginning of the protocol $(r_1, \ldots, r_n) \leftarrow \mathcal{D}_\pi$ includes the random string used by each party. That is, every party P_i generates its messages in each round as a function of r_i, possibly its input (if P_i is the sender), and its incoming messages in prior rounds. Again, without loss of generality, let P_1 denote be the sender, and split the remaining parties to two equal-size subsets $\mathcal{A}$ and $\mathcal{B}$ (for simplicity, assume that n is odd).

Consider the adversary Adv_1 that proceeds as follows:

1. Choose randomly a set $\mathcal{S} \subseteq \mathcal{A}$ of size $\epsilon(n) \cdot n - 1$ and a party $\mathsf{P}_{i^*} \in \mathcal{B}$.
2. Corrupt all parties except for $\mathcal{S} \cup \{\mathsf{P}_{i^*}\}$.
3. Receive the setup information of the corrupted parties $\{r_i \mid \mathsf{P}_i \notin \mathcal{S} \cup \{\mathsf{P}_{i^*}\}\}$.
4. Maintain two independent executions, denoted Exec_0 and Exec_1, as follows.
    - In the execution Exec_0, the adversary runs in its head the parties in $\mathcal{A} \setminus \mathcal{S}$ honestly on their setup information $\{r_i \mid \mathsf{P}_i \in \mathcal{A} \setminus \mathcal{S}\}$ and a copy of the sender, denoted P_1^0, running on input 0 and setup information r_1.
      The adversary communicates on behalf of the virtual parties in $(\mathcal{A} \setminus \mathcal{S}) \cup \{\mathsf{P}_1^0\}$ with the honest parties in $\mathcal{S}$ according to this execution. Every corrupted party in $\mathcal{B} \setminus \{\mathsf{P}_{i^*}\}$ crashes in this execution, and the adversary drops every message sent by the virtual parties in $(\mathcal{A} \setminus \mathcal{S}) \cup \{\mathsf{P}_1^0\}$ to P_{i^*} and does not deliver any message from P_{i^*} to these parties.
    - In the execution Exec_1, the adversary runs in its head the parties in $\mathcal{B} \setminus \{\mathsf{P}_{i^*}\}$ honestly on their setup information $\{r_i \mid \mathsf{P}_i \in \mathcal{B} \setminus \{\mathsf{P}_{i^*}\}\}$ and a copy of the sender, denoted P_1^1, running on input 1 and setup information r_1.
      The adversary communicates on behalf of the virtual parties in $(\mathcal{B} \setminus \{\mathsf{P}_{i^*}\}) \cup \{\mathsf{P}_1^1\}$ with the honest P_{i^*} according to this execution. Every corrupted party in $\mathcal{A} \setminus \mathcal{S}$ crashes in this execution, and the adversary drops every message sent by the virtual parties in $(\mathcal{B} \setminus \{\mathsf{P}_{i^*}\}) \cup \{\mathsf{P}_1^1\}$ to honest parties in $\mathcal{S}$ and does not deliver any message from $\mathcal{S}$ to these parties.

We start by defining a few notations. Consider the following random variables

$$\textsc{SetupAndCoins} = (R_1, \ldots, R_n, S, I^*),$$

where $R_1, \ldots, R_n$ are distributed according to $\mathcal{D}_\pi$, and S takes a value uniformly at random in the subsets of $\mathcal{A}$ of size $\epsilon(n) \cdot n - 1$, and I^* takes a value uniformly at random in $\mathcal{B}$. During the proof, R_i represents the setup information (including private randomness) of party P_i, whereas the pair (S, I^*) corresponds to the random coins of the adversary Adv_1 (used for choosing $\mathcal{S}$ and P_{i^*}). Unless stated otherwise, all probabilities are taken over these random variables.

Let AttackMain be the random variable defined by running the protocol π with the adversary Adv_1 over SetupAndCoins. That is, AttackMain consists of a vector of $n+1$ views: of the honest parties in $S \cup \{\mathsf{P}_{I^*}\}$ and of the corrupted parties in $\mathcal{A} \setminus S$ and $\mathcal{B} \setminus \{\mathsf{P}_{I^*}\}$, where the ith view is denoted by $\textsc{view}_i^{\mathsf{main}}$, and of two copies of the sender P_1^0 and P_1^1, denoted $\textsc{view}_{\text{1-0}}^{\mathsf{main}}$ and $\textsc{view}_{\text{1-1}}^{\mathsf{main}}$, respectively. Each view consists of the setup information R_i, possibly the input, and the set of received messages in each round. Specifically,

$$\textsc{AttackMain} = \left(\textsc{view}_{\text{1-0}}^{\mathsf{main}}, \textsc{view}_{\text{1-1}}^{\mathsf{main}}, \textsc{view}_2^{\mathsf{main}}, \ldots, \textsc{view}_n^{\mathsf{main}}\right).$$

Denote by $\mathcal{E}_{\mathsf{disconnect}}^{\mathsf{main}}$ the event that P_{I^*} and S do not communicate in AttackMain; that is, P_{I^*} does not send any message to parties in S (according to $\textsc{view}_{I^*}^{\mathsf{main}}$) and every party P_J with $J \in S$ does not send any message to P_{I^*} (according to $\textsc{view}_J^{\mathsf{main}}$). We proceed to prove that the event $\mathcal{E}_{\mathsf{disconnect}}^{\mathsf{main}}$ occurs with noticeable probability.

▶ **Lemma 7.** $\Pr\left[\mathcal{E}_{\mathsf{disconnect}}^{\mathsf{main}}\right] \geq \frac{1}{3}$.

Proof. Denote by $\mathcal{E}_{\mathsf{S}\to\mathsf{P}}^{\mathsf{main}}$ the event that a party in S sends a message to P_{I^*} in AttackMain, and P_{I^*} did not send any message to any party in S in any prior round. We begin by upper bounding the probability of $\mathcal{E}_{\mathsf{S}\to\mathsf{P}}^{\mathsf{main}}$.

▷ Claim 8. $\Pr\left[\mathcal{E}_{\mathsf{S}\to\mathsf{P}}^{\mathsf{main}}\right] \leq \frac{1}{3}$.

Proof. Consider a different adversary for π, denoted Adv_B, that statically corrupts all parties in $\mathcal{B}$ and crashes them (all other parties including the sender are honest). Let AttackCrashB denote the random variable defined by running the protocol π with the adversary Adv_B over SetupAndCoins, in which the honest sender's input is 1. That is, AttackCrashB consists of a vector of $n/2+1$ views: of the honest parties in $\mathcal{A}$, where the ith view is denoted by $\textsc{view}_i^{\mathsf{crash\text{-}B}}$, and the sender P_1 denoted by $\textsc{view}_1^{\mathsf{crash\text{-}B}}$. Each view consists of the setup information R_i, the input 1 for P_1, and the set of received messages in each round. Specifically,

$$\textsc{AttackCrashB} = \left(\textsc{view}_i^{\mathsf{crash\text{-}B}}\right)_{i \in \mathcal{A} \cup \{1\}}.$$

Denote by $\mathcal{E}_{\mathsf{S}\to\mathsf{P}}^{\mathsf{crash\text{-}B}}$ the event that a party in S sends a message to P_{I^*} in AttackCrashB such that P_{I^*} did not send any message to any party in S in any prior round. Note that as long as parties in S do not receive a message from P_{I^*} until some round ρ in AttackMain, their joint view is identically distributed as their joint view in AttackCrashB up until round ρ. Therefore,

$$\Pr\left[\mathcal{E}_{\mathsf{S}\to\mathsf{P}}^{\mathsf{main}}\right] = \Pr\left[\mathcal{E}_{\mathsf{S}\to\mathsf{P}}^{\mathsf{crash\text{-}B}}\right].$$

Note that, by the definition of Adv_B, the distribution of AttackCrashB, and therefore $\Pr\left[\mathcal{E}_{\mathsf{S}\to\mathsf{P}}^{\mathsf{crash\text{-}B}}\right]$, is *independent* of the random variables S and I^*. Hence, one can consider the mental experiment where $R_1, \ldots, R_n$ are first sampled for setting AttackCrashB, and later, S and I^* are independently sampled at random. This does not affect the event $\mathcal{E}_{\mathsf{S}\to\mathsf{P}}^{\mathsf{crash\text{-}B}}$.

Recall that the message complexity of π is $\mathsf{MC} = n \cdot \psi(n)$ for $\psi(n) = \frac{1}{12\epsilon(n)}$. Further, S is of size $|S| = \epsilon(n) \cdot n - 1$ and $|\mathcal{A}| = |\mathcal{B}| = n/2$. Observe that the message complexity upper-bounds the number of communication edges between $\mathcal{A}$ and $\mathcal{B}$. Further, the probability that a party in S talks first to P_{I^*} is upper-bounded by the probability that there exists a communication edge between S and P_{I^*}. Since S and I^* are uniformly distributed in $\mathcal{A}$ and $\mathcal{B}$, respectively, we obtain that this probability is bounded by

$$\begin{aligned}\Pr\left[\mathcal{E}^{\mathsf{crash\text{-}B}}_{\mathsf{S}\to\mathsf{P}}\right] &\le \mathsf{MC}\cdot\frac{1}{|\mathcal{B}|}\cdot\frac{|S|}{|\mathcal{A}|}\\ &= n\cdot\psi(n)\cdot\frac{1}{n/2}\cdot\frac{\epsilon(n)\cdot n-1}{n/2}\\ &\le n\cdot\psi(n)\cdot\frac{1}{n/2}\cdot\frac{\epsilon(n)\cdot n}{n/2}\\ &= 4\cdot\psi(n)\cdot\epsilon(n)\\ &= \frac{4\cdot\epsilon(n)}{12\cdot\epsilon(n)}=\frac{1}{3}.\end{aligned}$$

◁

Similarly, denote by $\mathcal{E}^{\mathsf{main}}_{\mathsf{P}\to\mathsf{S}}$ the event that P_{I^*} sends a message to a party in S in AttackMain, such that no party in S sent a message to P_{I^*} in any prior round; i.e., changing the order from $\mathcal{E}^{\mathsf{main}}_{\mathsf{S}\to\mathsf{P}}$. We upper bound the probability of $\mathcal{E}^{\mathsf{main}}_{\mathsf{P}\to\mathsf{S}}$ in an analogous manner.

▷ Claim 9. $\Pr\left[\mathcal{E}^{\mathsf{main}}_{\mathsf{P}\to\mathsf{S}}\right]\le\frac{1}{3}$.

Proof. Consider a different adversary for π, denoted Adv_A, that statically corrupts all parties in $\mathcal{A}$ and crashes them. Let AttackCrashA be a random variable defined by running the protocol π with the adversary Adv_A over SetupAndCoins, in which the honest sender's input is 0. That is, AttackCrashA consists of a vector of $n/2+1$ views: of the honest parties in $\mathcal{B}$, where the i^{th} view is denoted by $\textsc{view}^{\mathsf{crash\text{-}A}}_i$, and the sender P_1 denoted by $\textsc{view}^{\mathsf{crash\text{-}A}}_1$. Each view consists of the setup information R_i, the input 0 for P_1, and the set of received messages in each round. Specifically,

$$\textsc{AttackCrashA}=\left(\textsc{view}^{\mathsf{crash\text{-}A}}_i\right)_{i\in\mathcal{B}\cup\{1\}}.$$

Denote by $\mathcal{E}^{\mathsf{crash\text{-}A}}_{\mathsf{P}\to\mathsf{S}}$ the event that P_{I^*} sends a message to a party in S in AttackCrashA, and no party in S sent a message to P_{I^*} in any prior round. As long as P_{I^*} does not receive a message from parties in S until some round ρ in AttackMain, its view is identically distributed as its view in AttackCrashA up until round ρ. Therefore,

$$\Pr\left[\mathcal{E}^{\mathsf{main}}_{\mathsf{P}\to\mathsf{S}}\right]=\Pr\left[\mathcal{E}^{\mathsf{crash\text{-}A}}_{\mathsf{P}\to\mathsf{S}}\right].$$

An analogue analysis to the previous case shows that $\Pr\left[\mathcal{E}^{\mathsf{crash\text{-}A}}_{\mathsf{P}\to\mathsf{S}}\right]\le 1/3$, as desired. ◁

Combined together, we get that

$$\Pr\left[\neg\mathcal{E}^{\mathsf{main}}_{\mathsf{disconnect}}\right]=\Pr\left[\mathcal{E}^{\mathsf{main}}_{\mathsf{S}\to\mathsf{P}}\cup\mathcal{E}^{\mathsf{main}}_{\mathsf{P}\to\mathsf{S}}\right]\le\Pr\left[\mathcal{E}^{\mathsf{main}}_{\mathsf{S}\to\mathsf{P}}\right]+\Pr\left[\mathcal{E}^{\mathsf{main}}_{\mathsf{P}\to\mathsf{S}}\right]\le\frac{2}{3}.$$

Therefore, $\Pr\left[\mathcal{E}^{\mathsf{main}}_{\mathsf{disconnect}}\right]\ge 1/3$. This concludes the proof of Lemma 7. ◀

We proceed to show that conditioned on $\mathcal{E}^{\mathsf{main}}_{\mathsf{disconnect}}$, *agreement* of the protocol π is broken. Denote by Y^{main}_i the random variable denoting the output of P_i according to AttackMain. Further, denote by J^* the random variable corresponding to the minimal value in S.

▶ **Lemma 10.** $\Pr\left[Y^{\mathsf{main}}_{I^*}\ne Y^{\mathsf{main}}_{J^*}\mid\mathcal{E}^{\mathsf{main}}_{\mathsf{disconnect}}\right]\ge 1-\mathsf{negl}(\kappa)$.

Proof. We begin by showing that conditioned on $\mathcal{E}^{\mathsf{main}}_{\mathsf{disconnect}}$, party P_{I^*} outputs 0 with overwhelming probability.

▷ Claim 11. $\Pr\left[Y_{I^*}^{\mathsf{main}} = 0 \mid \mathcal{E}_{\mathsf{disconnect}}^{\mathsf{main}}\right] \geq 1 - \mathsf{negl}(\kappa)$.

Proof. Consider again the adversary Adv_A that statically corrupts all parties in $\mathcal{A}$ and crashes them, with the corresponding random variable AttackCrashA. Denote by $\mathcal{E}_{\mathsf{disconnect}}^{\mathsf{crash\text{-}A}}$ the event that P_{I^*} does not send any message to parties in S (according to $\textsc{view}_{I^*}^{\mathsf{crash\text{-}A}}$). It holds that

$$\Pr\left[\mathcal{E}_{\mathsf{disconnect}}^{\mathsf{crash\text{-}A}}\right] = \Pr\left[\neg\mathcal{E}_{\mathsf{P}\to\mathsf{S}}^{\mathsf{crash\text{-}A}}\right] = 1 - \Pr\left[\mathcal{E}_{\mathsf{P}\to\mathsf{S}}^{\mathsf{crash\text{-}A}}\right] \geq 2/3.$$

First, since the sender is honest and has input 0, by *validity* all honest parties in $\mathcal{B}$ output 0 in such execution, except for negligible probability. This holds even conditioned on $\mathcal{E}_{\mathsf{disconnect}}^{\mathsf{crash\text{-}A}}$ (since $\mathcal{E}_{\mathsf{disconnect}}^{\mathsf{crash\text{-}A}}$ occurs with noticeable probability). Denote by $Y_i^{\mathsf{crash\text{-}A}}$ the random variable denoting the output of P_i according to AttackCrashA. Then,

$$\Pr\left[Y_{I^*}^{\mathsf{crash\text{-}A}} = 0 \;\middle|\; \mathcal{E}_{\mathsf{disconnect}}^{\mathsf{crash\text{-}A}}\right] \geq 1 - \mathsf{negl}(\kappa). \tag{1}$$

Second, note that conditioned on $\mathcal{E}_{\mathsf{disconnect}}^{\mathsf{crash\text{-}A}}$ (by an analogous analysis of Lemma 7, this probability is non-zero), the view of P_{I^*} is identically distributed in AttackCrashA as its view in AttackMain conditioned on $\mathcal{E}_{\mathsf{disconnect}}^{\mathsf{main}}$. Indeed, conditioned on $\mathcal{E}_{\mathsf{disconnect}}^{\mathsf{main}}$, party P_{I^*} receives messages only from corrupt parties in AttackMain, which are consistently simulating precisely this execution where $\mathcal{A}$ has crashed and the sender has input 0. Therefore,

$$\Pr\left[Y_{I^*}^{\mathsf{crash\text{-}A}} = 0 \;\middle|\; \mathcal{E}_{\mathsf{disconnect}}^{\mathsf{crash\text{-}A}}\right] = \Pr\left[Y_{I^*}^{\mathsf{main}} = 0 \;\middle|\; \mathcal{E}_{\mathsf{disconnect}}^{\mathsf{main}}\right]. \tag{2}$$

The proof follows from Equations 1 and 2. This concludes the proof of Claim 11. ◁

We proceed to show that, conditioned on $\mathcal{E}_{\mathsf{disconnect}}^{\mathsf{main}}$, parties in S output 1 with overwhelming probability under the attack of Adv_1. Recall that J^* denotes the random variable corresponding to the minimal value in S.

▷ Claim 12. $\Pr\left[Y_{J^*}^{\mathsf{main}} = 1 \mid \mathcal{E}_{\mathsf{disconnect}}^{\mathsf{main}}\right] \geq 1 - \mathsf{negl}(\kappa)$.

Proof. The proof follows in nearly an identical manner. Namely, consider the adversary Adv_B that statically corrupts all parties in $\mathcal{B}$ and crashes them, and the random variable AttackCrashB. Denote by $\mathcal{E}_{\mathsf{disconnect}}^{\mathsf{crash\text{-}B}}$ the event that for every $J \in S$, party P_J does not send any message to P_{I^*} (according to $\textsc{view}_J^{\mathsf{crash\text{-}B}}$). It holds that

$$\Pr\left[\mathcal{E}_{\mathsf{disconnect}}^{\mathsf{crash\text{-}B}}\right] = \Pr\left[\neg\mathcal{E}_{\mathsf{S}\to\mathsf{P}}^{\mathsf{crash\text{-}B}}\right] = 1 - \Pr\left[\mathcal{E}_{\mathsf{S}\to\mathsf{P}}^{\mathsf{crash\text{-}B}}\right] \geq 2/3.$$

Since the sender is honest and has input 1, by *validity* all honest parties in $\mathcal{A}$ output 1 except for negligible probability. This holds even conditioned on $\mathcal{E}_{\mathsf{disconnect}}^{\mathsf{crash\text{-}B}}$ (since $\mathcal{E}_{\mathsf{disconnect}}^{\mathsf{crash\text{-}A}}$ occurs with noticeable probability). Denote by $Y_i^{\mathsf{crash\text{-}B}}$ the random variable denoting the output of P_i according to AttackCrashB, and recall that J^* corresponds to the minimal value in S. Then,

$$\Pr\left[Y_{J^*}^{\mathsf{crash\text{-}B}} = 1 \;\middle|\; \mathcal{E}_{\mathsf{disconnect}}^{\mathsf{crash\text{-}B}}\right] \geq 1 - \mathsf{negl}(\kappa). \tag{3}$$

Conditioned on $\mathcal{E}_{\mathsf{disconnect}}^{\mathsf{crash\text{-}B}}$, the view of P_{J^*} is identically distributed in AttackCrashB as its view in AttackMain conditioned in $\mathcal{E}_{\mathsf{disconnect}}^{\mathsf{main}}$. Therefore,

$$\Pr\left[Y_{J^*}^{\mathsf{crash\text{-}B}} = 1 \;\middle|\; \mathcal{E}_{\mathsf{disconnect}}^{\mathsf{crash\text{-}B}}\right] = \Pr\left[Y_{J^*}^{\mathsf{main}} = 1 \;\middle|\; \mathcal{E}_{\mathsf{disconnect}}^{\mathsf{main}}\right]. \tag{4}$$

The proof follows from Equations 3 and 4. This concludes the proof of Claim 12. ◁

Since P_{I^*} and P_{J^*} are honest, the proof of Lemma 10 follows from Claim 11 and Claim 12. ◀

Collectively, we have demonstrated an adversarial strategy Adv_1 that violates the agreement property of protocol π with noticeable probability:

$$\begin{aligned}\Pr\left[Y_{I^*}^{\mathsf{main}} \neq Y_{J^*}^{\mathsf{main}}\right] &= \Pr\left[Y_{I^*}^{\mathsf{main}} \neq Y_{J^*}^{\mathsf{main}} \mid \mathcal{E}_{\mathsf{disconnect}}^{\mathsf{main}}\right] \cdot \Pr\left[\mathcal{E}_{\mathsf{disconnect}}^{\mathsf{main}}\right] \\ &+ \Pr\left[Y_{I^*}^{\mathsf{main}} \neq Y_{J^*}^{\mathsf{main}} \mid \neg\mathcal{E}_{\mathsf{disconnect}}^{\mathsf{main}}\right] \cdot \Pr\left[\neg\mathcal{E}_{\mathsf{disconnect}}^{\mathsf{main}}\right] \\ &\geq \Pr\left[Y_{I^*}^{\mathsf{main}} \neq Y_{J^*}^{\mathsf{main}} \mid \mathcal{E}_{\mathsf{disconnect}}^{\mathsf{main}}\right] \cdot \Pr\left[\mathcal{E}_{\mathsf{disconnect}}^{\mathsf{main}}\right] \\ &\geq (1 - \mathsf{negl}(\kappa)) \cdot \frac{1}{3}.\end{aligned}$$

Note that the attack succeeds for any choice of distribution for setup information, and that the adversarial strategy runs in polynomial time, thus applying even in the presence of computational hardness assumptions. This concludes the proof of Theorem 6. ◀

4 Locality Lower Bound for Adaptive Corruptions

We proceed with the proof of Theorem 3. Here we show how a weakly adaptive adversary that can corrupt $n/2 + k$ parties can target any party of its choice and force a that party to communicate with k neighbors. We refer to Section 1.2 for a high-level overview of the attack.

▶ **Theorem 13** (Theorem 3, restated). *Let $0 < k < (n-1)/2$ and let π be an n-party broadcast protocol secure against $t = n/2 + k$ adaptive corruptions. Then, for any non-sender party P_{i^*} there exists a PPT adversary that can force the locality of P_{i^*} to be larger than k, except for negligible probability.*

Proof. Let π be a broadcast protocol that is secure against $t = n/2 + k$ adaptive corruptions. Without loss of generality, we assume that the setup information sampled before the beginning of the protocol $(r_1, \ldots, r_n) \leftarrow \mathcal{D}_\pi$ includes the random string used by each party. That is, every party P_i generates its messages in each round as a function of r_i, possibly its input (if P_i is the sender), and its incoming messages in prior rounds. Again, without loss of generality, let P_1 denote be the sender. Further, fix the party P_{i^*}, and split the remaining parties (without P_1 and P_{i^*}) to two equal-size subsets $\mathcal{S}_0$ and $\mathcal{S}_1$ (for simplicity, assume that n is even).

Consider the following adversary Adv that proceeds as follows:

1. Wait for the setup phase to complete. Later on, whenever corrupting a party P_i, the adversary receive its setup information r_i.
2. Corrupt the sender P_1.
3. Toss a random bit $b \leftarrow \{0, 1\}$ and corrupt all parties in $\mathcal{S}_{1-b}$.
4. Maintain two independent executions, denoted Exec_0 and Exec_1, as follows.
   - In the execution Exec_b, the adversary runs in its head a copy of the sender, denoted P_1^b, honestly running on input b and setup r_1. The adversary communicates on behalf of the virtual party P_1^b, and eventually corrupted parties in $\mathcal{S}_b$, with all honest parties $\mathcal{S}_b \cup \{\mathsf{P}_{i^*}\}$ according to this execution. The virtual parties in $\mathcal{S}_{1-b}$ are emulated as crashed in this execution.

     Whenever P_{i^*} sends a message to a party $\mathsf{P}_i \in \mathcal{S}_b$ this party gets corrupted and ignores this message (i.e., the adversary does not deliver messages from P_{i^*} to P_i).

- In the execution Exec_{1-b}, the adversary runs in its head the parties in $\mathcal{S}_{1-b}$ honestly on their setup information $\{r_i \mid \mathsf{P}_i \in \mathcal{S}_{1-b}\}$ and a copy of the sender, denoted P_1^{1-b}, running on input $1-b$ and setup r_1. The adversary communicates on behalf of the virtual parties in $(\mathcal{S}_{1-b}) \cup \{\mathsf{P}_1^{1-b}\}$ with P_{i^*} according to this execution. The honest parties in $\mathcal{S}_b$ are emulated as crashed in this execution; that is, the adversary drops every message sent by the virtual parties in $(\mathcal{S}_{1-b}) \cup \{\mathsf{P}_1^{1-b}\}$ to $\mathcal{S}_b$ and does not deliver any message from $\mathcal{S}_b$ to these parties.
  Whenever P_{i^*} sends a message to a party $\mathsf{P}_i \in \mathcal{S}_{1-b}$ this party ignores this message (i.e., the adversary does not deliver the message to P_i).

We start by defining a few notations. Consider the following random variables

$$\textsc{SetupAndCoins} = (R_1, \ldots, R_n, B)\,,$$

where $R_1, \ldots, R_n$ are distributed according to $\mathcal{D}_\pi$, and B takes a value uniformly at random in $\{0,1\}$. During the proof, R_i represents the setup information (including private randomness) of party P_i, whereas B corresponds to the adversarial choice of which set to corrupt. Unless stated otherwise, all probabilities are taken over these random variables.

Let $\textsc{AttackMain}$ be the random variable defined by running the protocol π with the adversary Adv over $\textsc{SetupAndCoins}$. That is, $\textsc{AttackMain}$ consists of a vector of $n+1$ views: of the parties in $S_b \cup \{\mathsf{P}_{I^*}\}$, of the corrupted parties in $\mathcal{S}_{1-b}$, where the i^{th} view is denoted by $\textsc{view}_i^{\mathsf{main}}$, and of two copies of the sender P_1^0 and P_1^1, denoted $\textsc{view}_{\text{1-0}}^{\mathsf{main}}$ and $\textsc{view}_{\text{1-1}}^{\mathsf{main}}$, respectively. Each view consists of the setup information R_i, possibly the input (for the sender), and the set of received messages in each round. Specifically,

$$\textsc{AttackMain} = \left(\textsc{view}_{\text{1-0}}^{\mathsf{main}}, \textsc{view}_{\text{1-1}}^{\mathsf{main}}, \textsc{view}_2^{\mathsf{main}}, \ldots, \textsc{view}_n^{\mathsf{main}}\right).$$

Denote by $\mathcal{E}_{\mathsf{low\text{-}locality}}^{\mathsf{main}}$ the event that the output-locality of P_{I^*} is at most k in $\textsc{AttackMain}$; that is, P_{I^*} sends messages to at most k parties (according to $\textsc{view}_{I^*}^{\mathsf{main}}$). If $\Pr[\mathcal{E}_{\mathsf{low\text{-}locality}}^{\mathsf{main}}] = \mathsf{negl}(\kappa)$, then the proof is completed. Otherwise, it holds that $\Pr[\mathcal{E}_{\mathsf{low\text{-}locality}}^{\mathsf{main}}]$ is non-negligible (in particular, $\Pr[\mathcal{E}_{\mathsf{low\text{-}locality}}^{\mathsf{main}}] > 0$). We will show that conditioned on $\mathcal{E}_{\mathsf{low\text{-}locality}}^{\mathsf{main}}$, *agreement* is broken. Denote by Y_i^{main} the random variable denoting the output of P_i according to $\textsc{AttackMain}$.

First, note that conditioned on $\mathcal{E}_{\mathsf{low\text{-}locality}}^{\mathsf{main}}$, the view of P_{i^*} is identically distributed no matter which set $\mathcal{S}_b$ is corrupted.

▷ Claim 14. For every $\beta \in \{0,1\}$ it holds that

$$\Pr\left[Y_{i^*}^{\mathsf{main}} = \beta \mid \mathcal{E}_{\mathsf{low\text{-}locality}}^{\mathsf{main}} \cap (B=0)\right] = \Pr\left[Y_{i^*}^{\mathsf{main}} = \beta \mid \mathcal{E}_{\mathsf{low\text{-}locality}}^{\mathsf{main}} \cap (B=1)\right].$$

Proof. By the construction of Adv, for each $\beta \in \{0,1\}$ party P_{i^*} receives from the parties in $\mathcal{S}_\beta$ and from P_1 messages that correspond to an execution by honest parties on sender input β as if the parties in $\mathcal{S}_{1-\beta}$ all crashed, and where every party in $\mathcal{S}_\beta$ that P_{i^*} talks to ignores its message (since P_{i^*} talks to at most k parties conditioned on $\mathcal{E}_{\mathsf{low\text{-}locality}}^{\mathsf{main}}$, the adversary can corrupt all of them).

Further, P_{i^*} receives from the parties in $\mathcal{S}_{1-\beta}$ and from P_1 messages that correspond to a simulated execution by honest parties on sender input $1-\beta$ as if the parties in $\mathcal{S}_\beta$ all crashed, and where every party in $\mathcal{S}_{1-\beta}$ that P_{i^*} talks to ignores its message.

Clearly, the view of P_{i^*} is identically distributed in both cases; hence, its output bit is identically distributed as well. ◁

We proceed to show that conditioned on $\mathcal{E}^{\mathsf{main}}_{\mathsf{low\text{-}locality}}$, party P_{i^*} outputs 0 for $B = 0$ and outputs 1 for $B = 1$.

▷ Claim 15. For every $\beta \in \{0, 1\}$ it holds that

$$\Pr\left[Y^{\mathsf{main}}_{i^*} = \beta \mid \mathcal{E}^{\mathsf{main}}_{\mathsf{low\text{-}locality}} \cap (B = \beta)\right] = 1 - \mathsf{negl}(\kappa).$$

Proof. Consider a different adversary for π, denoted Adv_β, that proceeds as follows:

1. Wait for the setup phase to complete.
2. Corrupt all parties in $\mathcal{S}_{1-\beta}$ and crash them.
3. Whenever P_{i^*} sends a message to a party $\mathsf{P}_i \in \mathcal{S}_\beta$ this party gets corrupted and ignores this message (i.e., the adversary does not deliver messages from P_{i^*} to P_i).

Let $\textsc{AttackCrashS}_\beta$ be the random variable defined by running the protocol π with the adversary Adv_β over $\textsc{SetupAndCoins}$, in which the honest sender's input is β. That is, $\textsc{AttackCrashS}_\beta$ consists of a vector of $n/2$ views: of the parties in $\mathcal{S}_\beta \cup \{\mathsf{P}_{i^*}\}$ (both honest and corrupted), where the ith view is denoted by $\textsc{view}^{\mathsf{crash\text{-}S}_\beta}_i$, and the sender P_1 denoted by $\textsc{view}^{\mathsf{crash\text{-}S}_\beta}_1$. Each view consists of the setup information R_i, the input β for P_1, and the set of received messages in each round. Specifically,

$$\textsc{AttackCrashS}_\beta = \left(\textsc{view}^{\mathsf{crash\text{-}S}_\beta}_i\right)_{i \in \mathcal{S}_\beta \cup \{1, i^*\}}.$$

Let us denote by $\mathcal{E}^{\mathsf{crashS}_\beta}_{\mathsf{low\text{-}locality}}$ the event that the output-locality of P_{I^*} is at most k in $\textsc{AttackCrashS}_\beta$; that is, P_{I^*} sends messages to at most k parties (according to $\textsc{view}^{\mathsf{crash\text{-}S}_\beta}_{i^*}$). If $\Pr[\mathcal{E}^{\mathsf{crashS}_\beta}_{\mathsf{low\text{-}locality}}] = \mathsf{negl}(\kappa)$, then Adv_β can force the locality of P_{I^*} to be high in $\textsc{AttackCrashS}_\beta$, and the proof is completed. Otherwise, it holds that $\Pr[\mathcal{E}^{\mathsf{crashS}_\beta}_{\mathsf{low\text{-}locality}}]$ is non-negligible.

Note that since $|\mathcal{S}_\beta| = (n-1)/2$ and $k < (n-1)/2$, then conditioned on $\mathcal{E}^{\mathsf{crashS}_\beta}_{\mathsf{low\text{-}locality}}$ there exists at least one remaining honest party in $\mathcal{S}_\beta$ at the end of the execution with Adv_β. By *validity*, each such honest party must output β with overwhelming probability. Denote by $Y^{\mathsf{crash\text{-}S}_\beta}_i$ the random variable denoting the output of P_i according to $\textsc{AttackCrashS}_\beta$. Denote by J^* the random variable corresponding to the minimal index of an honest party in $\mathcal{S}_\beta$ at the end of the execution with Adv_β. Then

$$\Pr\left[Y^{\mathsf{crash\text{-}S}_\beta}_{J^*} = \beta \mid \mathcal{E}^{\mathsf{crashS}_\beta}_{\mathsf{low\text{-}locality}}\right] = 1 - \mathsf{negl}(\kappa). \tag{5}$$

Further, note that the set of all honest parties in $\mathcal{S}_\beta$ and their joint view in an execution with Adv_β conditioned on $\mathcal{E}^{\mathsf{crashS}_\beta}_{\mathsf{low\text{-}locality}}$ is identically distributed as in an execution with Adv conditioned on $\mathcal{E}^{\mathsf{main}}_{\mathsf{low\text{-}locality}} \cap (B = \beta)$. Therefore,

$$\Pr\left[Y^{\mathsf{crash\text{-}S}_\beta}_{J^*} = \beta \mid \mathcal{E}^{\mathsf{crashS}_\beta}_{\mathsf{low\text{-}locality}}\right] = \Pr\left[Y^{\mathsf{main}}_{J^*} = \beta \mid \mathcal{E}^{\mathsf{main}}_{\mathsf{low\text{-}locality}} \cap (B = \beta)\right]. \tag{6}$$

Finally, by *agreement*, since both P_{J^*} and P_{i^*} are honest at the end of the execution with Adv, conditioned on $\mathcal{E}^{\mathsf{main}}_{\mathsf{low\text{-}locality}} \cap (B = \beta)$, it holds that

$$\Pr\left[Y^{\mathsf{main}}_{i^*} = \beta \mid \mathcal{E}^{\mathsf{main}}_{\mathsf{low\text{-}locality}} \cap (B = \beta)\right] = \Pr\left[Y^{\mathsf{main}}_{J^*} = \beta \mid \mathcal{E}^{\mathsf{main}}_{\mathsf{low\text{-}locality}} \cap (B = \beta)\right] - \mathsf{negl}(\kappa). \tag{7}$$

The claim follows from Equations 5, 6, and 7. ◁

By Claim 14 and Claim 15 it follows that $\Pr[\mathcal{E}^{\mathsf{main}}_{\mathsf{low\text{-}locality}}] = \mathsf{negl}(\kappa)$. This concludes the proof of Theorem 13. ◀

References

1 Ittai Abraham, T.-H. Hubert Chan, Danny Dolev, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. Communication complexity of byzantine agreement, revisited. In *Proceedings of the 38th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 317–326, 2019.

2 Nicolas Alhaddad, Sourav Das, Sisi Duan, Ling Ren, Mayank Varia, Zhuolun Xiang, and Haibin Zhang. Balanced byzantine reliable broadcast with near-optimal communication and improved computation. In *Proceedings of the 41st Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 399–417, 2022.

3 Donald Beaver. Precomputing oblivious transfer. In *14th Annual International Cryptology Conference (CRYPTO)*, pages 97–109, 1995.

4 Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 27–30, 1983.

5 Piotr Berman, Juan A Garay, and Kenneth J Perry. Bit optimal distributed consensus. *Computer Science Research*, pages 313–322, 1992.

6 Erica Blum, Jonathan Katz, Chen-Da Liu-Zhang, and Julian Loss. Asynchronous Byzantine agreement with subquadratic communication. In *Proceedings of the 18th Theory of Cryptography Conference (TCC), part I*, pages 353–380, 2020.

7 Elette Boyle, Ran Cohen, Deepesh Data, and Pavel Hubáček. Must the communication graph of MPC protocols be an expander? In *38th Annual International Cryptology Conference (CRYPTO), part III*, pages 243–272, 2018.

8 Elette Boyle, Ran Cohen, and Aarushi Goel. Breaking the O($\sqrt{}$n)-bit barrier: Byzantine agreement with polylog bits per party. In *Proceedings of the 40th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 319–330, 2021.

9 Elette Boyle, Shafi Goldwasser, and Stefano Tessaro. Communication locality in secure multi-party computation - how to run sublinear algorithms in a distributed setting. In *Proceedings of the 10th Theory of Cryptography Conference (TCC)*, pages 356–376, 2013.

10 Nicolas Braud-Santoni, Rachid Guerraoui, and Florian Huc. Fast Byzantine agreement. In *Proceedings of the 32th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 57–64, 2013.

11 Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings of the 42nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 136–145, 2001.

12 Ran Canetti, Uriel Feige, Oded Goldreich, and Moni Naor. Adaptively secure multi-party computation. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing (STOC)*, pages 639–648, 1996.

13 T.-H. Hubert Chan, Rafael Pass, and Elaine Shi. Sublinear-round byzantine agreement under corrupt majority. In *Proceedings of the 23rd International Conference on the Theory and Practice of Public-Key Cryptography (PKC), part II*, pages 246–265, 2020.

14 Jing Chen and Silvio Micali. Algorand: A secure and efficient distributed ledger. *Theoretical Computer Science*, 777:155–183, 2019.

15 Brian A Coan and Jennifer L Welch. Modular construction of a byzantine agreement protocol with optimal message bit complexity. *Information and Computation*, 97(1):61–85, 1992.

16 Ran Cohen, Juan A. Garay, and Vassilis Zikas. Completeness theorems for adaptively secure broadcast, 2023. CRYPTO '23 (to appear).

17 Shir Cohen, Idit Keidar, and Alexander Spiegelman. Not a COINcidence: Sub-quadratic asynchronous Byzantine agreement WHP. In *Proceedings of the 34th International Symposium on Distributed Computing (DISC)*, pages 25:1–25:17, 2020.

18 Sourav Das, Zhuolun Xiang, and Ling Ren. Asynchronous data dissemination and its applications. In *Proceedings of the 28th ACM Conference on Computer and Communications Security (CCS)*, pages 2705–2721, 2021.

19 Alan J. Demers, Daniel H. Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard E. Sturgis, Daniel C. Swinehart, and Douglas B. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 1–12, 1987.

20 Danny Dolev. The byzantine generals strike again. *J. Algorithms*, 3(1):14–30, 1982.

21 Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for Byzantine agreement. *Journal of the ACM*, 32(1):191–204, 1985.

22 Danny Dolev and H. Raymond Strong. Authenticated algorithms for Byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

23 Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

24 Cynthia Dwork, David Peleg, Nicholas Pippenger, and Eli Upfal. Fault tolerance in networks of bounded degree. *SIAM Journal on Computing*, 17(5):975–988, 1988.

25 Paul Feldman. *Optimal Algorithms for Byzantine Agreement.* PhD thesis, Stanford University, 1988. URL: `https://dspace.mit.edu/handle/1721.1/14368`.

26 Michael J. Fischer and Nancy A. Lynch. A lower bound for the time to assure interactive consistency. *Information Processing Letters*, 14(4):183–186, 1982.

27 Michael J. Fischer, Nancy A. Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1(1):26–39, 1986.

28 Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. In *Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 1–7, 1983.

29 Matthias Fitzi, Chen-Da Liu-Zhang, and Julian Loss. A new way to achieve round-efficient byzantine agreement. In *Proceedings of the 40th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 355–362, 2021.

30 Juan A. Garay, Jonathan Katz, Ranjit Kumaresan, and Hong-Sheng Zhou. Adaptively secure broadcast, revisited. In *Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 179–186, 2011.

31 Martin Hirt and Vassilis Zikas. Adaptively secure broadcast. In *29th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 466–485, 2010.

32 Dan Holtby, Bruce M. Kapron, and Valerie King. Lower bound for scalable Byzantine agreement. *Distributed Computing*, 21(4):239–248, 2008.

33 Richard M. Karp, Christian Schindelhauer, Scott Shenker, and Berthold Vöcking. Randomized rumor spreading. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 565–574, 2000.

34 Valerie King and Jared Saia. From almost everywhere to everywhere: Byzantine agreement with $\tilde{\mathrm{o}}(\mathrm{n}^{3/2})$ bits. In *Proceedings of the 23th International Symposium on Distributed Computing (DISC)*, pages 464–478, 2009.

35 Valerie King and Jared Saia. Breaking the $O(n^2)$ bit barrier: Scalable Byzantine agreement with an adaptive adversary. *Journal of the ACM*, 58(4):18:1–18:24, 2011. A preliminary version appeared at PODC'10.

36 Valerie King, Jared Saia, Vishal Sanwalani, and Erik Vee. Scalable leader election. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 990–999, 2006.

37 Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

38 Chen-Da Liu-Zhang, Christian Matt, and Søren Eller Thomsen. Asymptotically optimal message dissemination with applications to blockchains. Cryptology ePrint Archive, Paper 2022/1723, 2022. URL: `https://eprint.iacr.org/2022/1723`.

39 Silvio Micali. Very simple and efficient Byzantine agreement. In *Proceedings of the 8th Annual Innovations in Theoretical Computer Science (ITCS) conference*, pages 6:1–6:1, 2017.

40 Silvio Micali, Michael O. Rabin, and Salil P. Vadhan. Verifiable random functions. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 120–130, 1999.

41 Atsuki Momose and Ling Ren. Optimal communication complexity of authenticated byzantine agreement. In *Proceedings of the 35th International Symposium on Distributed Computing (DISC)*, pages 32:1–32:16, 2021.

42 Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

43 Birgit Pfitzmann and Michael Waidner. Unconditional Byzantine agreement for any number of faulty processors. In *Proceedings of the 9th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 339–350, 1992.

44 Michael O. Rabin. Randomized byzantine generals. In *Proceedings of the 24th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 403–409, 1983.

45 Georgios Tsimos, Julian Loss, and Charalampos Papamanthou. Gossiping for communication-efficient broadcast. In *42nd Annual International Cryptology Conference (CRYPTO), part III*, pages 439–469, 2022.

Time and Space Optimal Massively Parallel Algorithm for the 2-Ruling Set Problem

Mélanie Cambus
Aalto University, Finland

Fabian Kuhn
University of Freiburg, Germany

Shreyas Pai
Aalto University, Finland

Jara Uitto
Aalto University, Finland

Abstract

In this work, we present a constant-round algorithm for the 2-ruling set problem in the Congested Clique model. As a direct consequence, we obtain a constant round algorithm in the MPC model with linear space-per-machine and optimal total space. Our results improve on the $O(\log\log\log n)$-round algorithm by [HPS, DISC'14] and the $O(\log\log\Delta)$-round algorithm by [GGKMR, PODC'18]. Our techniques can also be applied to the semi-streaming model to obtain an $O(1)$-pass algorithm.

Our main technical contribution is a novel sampling procedure that returns a small subgraph such that *almost* all nodes in the input graph are adjacent to the sampled subgraph. An MIS on the sampled subgraph provides a 2-ruling set for a large fraction of the input graph. As a technical challenge, we must handle the remaining part of the graph, which might still be relatively large. We overcome this challenge by showing useful structural properties of the remaining graph and show that running our process twice yields a 2-ruling set of the original input graph with high probability.

2012 ACM Subject Classification Theory of computation → Massively parallel algorithms; Theory of computation → Streaming models

Keywords and phrases Ruling Sets, Parallel Algorithms, Congested Clique, Massively Parallel Computing, Semi-Streaming

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.11

Related Version *Full Version*: `https://arxiv.org/abs/2306.00432`

Funding *Mélanie Cambus*: Academy of Finland, Grant 334238.
Shreyas Pai: Academy of Finland, Grant 334238.

1 Introduction

In this paper, we design and analyze a parallel algorithm for finding ruling sets. For a graph $G = (V, E)$ (with $|V| = n$ and $|E| = m$) and integer $\beta \geq 1$, a β-ruling set $S \subseteq V$ is a set of non-adjacent nodes such that for each node $u \in V$, there is a ruling set node $v \in S$ within β hops. This is a natural generalization of one of the most fundamental problems in parallel and distributed graph algorithms: Maximal Independent Set (MIS), which corresponds to a 1-ruling set. Ruling sets are closely related to clustering problems like Metric Facility Location, as fast algorithms for β-ruling sets imply fast algorithms for $O(\beta)$-approximate metric facility location [7, 18].

Our main contribution is an $O(1)$ round algorithm for 2-ruling sets in Congested Clique, improving on the $O(\log\log\log n)$ time algorithm by [16] and $O(\log\log\Delta)$ algorithm by [14], where Δ denotes the maximum degree of the input graph. While problems like minimum spanning tree [27] and $(\Delta+1)$-coloring [9] surprisingly admit constant round solutions in the Congested Clique model, the MIS problem and even β-ruling set for $\beta = O(1)$ have resisted attempts to obtain constant round algorithms. We make significant progress in this direction by giving the first $O(1)$ round algorithm for 2-ruling sets in Congested Clique.

The Congested Clique model [22], is a distributed synchronous message-passing model where each node is given its incident edges as an input and the nodes perform all-to-all communication in synchronous rounds. The crucial limitation is that the size of the messages sent between any pair of nodes, in a single round, is limited to $O(\log n)$ bits, where n is the number of nodes in the input graph. The goal is to minimize the number of required synchronous communication rounds.

As a consequence of our Congested Clique algorithm, we obtain a constant round algorithm for 2-ruling sets in the Linear Memory MPC model [19, 15, 5] and a constant pass algorithm in the semi-streaming model [10, 11, 26]. The Congested Clique and MPC implementations are asymptotically optimal in both time and space, as they use constant rounds and $O(n+m)$ total memory. In the semi-streaming model, one typically aims for very few passes, ideally just one. While we show that constant passes are enough to solve 2-ruling sets with $O(n)$ space, discovering the precise number of passes required is an interesting open question. The implementation details and definitions of the different models of computation can be found in Section 3. Our main results are captured in the following theorem.

▶ **Theorem** (Main Theorem). *There is a randomized parallel algorithm to the* 2*-ruling set problem that can be implemented in* $O(1)$ *rounds/passes (1) in the Congested Clique model, (2) in the MPC model with* $O(n)$ *words of local memory and* $O(n+m)$ *words of total memory, and (3) in the semi-streaming model with* $O(n)$ *words of space. The running time guarantee of the algorithm holds with high probability (w.h.p.)*[1].

1.1 Previous Works on Ruling Sets and MIS

Recall that a 2-ruling set S is a set of non-adjacent nodes, such that each node in the input graph has a node from S within its 2-hop neighborhood. This is a strictly looser requirement than the one of an MIS and hence, an MIS algorithm directly implies a 2-ruling set algorithm. The classic algorithms by Luby [23] and Alon, Babai, and Itai [2] yield $O(\log n)$ time algorithms for ruling sets. In Congested Clique and MPC, this was later improved to $\widetilde{O}(\sqrt{\log \Delta})$ [13] and the current state-of-the-art for MIS is $O(\log\log \Delta)$ rounds [14].

When focusing on the 2-ruling set problem, roughly a decade ago, [7] gave an expected $O(\log\log n)$ time algorithm and [16] gave an $O(\log\log n)$ time algorithm w.h.p. in the Congested Clique model. Combining the result of [16] with the $O(\log\log \Delta)$ algorithm for MIS, this was improved to $O(\log\log\log n)$ rounds (w.h.p.). The 2-ruling set problem can be solved *deterministically* in $O(\log\log n)$ rounds in the Congested Clique model [28]. On the other hand, [4] shows that $\Omega(\log\log n)$ rounds are required to compute an MIS in the Broadcast Congested Clique model.

The $O(\log\log \Delta)$ algorithm by [14] carries over to the semi-streaming model. We note that there is an older variant of this algorithm that also yields an $O(\log\log \Delta)$-pass randomized *greedy* MIS, which is given by picking a random permutation over the nodes [1]. Then the permutation is iterated over and, whenever possible, the current node is added to the MIS. However, this approach requires a polylogarithmic overhead in space and hence, does

[1] Following the standard, we say that an event holds with high probability if it holds with probability $1 - 1/n^c$ for a constant $c \geq 1$ we can choose.

not work in the Congested Clique model. While it is known that computing an MIS in a *single-pass* of the stream requires $\Omega(n^2)$ memory [8], there has been progress towards designing single-pass semi-streaming algorithms for 2-ruling sets, but the current approaches require significantly larger than $O(n)$ memory: [20] shows that it can be done in $O(n\sqrt{n})$ memory and this was improved to $O(n^{4/3})$ by [3]. The $\Omega(n^2)$-space lower bound for MIS was generalized to $(\alpha, \alpha - 1)$-ruling sets by [3]. An (α, β)-ruling set is a set S of nodes such that nodes within S are distance at least α apart and every node in $V \setminus S$ has a node in S within distance β. An MIS is a $(2, 1)$-ruling set, and a 2-ruling set is a $(2, 2)$-ruling set, in this paper we drop the α parameter for sake of convenience as it is always 2. A single-pass semi-streaming algorithm for 2-ruling set has not been ruled out.

1.2 A High-Level Technical Overview of Our Algorithm

Our strategy is to compute an MIS iteratively on subgraphs of size $O(n)$ until all nodes are covered. We begin with a sampling process where each node u is sampled independently with probability $1/\sqrt{\deg(u)}$. Call the set of sampled vertices V_{samp}. The intuition behind this sampling probability is to maximize the probability that each node has a sampled neighbor while ensuring $O(n)$ edges in the sampled subgraph $G[V_{\text{samp}}]$. To see why the sampled subgraph is small, assume we have a d-regular graph. Hence, each node is independently sampled with probability $1/\sqrt{d}$. For an edge $\{u, v\}$ to be sampled, both u and v must be sampled together. Therefore, an edge exists in the sampled graph with probability $1/d$, and we can say that there are at most $O(n)$ edges in the sampled graph in expectation (since the total number of edges is nd).

We show that the same expectation holds for general graphs by orienting the edges based on the degree of their end points and counting the number of outgoing sampled edges per node. In order to show that $G[V_{\text{samp}}]$ has $O(n)$ edges w.h.p., we face a technical challenge that the random choices for all edges are not independent. The dependencies disallow the use of standard Chernoff bounds, but we overcome the challenge by using the method of bounded differences to show concentration. For convenience, we state the concentration bounds in Section 4.

We classify nodes in the graph as either *good* or *bad*, based on how their initial degree compares with the sum of sampling probabilities of their neighbors:

▶ **Definition 1** (Good and Bad Nodes). *A node u is good if $\sum_{v \in N(u)} 1/\sqrt{\deg(v)} \geq \gamma \log \deg(u)$ (for a constant γ). If a node is not good, it is bad.*

Note that this definition is based entirely on the graph structure (i.e., degrees of all nodes). Furthermore, it is straightforward to detect which nodes are good and bad in $O(1)$ rounds.

By definition, good nodes are expected to have many sampled neighbors, so they have a relatively high probability of being covered by a node in V_{samp}. So we put the good nodes that do not have any sampled neighbors into a set V^* and process it later. We call V^* the nodes that are *set aside*. Therefore, the remaining graph of uncovered nodes only consists of bad nodes. In addition to all the good nodes without a sampled neighbor, we also add some bad nodes that are very likely to be covered (but weren't) into V^*. We use the fact that all nodes in V^* have a good chance of being covered to show that the graph $G[V^*]$ has $O(n)$ edges w.h.p. The main difficulty in proving this claim is that nodes are not put independently in V^*, and only nodes that are a certain distance apart are independent of each other. Moreover, the probability of a single node in V^* with degree d being covered is only $1 - 1/\text{poly}(d)$ which is not enough to do a simple union bound over all nodes. We overcome this challenge by showing that V^* can be partitioned into large sets of far apart nodes, which allows us to (1) boost the probability of each set being covered, and (2) apply a union bound over all sets.

Finally, we do an intricate counting argument that bounds the number of uncovered bad nodes. We do this by counting the number of edges that can exist between bad nodes and higher degree nodes, and show that the number of bad nodes with degree d is roughly bounded by $n/\sqrt{d}$. Note that this bound is not enough to show that the remaining graph is small. So we run the entire sampling and setting nodes aside process again to get the bound to be roughly n/d, which makes it straightforward to show that the remaining graph has $O(n)$ edges.

2 Parallel 2-Ruling-Set

In this section, we present a parallel algorithm for finding a 2-ruling set. For technical convenience, we assume that the maximum degree of the input graph is bounded by n^α for some constant $\alpha < 1/8$, and in Section 2.3 we show how to remove this assumption. When computing an MIS on the sampled vertices, we use a version of the Luby's algorithm that works as follows: in an iteration, each node picks a real number independently and uniformly at random in $[0, 1]$, local minima join the MIS, and these two steps are repeated until all nodes are either in the MIS or have a neighbor in the MIS. It is well known that this algorithm terminates in $O(\log n)$ iterations w.h.p. (see for example [12]).

Recalling the definition of good and bad nodes from Definition 1, let B_d be the set of bad nodes in G with (initial) degree in the range $[d, 2d)$. We describe the Parallel 2-Ruling-Set algorithm in Algorithm 1.

Algorithm 1 Parallel 2-Ruling-Set.

```
   Input: Graph $G = (V, E)$, with $\Delta \le n^\alpha$ ($\alpha < 1/8$). Each node $v \in V$ knows its degree
   $\deg(v)$ in $G$.
1: Each node $v \in G$ is independently sampled into $V_{\text{samp}}$ with probability $1/\sqrt{\deg(v)}$.
2: We put good nodes that do not have any neighbors in $V_{\text{samp}}$ in the set $V^*$.
3: Compute an MIS $I$ on $G_{\text{samp}} = G[V_{\text{samp}}]$ using Luby's algorithm first on the bad nodes
   and then on the rest of the nodes.
4: If any uncovered $u \in B_d$ has a neighbor $v$ that has at least $c\sqrt{d}\log^5(d)$ neighbors in $B_d$,
   we put $u$ in $V^*$.          ▷ $B_d$ is the set of bad nodes in $G$ with (initial) degree in $[d, 2d)$
5: Compute an MIS on $G[V^*]$.
6: Run Lines 1 to 5 on $G' = G[V \setminus (\mathcal{I} \cup N(\mathcal{I}) \cup N(N(\mathcal{I})))]$ where $\mathcal{I}$ is the set of MIS nodes.
7: Compute an MIS on the graph induced by the uncovered nodes.
   Output: The set of nodes that joined an MIS during the algorithm.
```

A node is considered covered if and only if it is at most 2-hops away from a node in the MIS, and two adjacent nodes can never join the MIS. Since the last step of the algorithm computes an MIS on the uncovered nodes, it is guaranteed to output a valid 2-ruling set, hence proving the following theorem.

▶ **Theorem 2.** *The Parallel 2-Ruling-Set algorithm (Algorithm 1) outputs a valid 2-ruling set.*

2.1 Structural Properties of the Subgraphs

We will now prove key structural properties of the different subgraphs on which we compute an MIS in Algorithm 1. This will allow for fast implementation of this algorithm in linear memory MPC, Congested Clique, and semi-streaming models (see Section 3). We first prove the fact that the sampled graph has a linear number of edges w.h.p.

▶ **Lemma 3.** *The sampled graph G_{samp} has $O(n)$ edges w.h.p.*

Proof. Let X be the random variable denoting the number of edges in G_{samp}. Let X_u be the indicator random variable for the event that u is sampled in V_{samp} and let Y_e be the indicator random variable for the event that edge e belongs to G_{samp}. We orient all edges from the end point with lower initial degree to higher initial degree. By the degree sum lemma, we have $X = \sum_{u \in V} \sum_{e \in \text{Out}(u)} Y_e$, where $\text{Out}(u)$ is defined as the set of outgoing edges of u.

Consider an oriented edge $e = (u, v)$ with $\deg(u) \le \deg(v)$. We have that both u and v are sampled with probability at most $1/\sqrt{\deg(u)}$, so the probability that e is in G_{samp} is at most $1/\deg(u)$. Therefore, $\mathbb{E}[Y_e] \le 1/\deg(u)$, and $\mathbb{E}[X] \le n$.

We can interpret X as a function of the random variables $X_u, u \in V$, where changing one coordinate changes X by at most $\Delta = n^\alpha$. Therefore, X follows the bounded differences property with bounds $c_u = n^\alpha$ for all $u \in V$. Since the X_u's are independent of each other, we can use Lemma 18 with $\mu = t = n$ to say that: $\Pr[X > 2n] \le 2\exp(n^2/n^{1+2\alpha}) \le 2\exp(n^{1-2\alpha}) \le 1/\text{poly}(n)$. ◀

As we observed earlier, good nodes expect to see a lot of sampled neighbors, therefore it is very unlikely that a good node has no sampled neighbors. The following lemma formalizes this intuition.

▶ **Lemma 4.** *In Line 2 of Algorithm 1, a good node u with $\deg(u) = d$ is added to V^* with probability $1/\text{poly}(d)$. This event is independent of the randomness (for sampling into V_{samp}) of nodes more than distance 1 from u in G.*

Proof. Since u is good, the sum of sampling probabilities of its neighbors is at least $\gamma \cdot \log d$. So we can bound the expected number of neighbors in V_{samp} as $\mathbb{E}\left[|N(u) \cap V_{\text{samp}}|\right] \ge \gamma \cdot \log d$. Since the sampling is done independently for each node, we can use Chernoff bound (Lemma 16) to compute the probability that no neighbor of u is in V_{samp}. We get that $\Pr\left[|N(u) \cap V_{\text{samp}}| = 0\right] < 1/d^{\gamma/2}$. Hence, u is added to V^* with probability at most $1/\text{poly}(d)$. This event only depends on the randomness of u and its neighbors in G, therefore it is independent of the randomness of nodes at distance more than 1 from u. ◀

On the other hand, bad nodes expect to have few sampled neighbors. So a sampled bad node will have a good probability of being a local minimum in the first iteration of Luby's algorithm. Therefore, nodes having many bad neighbors are very likely to have one such bad neighbor join the MIS, and hence all such bad neighbors are 2-hop covered.

▶ **Lemma 5.** *In Line 4 of Algorithm 1, each node $u \in B_d$ is added to V^* with probability at most $1/\text{poly}(d)$. This happens independently of the randomness (for sampling into V_{samp} and for the first Luby round when computing I) of nodes more than distance 3 from u in G.*

Proof. Recall that u is added to V^* if there is a node $v \in N(u)$ such that v has more than $c\sqrt{d} \cdot \log^5 d$ neighbors in B_d. Let v be an arbitrary such node and let A_u be a subset of $N(v) \cap B_d$ such that $|A_u| = c\sqrt{d}\log^5 d$ and $u \in A_u$. We will show that at least one node of A_u joins the MIS in Line 3 of Algorithm 1 in the first Luby round with probability at least $1 - 1/\text{poly}(d)$.

A node $w \in A_u$ joins I in the first Luby round iff w is sampled and if w has the smallest random number (in the first Luby round) among all its sampled neighbors. Note that if one node of A_u joins the MIS only depends on the randomness of the bad nodes in $A_u \cup N(A_u)$, which are a subset of the 2-hop neighborhood of v and thus of the 3-hop neighborhood of u.

First, note that the number of nodes in $A_u \cup N(A_u)$ is at most $O(d^{3/2}\log^5(d))$ because $A_u \subseteq B_d$, so every node has at most $2d$ neighbors in $N(A_u)$.

Let S_u be the set of sampled nodes in A_u. Every node in $N(A_u)$ with at most $O(\sqrt{d}\log^2 d)$ neighbors in A_u has at most $O(\log^2 d)$ neighbors in S_u with probability at least $1 - 1/poly(d)$ (by using Chernoff bound Lemma 16 and then union bound over all such nodes in $N(A_u)$). On the other hand, every node $w \in N(A_u)$ with $\Omega(\sqrt{d}\log^2 d)$ neighbors in A_u has $\Omega(\log^2 d)$ neighbors in S_u in expectation. If w is a good node, it does not participate in the first Luby round carried out by the nodes in A_u, and if w is a bad node, the expected number of sampled neighbors of w is at most $\gamma \log(\deg(w))$ and we therefore have $\log(\deg(w)) = \Omega(\log^2 d)$ and thus $\deg(w) = \exp(\Omega(\log^2 d))$. The probability that w is sampled is therefore $\ll 1/poly(d)$.

With probability $1 - 1/\text{poly}(d)$, all the sampled bad nodes in $N(A_u)$ therefore have at most $O(\log^2 d)$ sampled neighbors in A_u. Moreover, all the sampled nodes in A_u have at most $O(\log d)$ overall sampled neighbors, since A_u is a subset of B_d, it has at most $\gamma \log 2d$ sampled neighbors in expectation. Using a Chernoff bound (Lemma 16) gives us that each node in A_u has $O(\log d)$ sampled neighbors with probability $1 - 1/poly(d)$. In the following, we condition on this event happening.

Consider the graph G_S induced by the sampled nodes in $A_u \cup N(A_u)$. For any two nodes $x, y \in S_u$ that are at distance at least 3 in G_S, the events that x and y join the MIS I in the first Luby step are independent. For every node $x \in S_u$, there are at most $O(\log^3 d)$ other nodes in S_u at distance at most 2 in G_S (at most $O(\log d)$ direct neighbors and because the direct neighbors can be in $N(A_u)$ at most $O(\log^3 d)$ 2-hop neighbors). By greedily picking nodes in S_u, we can therefore find a set of size $\Omega(|S_u|/\log^3 d)$ of nodes in S_u that independently join the MIS I in the first Luby step. Because with probability $1 - 1/\text{poly}(d)$, $|S_u| = \Omega(\log^5 d)$, we have $\Omega(|S_u|/\log^3 d) = \Omega(\log^2 d)$.

Each of those nodes independently joins I with probability at least $1/O(\log d)$ and therefore, one of those nodes joins I with probability at least $1 - 1/\text{poly}(d)$. ◀

We use the fact that nodes are added mostly independently and with low probability to V^* in order to show that the graph induced by these nodes cannot have many edges.

▶ **Lemma 6.** *The induced subgraph $G[V^*]$ has $O(n)$ edges w.h.p.*

Proof. A node v is placed in V^* if either (1) v is a good node with no neighbors in V_{samp}, or (2) v is an uncovered bad node in B_d and has a neighbor v that has at least $c \cdot \sqrt{d} \cdot \log^5(d)$ neighbors in B_d. By Lemmas 4 and 5, each such node v is put in V^* with probability at most $1/\text{poly}(\deg(v))$, and this happens independently of the randomness (for sampling into V_{samp} and for the first Luby round when computing I) of nodes at distance more than 3 from v. The exponent of the polynomial depends on c and γ.

Nodes with constant degree can be ignored, as they will contribute at most $O(n)$ edges. Therefore, we can assume that each node is put in V^* with probability at most $1/2$.

For the sake of analysis, we compute a greedy coloring of $G^7[V^*]$. To get $G^7[V^*]$, we first build the graph G^7 which is the graph where we add an edge between any pair of nodes that are at distance at most 7 in G, and then we take the induced subgraph of G^7 on V^*.

For each color class that has at least $n^{1-8\alpha}$ nodes, all of which are at distance more than 7 from each other in G, they join V^* independently of each other. Therefore, the probability that all nodes in a single color class belongs to V^* is at most $(1/2)^{n^{1-8\alpha}} \ll 1/\text{poly}(n)$. By union bounding over the color classes, we get that with probability $1 - 1/\text{poly}(n)$, the size of each color class is less than $n^{1-8\alpha}$.

Each node in $G^7[V^*]$ has degree at most $n^{7\alpha}$ and hence there are $n^{7\alpha} + 1$ color classes. Recall that $\Delta \le n^\alpha$ in G, so if a color class $C \subseteq V^*$ has less than $n^{1-8\alpha}$ nodes, the number of edges in G that are incident on C is at most $n^{1-7\alpha}$. Therefore, all color classes with less than $n^{1-8\alpha}$ nodes can only add at most $O(n)$ edges to $G[V^*]$.

The lemma follows since we already showed that w.h.p., the size of each color class is less than $n^{1-8\alpha}$. ◀

Recall that B_d is the set of bad nodes in G with (initial) degree in the range $[d, 2d)$. Let B_d^* be the nodes in B_d that have a neighbor v with more than $c\sqrt{d}\log^5 d$ neighbors in B_d. If a node in B_d^* is not covered by the MIS I on V_{samp}, then it is put into V^*. Let $\overline{B}_d = B_d \setminus B_d^*$. Define $B = \cup_{i=0}^{\log n} B_{2^i}$ and $\overline{B} = \cup_{i=0}^{\log n} \overline{B}_{2^i}$.

▶ **Lemma 7.** *The graph $G' = G[V \setminus (\mathcal{I} \cup N(\mathcal{I}) \cup N(N(\mathcal{I})))]$ of nodes that are uncovered before Line 6 contains only nodes in $\overline{B}$.*

Proof. Nodes not in $\overline{B}$ are the good nodes and the bad nodes in B_d^* for all d. We show that all these nodes are covered before Line 6 and hence cannot belong to G'. Nodes that are either in V_{samp}, or have a neighbor in V_{samp} are covered by the MIS I computed in Line 3. Good nodes that are not covered by I are put in V^* in Line 2. Similarly, nodes in B_d^* that are not covered by I are put in V^* in Line 4. All nodes in V^* are covered because we compute an MIS on $G[V^*]$ in Line 5. Therefore, all nodes not in $\overline{B}$ are covered before Line 6. ◀

2.2 Counting the Bad Nodes

Let $V_{\ge d}$ be the set of all nodes in G of (initial) degree at least d. Intuitively, we now want to say: (1) for each bad node in $\overline{B}_d$, there are at least $d/2$ edges to higher degree nodes and (2) from the higher degree nodes, only roughly $\sqrt{d}$ edges to $\overline{B}_d$. Hence, we can conclude that $d \cdot |\overline{B}_d| \le |V_{\ge d^2}| \cdot \sqrt{d}$ which further implies that $|\overline{B}_d| \le |V_{\ge d^2}|/\sqrt{d}$.

▶ **Lemma 8.** *Consider a bad node u with $\deg(u) = d$. Then for at least $d/2$ nodes $v \in N(u)$ it holds that $\deg(v) \ge d^2/(2\gamma^2 \log^2 d)$.*

Proof. Otherwise, more than half of the neighbors have degree less than $d^2/(2\gamma^2 \log^2 d)$. Hence,

$$\sum_{v \in N(u)} \frac{1}{\sqrt{\deg(v)}} \ge \frac{d}{2} \cdot \frac{\sqrt{2\gamma^2 \log^2 d}}{\sqrt{d^2}} = \frac{d}{2} \cdot \frac{2\gamma \log d}{d} = \gamma \log d \ ,$$

which is a contradiction with u being bad. ◀

▶ **Lemma 9.** *For any d, we have that $|\overline{B}_d| \le 2|V_{\ge d^2/(2\gamma^2 \log^2 d)}| \cdot \log^5 d/\sqrt{d} \le 2n \log^5 d/\sqrt{d}$.*

Proof. Let $d' = d^2/(2\gamma^2 \log^2 d)$. From Lemma 8, we know that for any $u \in \overline{B}_d$, at least $d/2$ edges go to $V_{\ge d'}$. Furthermore, we have that any $v \in V_{\ge d'}$ has at most $c\sqrt{d}\log^5 d$ edges to $\overline{B}_d$, since otherwise, none of v's neighbors are in $\overline{B}_d$. Hence, we can conclude that $\frac{d}{2} \cdot |\overline{B}_d| \le |V_{\ge d'}| \cdot \sqrt{d}\log^5 d$ which proves the lemma. ◀

Now we have that just before Line 6, $|\overline{B}_d| \approx n/\sqrt{d}$, and we now run the entire algorithm again on the uncovered graph G'. For G', we define the sets $V'_{\geq d}$, B'_d, $\overline{B'}_d$, B', and $\overline{B'}$ similarly as we did for G. Now we would expect that $|\overline{B'}_d| \approx n/d$, which is good because all nodes have degree at most $2d$.

▶ **Lemma 10.** *The graph induced by the uncovered nodes before Line 7 has $O(n)$ edges.*

Proof. Again let $d' = d^2/(2\gamma^2 \log^2 d)$. By a similar argument as Lemma 7, the uncovered nodes are a subset of $\overline{B'}$. We can assume that d is at least some large enough constant, as the nodes in all B'_d for constant d have at most $O(n)$ edges. By Lemma 9, we have that $|\overline{B}_d| \leq 2|V_{\geq d'}| \log^5 d/\sqrt{d}$ for any d, and we can similarly argue $|\overline{B'}_d| \leq 2|V'_{\geq d'}| \log^5 d/\sqrt{d}$ for any d.

Now $V'_{\geq d'} \subseteq \overline{B}_{\geq d'}$ where $\overline{B}_{\geq d'} = \cup_{i=\log d'}^{\log n} \overline{B}_{2^i}$. Therefore,

$$|V'_{\geq d'}| \leq \sum_{i=\log d'}^{\log n} \frac{2n \log^5 2^i}{\sqrt{2^i}} = \sum_{i=\log d'}^{\log n} \frac{2ni^5}{2^{i/2}} \leq \sum_{i=\log d'}^{\log n} \frac{2n}{2^{i/3}} \leq O\left(\frac{n \log^{2/3} d}{d^{2/3}}\right)$$

Where the last inequality follows because the sum on the left is a geometric sequence with rate $2^{-1/3}$ and it is well known that if the rate is between 0 and 1, the sum is asymptotically dominated by the first term. Therefore, $|\overline{B'}_d| \leq O(n \cdot \text{poly}(\log d)/d^{7/6}) \leq O(n/d)$. Since each node in $\overline{B'}_d$ has degree at most $2d$, and each node belongs to exactly one set, the graph induced by nodes in $\overline{B'}$ has $O(n)$ edges. ◀

2.3 Degree Reduction

We define the degree reduction process similar to [14] and for sake of completeness we provide a self-contained explanation here. Our goal is to lower the maximum degree of the input graph G such that, after a constant number i of steps, the maximum degree is strictly less than n^α for some fixed constant $\alpha < 1/8$. Let $G_1 = G$, and $\Delta_j = \Delta^{(3/4)^j}$. Let i be a value such that for all $1 \leq j < i$, $\Delta_j > n^{\alpha/2}$ and $\Delta_i \leq n^{\alpha/2}$. Since $\Delta \leq n$, we can say that the largest value i can take is $\lceil \log_{4/3}(2/\alpha) \rceil = O(1)$.

In each step $j = 1 \ldots i$, we sample nodes S_j in G_j with probability $1/\Delta_j$, and then compute an MIS on the subgraph induced by the sampled nodes $G_j[S_j]$. The residual graph G_{j+1} is obtained by removing all the neighbors of the sampled nodes. For each of these steps to be possible, the graph induced by the sampled nodes must have $O(n)$ edges w.h.p. In order to guarantee the feasibility of step j, the maximum degree in the residual graph after step $j-1$ (i.e. G_j) has to be sufficiently small, which we show in the following lemma.

▶ **Lemma 11.** *If we process a graph G_j induced by nodes picked uniformly at random with probability $1/\Delta_j$, the maximum degree in the residual graph G_{j+1} is $O(\Delta_j \log n)$ w.h.p.*

Proof. Consider a node v in the residual graph such that $\deg(v) > d$. The probability that a neighbor of v is sampled is at least $1/\Delta_j$. Therefore, the probability that no neighbor of v is sampled is at most $\left(1 - \frac{1}{\Delta_j}\right)^d \leq \exp(-d/\Delta_j)$.

Denote $c > 1$ an arbitrary constant, and suppose that $d = c\Delta_j \log n$. Then, the probability that $\deg(v) > d$ is at most $\exp(-c \log n) = n^{-c}$. Hence, $\deg(v) = O(\Delta_j \log n)$ w.h.p. We conclude the lemma by union bounding over all nodes of the residual graph. ◀

Therefore, the residual graph G_{i+1} has maximum degree $O(\Delta_i \log n) \leq O(n^{\alpha/2} \log n)$ $\leq n^\alpha$ w.h.p., which is our assumption in Algorithm 1. We now finish by showing that each induced subgraph has linear size.

▶ **Lemma 12.** *For all $1 \leq j \leq i$, the graph induced by sampled nodes in step j, $G_j[S_j]$, has $O(n)$ edges w.h.p.*

Proof. First, consider a node $v \in S_j$, and $u \in N(v)$. The probability that $u \in S_j$ is $1/\Delta_j$. By Lemma 11, we condition on the high probability event that $O(\Delta_{j-1} \log n)$ is the maximum degree of G_j. Note that this conditioning only affects the randomness used for sampling in iterations $1, \ldots, j-1$. In particular, this implies that the conditioning does not affect the randomness used for sampling in iteration j. Therefore, the expected degree of v in S_j is at most $\mu = O(\log n \cdot \Delta_{j-1}/\Delta_j)$. Using Lemma 16, $\Pr[\deg(v) \geq (1+c)\mu] \leq \exp(-c^2\mu/(2+c)) \leq n^{-c}$, since $\mu \geq \log n$. By union bounding over all nodes of S_j, the maximum degree in $G_j[S_j]$ is $O(\log n \cdot \Delta_{j-1}/\Delta_j)$ w.h.p.

Second, the expected number of nodes in S_j is at most $\mu' = n/\Delta_j$. Using Lemma 16, $\Pr[|S_j| \geq (1 + c\log n)\mu'] \leq \exp(-c^2 \log^2 n\mu'/(2 + c\log n)) \leq n^{-c}$, since $\mu' \geq 1$. Therefore, $|S_j| = O(n \log n/\Delta_j)$ w.h.p.

Now we can upper bound the number of edges in $G_j[S_j]$ by $|S_j| \cdot \Delta(G_j[S_j])$. Therefore, we can say w.h.p. that

$$|S_j| \cdot \Delta(G_j[S_j]) = O\left(\frac{n\log n}{\Delta_j} \cdot \frac{\Delta_{j-1}\log n}{\Delta_j}\right) = O\left(n\log^2 n \cdot \frac{\Delta_{j-1}}{\Delta_j^2}\right)$$

Moreover, since $\Delta_j = \Delta^{(3/4)^j}$, we have that $\Delta_{j-1}/\Delta_j^2 = \Delta^{(3/4)^{j-1}-2(3/4)^j} = \Delta^{-(1/2)(3/4)^{j-1}} = 1/\sqrt{\Delta_{j-1}}$. This term cancels the $\log^2 n$ term since $\Delta_{j-1} > n^{\alpha/2} \gg \log^4 n$. Hence, the number of edges in $G_j[S_j]$ is $O(n)$ w.h.p. ◀

3 Implementation of Parallel 2-Ruling-Set

In this section, we show how to implement the algorithm Parallel 2-Ruling-Set in several models of parallel computation. In each case of the implementations, we only need to bound the runtime and memory usage of the algorithm in the corresponding model. Since we faithfully execute the Parallel 2-Ruling-Set algorithm, the solutions computed are guaranteed to be correct.

3.1 Congested Clique

In the Congested Clique model [22], we have n machines, where each machine is identified with a single node in the input graph. The communication network is a clique, that is, the machines are connected in all-to-all fashion, and the input graph is considered to be a subgraph of the network. Machines can send unique messages to all other machines via the edges in the clique, and the bandwidth of each edge is limited to $O(\log n)$ bits. Congested Clique and MPC are closely related to each other, for example [17, 6] show how to implement any Congested Clique algorithm in the MPC model.

By the definition of Parallel 2-Ruling-Set, we compute an MIS sequentially on several subgraphs: (1) the sampled graphs $G_j[S_j]$ $(1 \leq j \leq i = O(1))$ for reducing the degree, (2) the graphs induced by V_{samp} and V^* on G during the first run and on G' during the second run, and finally (3) the graph induced by uncovered nodes in the last step. Creating a subgraph takes a constant number of rounds, since nodes just need to know the random choices and aggregate information of their 1-hop neighbors.

By the lemmas in Section 2.1 and Section 2.3, all these subgraphs have $O(n)$ edges w.h.p. Therefore, we can use Lenzen's routing protocol [21] to gather all the subgraphs one after the other at a single machine in $O(1)$ rounds. This machine computes the MIS according to Algorithm 1, and informs the rest of the nodes whether they joined the MIS or not, which allows us to identify the next subgraph. Therefore, we get the following result.

▶ **Theorem 13.** *There is a Congested Clique algorithm to find a* 2*-ruling set. The algorithm runs in* $O(1)$ *rounds w.h.p.*

3.2 Linear Memory MPC Model

In the MPC model [19, 15, 5], we have M machines with S words of memory each, where each word corresponds to $O(\log n)$ bits. Notice that an identifier of an edge or a node requires one word to store. The machines communicate in an all-to-all fashion. The input graph is divided among the machines and for simplicity and without loss of generality, we assume that the edges of each node are placed on the same machine. In the *linear-space* MPC model, we set $S = \Theta(n)$. Furthermore, the *total space* is defined as $M \cdot S$ and in our case, we have $M \cdot S = \Theta(m)$. Notice that $M \cdot S = \Omega(m)$, for the number of edges m in the input graph, simply to store the input.

The implementation follows as a direct consequence of the Congested Clique model. Since the local memory is $\Theta(n)$, we can send each of the $O(n)$ sized subgraphs one by one to a single machine in $O(1)$ rounds and use this machine to compute the MIS as described in the algorithm. We obtain the following theorem.

▶ **Theorem 14.** *There is an MPC algorithm with* $O(n)$ *words of local and* $O(m)$ *total memory to find a* 2*-ruling set. The algorithm runs in* $O(1)$ *rounds w.h.p.*

3.3 Semi-streaming

Typically, in the distributed and parallel settings, the input graph is too large to fit a single computer. Hence, it is divided among several computers (in one way or another) and the computers need to communicate with each other to solve a problem. Another angle at tackling large datasets and graphs is through the *graph streaming models* [10, 11, 26]. In these models, the graph is not stored centrally, but an algorithm has access to the edges one by one in an input stream, chosen randomly or by an adversary (the choice sometimes makes a difference). We assume that each edge is processed before the next pass starts. In the semi-streaming setting, the algorithm has $\widetilde{O}(n)$ working space, that it can use to store its state. The goal is to make as few *passes* over the edge-stream as possible, ideally just a small constant amount. Notice that in the case of many problems, such as matching approximation or correlation clustering, simply storing the output might demand $\Omega(n)$ words.

In the semi-streaming model, we use one pass to process one subgraph of size $O(n)$, by storing it in memory and computing an MIS as described in the algorithm. Since there are $O(1)$ such subgraphs, we require $O(1)$ passes. This leads to the following theorem.

▶ **Theorem 15.** *There is an* $O(1)$*-pass semi-streaming algorithm with* $O(n)$ *words of space to find a* 2*-ruling set w.h.p.*

4 Concentration Inequalities

▶ **Lemma 16** (Chernoff Bounds). *Let* $X_1, \dots, X_k$ *be independent* $\{0,1\}$ *random variables. Let* X *denote the sum of the random variables,* μ *the sum's expected value. Then,*

1. *For* $0 \le \delta \le 1$, $\Pr[X \le (1-\delta)\mu] \le \exp(-\delta^2\mu/2)$ *and* $\Pr[X \ge (1+\delta)\mu] \le \exp(-\delta^2\mu/3)$,
2. *For* $\delta \ge 1$, $\Pr[X \ge (1+\delta)\mu] \le \exp(-\delta^2\mu/(2+\delta))$.

▶ **Definition 17** (Bounded Differences Property). *A function* $f : \mathcal{X} \to \mathbb{R}$ *for* $\mathcal{X} = \mathcal{X}_1 \times \mathcal{X}_2 \cdots \times \mathcal{X}_n$ *is said to satisfy the bounded differences property with bounds* $c_1, c_2, \dots, c_n \in \mathbb{R}^+$ *if for all* $\overline{x} = (x_1, x_2, \dots, x_n) \in \mathcal{X}$ *and all integers* $k \in [1, n]$ *we have*

$$\sup_{x'_k \in X_k} |f(\overline{x}) - f(x_1, x_2, \dots, x_{i-1}, x'_k, \dots, x_n)| \le c_k$$

▶ **Lemma 18** (Bounded Differences Inequality [25, 24]). *Let $f : \mathcal{X} \to \mathbb{R}$ satisfy the bounded differences property with bounds $c_1, c_2, \ldots, c_n$. Consider independent random variables $X_1, X_2, \ldots, X_n$ where $X_k \in \mathcal{X}_k$ for all integers $k \in [1, n]$. Let $\overline{X} = (X_1, X_2, \ldots, X_n)$ and $\mu = \mathbb{E}[f(\overline{X})]$. Then for any $t > 0$ we have:*

$$\Pr[|f(\overline{X}) - \mu| \geq t] \leq 2 \exp\left(\frac{-t^2}{\sum_{k=1}^{n} c_k^2}\right)$$

References

1 Kook Jin Ahn, Graham Cormode, Sudipto Guha, Andrew McGregor, and Anthony Wirth. Correlation Clustering in Data Streams. In *International Conference on Machine Learning (ICML)*, pages 2237–2246, 2015.

2 Noga Alon, Lásló Babai, and Alon Itai. A Fast and Simple Randomized Parallel Algorithm for the Maximal Independent Set Problem. *Journal of Algorithms*, 7(4):567–583, 1986.

3 Sepehr Assadi and Aditi Dudeja. Ruling Sets in Random Order and Adversarial Streams. In Seth Gilbert, editor, *35th International Symposium on Distributed Computing (DISC 2021)*, volume 209 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:18, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.DISC.2021.6.

4 Sepehr Assadi, Gillat Kol, and Zhijun Zhang. Rounds vs communication tradeoffs for maximal independent sets. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1193–1204. IEEE, 2022.

5 Paul Beame, Paraschos Koutris, and Dan Suciu. Communication Steps for Parallel Query Processing. *J. ACM*, 64(6), 2017. doi:10.1145/3125644.

6 Soheil Behnezhad, Mahsa Derakhshan, and MohammadTaghi Hajiaghayi. Semi-mapreduce meets congested clique, 2018. doi:10.48550/ARXIV.1802.10297.

7 Andrew Berns, James Hegeman, and Sriram V. Pemmaraju. Super-Fast Distributed Algorithms for Metric Facility Location. In *Proceedings of the International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 428–439, 2012.

8 Graham Cormode, Jacques Dark, and Christian Konrad. Independent Sets in Vertex-Arrival Streams. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*, volume 132 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 45:1–45:14, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ICALP.2019.45.

9 Artur Czumaj, Peter Davies, and Merav Parter. Simple, deterministic, constant-round coloring in the congested clique. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, PODC '20, pages 309–318, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3382734.3405751.

10 Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. Graph Distances in the Streaming Model: The Value of Space. In *the Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 745–754, 2005.

11 Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. On Graph Problems in a Semi-Streaming Model. *Theor. Comput. Sci.*, 348(2):207–216, 2005. doi:10.1016/j.tcs.2005.09.013.

12 Mohsen Ghaffari. An improved distributed algorithm for maximal independent set. In *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 270–277. SIAM, 2016.

13 Mohsen Ghaffari. Distributed MIS via All-to-All Communication. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 141–149, 2017. doi:10.1145/3087801.3087830.

14 Mohsen Ghaffari, Themis Gouleakis, Christian Konrad, Slobodan Mitrović, and Ronitt Rubinfeld. Improved Massively Parallel Computation Algorithms for MIS, Matching, and Vertex Cover. In *the Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 129–138, 2018.

15 Michael T. Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, Searching, and Simulation in the Mapreduce Framework. In *Proceedings of the International Symposium on Algorithms and Computation (ISAAC)*, pages 374–383, 2011. doi:10.1007/978-3-642-25591-5_39.

16 James Hegeman, Sriram Pemmaraju, and Vivek Sardeshmukh. Near-Constant-Time Distributed Algorithms on a Congested Clique. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, pages 514–530, 2014. doi:10.1007/978-3-662-45174-8_35.

17 James W. Hegeman and Sriram V. Pemmaraju. Lessons from the congested clique applied to mapreduce. *Theoretical Computer Science*, 608:268–281, 2015. Structural Information and Communication Complexity. doi:10.1016/j.tcs.2015.09.029.

18 Tanmay Inamdar, Shreyas Pai, and Sriram V. Pemmaraju. Large-Scale Distributed Algorithms for Facility Location with Outliers. In Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira, editors, *22nd International Conference on Principles of Distributed Systems (OPODIS 2018)*, volume 125 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:16, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.OPODIS.2018.5.

19 Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A Model of Computation for MapReduce. In *the Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 938–948, 2010.

20 Christian Konrad, Sriram V Pemmaraju, Talal Riaz, and Peter Robinson. The complexity of symmetry breaking in massive graphs. In *33rd International Symposium on Distributed Computing (DISC 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

21 Christoph Lenzen. Optimal deterministic routing and sorting on the congested clique. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, pages 42–50. Association for Computing Machinery, 2013. doi:10.1145/2484239.2501983.

22 Zvi Lotker, Elan Pavlov, Boaz Patt-Shamir, and David Peleg. Mst construction in o(log log n) communication rounds. In *Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 94–100. Association for Computing Machinery, 2003. doi:10.1145/777412.777428.

23 M. Luby. A Simple Parallel Algorithm for the Maximal Independent Set Problem. *SIAM Journal on Computing*, 15:1036–1053, 1986.

24 Colin McDiarmid. *Concentration*, pages 195–248. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998. doi:10.1007/978-3-662-12788-9_6.

25 Colin McDiarmid et al. On the method of bounded differences. *Surveys in combinatorics*, 141(1):148–188, 1989.

26 S. Muthukrishnan. Data Streams: Algorithms and Applications. *Theoretical Computer Science*, 1(2):117–236, 2005. doi:10.1561/0400000002.

27 Krzysztof Nowicki. A deterministic algorithm for the mst problem in constant rounds of congested clique. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, STOC 2021, pages 1154–1165, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3406325.3451136.

28 Shreyas Pai and Sriram V. Pemmaraju. Brief Announcement: Deterministic Massively Parallel Algorithms for Ruling Sets. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 366–368, 2022. doi:10.1145/3519270.3538472.

Self-Stabilizing Clock Synchronization in Probabilistic Networks

Bernadette Charron-Bost
DI ENS, École Normale Supérieure, 75005 Paris, France

Louis Penet de Monterno
École polytechnique, IP Paris, 91128 Palaiseau, France

Abstract

We consider the fundamental problem of clock synchronization in a synchronous multi-agent system. Each agent holds a clock with an arbitrary initial value, and clocks must eventually indicate the same value, modulo some integer P. A known solution for this problem in dynamic networks is the self-stabilization *SAP* (for self-adaptive period) algorithm, which uses finite memory and relies solely on the assumption of a finite *dynamic diameter* in the communication network.

This paper extends the results on this algorithm to probabilistic communication networks: We introduce the concept of *strong connectivity with high probability* and we demonstrate that in any probabilistic communication network satisfying this hypothesis, the *SAP* algorithm synchronizes clocks with high probability. The proof of such a *probabilistic hyperproperty* is based on novel tools and relies on weak assumptions about the probabilistic communication network, making it applicable to a wide range of networks, including the classical PUSH model. We provide an upper bound on time and space complexity.

Building upon previous works by Feige et al. and Pittel, the paper provides solvability results and evaluates the stabilization time and space complexity of *SAP* in two specific cases of communication topologies.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Theory of computation → Dynamic graph algorithms

Keywords and phrases Self-stabilization, Clock synchronization, Probabilistic networks

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.12

Acknowledgements We would like to thank Patrick Lambein-Monette, Stephan Merz, and Guillaume Prémel for very useful discussions.

1 Introduction

There is a considerable interest in distributed systems consisting of multiple, potentially mobile, agents. This is mainly motivated by the emergence of large scale networks, characterized by the lack of centralized control, the access to limited information and a time-varying connectivity. Control and optimization algorithms deployed in such networks should be completely distributed, relying only on local observations and information, and robust against unexpected changes in topology such as link or node failures.

A canonical problem in distributed control is *clock synchronization*: In a system where agents are equipped with local discrete clocks with common pulses, the objective is that all clocks eventually synchronize despite arbitrary initializations. That corresponds to synchronization in *phase*, as opposed to the problem of synchronization in *frequency* (e.g. for instance in [34, 21, 23, 32]).

Clock synchronization is a fundamental problem arising in a number of applications, both in engineering and natural systems. A synchronized clock is a fundamental basic block used in many engineering systems, e.g. in the universal self-stabilizing algorithm developed by Boldi and Vigna [9], or for deploying distributed algorithms structured into synchronized *phases* (e.g., the *Two-Phase* and *Three-Phase Commit* algorithms [6], or many consensus algorithms [5, 20, 31, 14]). Clock synchronization also corresponds to a ubiquitous phenomenon in the natural world and finds numerous applications in physics and biology, e.g., the Kuramoto model for the synchronization of coupled oscillators [35], synchronous flashing fireflies, collective synchronization of pancreatic beta cells [29].

In the model we consider, we assume agents take steps in synchronous rounds, but do not have a consistent numbering of the rounds (e.g., agents regularly receive a common pulse). The communication pattern at each round is captured by a directed graph that may change continually from one round to the next, and the history of communications in the network is modeled as a whole by a *dynamic graph*, that is an infinite sequence of directed graphs with the same set of nodes. For a given set of agents (nodes), a communication model is thus naturally represented by a *subset* of dynamic graphs, and corresponds to a *probability distribution* on the set of dynamic graphs (or, equivalently, to a probability distribution on the communication graph at each round) in the probabilistic setting.

In deterministic communication models, Charron-Bost and Monterno [12] proposed a synchronization algorithm of periodic clocks, termed *SAP* (for self-adaptive period), which is finite state and self-stabilizing, i.e., the complete initial state of each agent (not just its clock value) can be arbitrary. This algorithm does not assume any global knowledge on the network, and tolerates time-varying topologies. It is proved to solve the mod P-synchronization problem in any dynamic network with a finite *dynamic diameter*, that is, from any time onward and for every pair of agents i and j, there is a temporal path of bounded length connecting i to j. In this paper, we focus instead on the probabilistic setting, and explore the behavior of the *SAP* algorithm in a general probabilistic network.

In a deterministic communication model, each execution of *SAP* is considered individually, and the *property* of mod-P synchronization must be satisfied by each possible execution in this model. By contrast, in the probabilistic framework, the set of executions of *SAP* is considered as a whole, and the correctness of *SAP* then corresponds to the following *hyperproperty* [16]: the probability of executions of *SAP* in which mod P-synchronization is achieved is greater than a chosen real $p \in [0, 1)$.

Unfortunately, verifying probabilistic hyperproperties is technical, and requires to develop novel proof strategies and new analysis tools. In particular, the notion of dynamic diameter, which is central in the correctness proof of *SAP* in [12], is no more relevant in a probabilistic framework. Indeed, we may see that in most of probabilistic networks, the dynamic diameter is almost surely infinite (cf. Section 2.4). Instead, we provide a suitable notion of *probabilistic diameter*, and define a probabilistic network to be *strongly connected with high probability.*

Importantly, *SAP* uses a finite but unbounded amount of memory, that is, in each execution, the memory usage of each node eventually stops growing. Regarding memory size issue, the *SAP* algorithm appears to be optimal since, in an unpublished companion paper [13], we prove that there is no self-stabilizing and bounded-memory algorithm solving the mod P-synchronization problem in a deterministic communication model if no bound on the dynamic diameter is known.

1.1 Related work

Self-stabilizing clocks have been extensively studied in different communication models and under different assumptions. In particular, clocks may be unbounded, in which case they are required to be eventually equal, instead of only congruent. The synchronization problem of unbounded clocks admits simple solutions in strongly connected networks, namely the *Min* and *Max* algorithms [22, 27].

Periodic clocks require more sophisticated synchronization mechanisms. In addition to strong connectivity and static networks, the pioneer papers on periodic clock synchronization [3, 28, 10, 1] all assume that a bound on the diameter is available. Then Boldi and Vigna [9] proposed a synchronization algorithm, based on a self-adaptive period mechanism, that dispenses with the latter assumption.

More recently, periodic clock synchronization has been studied in the *Beeping model* [17] in which agents have severely limited communication capabilities: given a static and connected bidirectional communication graph, in each round, each agent can either send a "beep" to all its neighbors or stay silent. A self-stabilizing algorithm has been proposed by Feldmann et al. [25], which is optimal both in time and space, but which requires to know a bound on the network size. As explained above, without this global information available at each node, such an algorithm does not exist: The best we can do with a self-stabilizing synchronization algorithm in a network of unknown dynamic diameter[1] is to use finite memory as achieved in *SAP*.

There are also numerous results for mod P-synchronization with faulty agents. The fault-tolerant solutions that have been proposed in various failure models, including the Byzantine failure model, using algorithmic schemes initially developed for consensus (e.g., see [18, 19]). They typically require a bidirectional connected (most of the time fully-connected) network.

All these works assume a *static* network. In [12], Charron-Bost et al. tackled the dynamic setting: they extended the method of self-adaptive periods developed in [9] to dynamic networks, and proposed the *SAP* algorithm for this type of networks with "dynamic disconnectivity".

For probabilistic communication models, the problem of clock synchronization has been addressed by Boczkowski et al. [8] and later on by Bastide et al. [4], both in the particular framework of the PULL model [30] over the fully-connected graph: In each round each agent receives a message from an agent sampled uniformly at random. Their focus is on minimizing message size and they both obtain a stabilization time of $O(\log n)$ in a network of size n. Unfortunately, the algorithms in these papers are specific to the PULL model, and their good performances highly rely on the assumption of a fully-connected network. Observe that, in the case where P is not a power of 2, those algorithms are not bounded-memory.

Clock synchronization has been studied in another probabilistic communication model, namely, the model of *population protocols*, consisting of a set of agents, interacting in randomly chosen pairs. This is basically an asynchronous model, where the synchronization task is quite different from the one studied in this paper since it amounts to implementing the abstraction of rounds [2]. In other words, the point in the population protocol model is to achieve synchronization in *frequency* instead of synchronization in *phase*.

1.2 Contribution

Our main contribution in this paper is the probabilistic correctness proof of the *SAP* algorithm: we show that it synchronizes periodic clocks in the large class of networks that are strongly connected with high probability. The probabilistic communication model is totally general in the sense that we only assume it to be *memoryless*, meaning that the communication graph at each round does not depend on the previous rounds. As a byproduct, our correctness proof provides upper bounds on the stabilization time and the space complexity of *SAP*.

[1] or unknown size in the static case.

Verifying probabilistic hyperproperties requires to develop novel proof strategies and analysis tools. In particular, we devise new parameters for probabilistic dynamic graphs, namely, a hierarchy of probabilistic diameters, and prove several basic properties on these diameters that are interesting on their own.

Finally, we apply our results for general probabilistic communication models to the classical PUSH model: Leveraging the fundamental properties of this rumor spreading model established by Feige et al. [24] and Pittel [33], we prove the probabilistic correctness of *SAP* in the PUSH model and provide an estimate of its stabilization time and its space complexity, first for a general bidirectional network, and then for the case of a fully-connected network.

2 Preliminaries

2.1 The computing model

We consider a networked system of n agents (nodes), denoted $1, 2, \ldots, n$. Computation proceeds in *synchronized rounds*, which are communication closed in the sense that no message received in round t is sent in a round different from t. In round $t\,(t = 1, 2, \cdots)$, each node successively (a) broadcasts a message at the beginning of round t, (b) receives some messages, and (c) undergoes an internal transition to a new state. Communications that occur at round t are modeled by a directed graph (digraph) $\mathbb{G}(t) = ([n], E_t)$ where $[n] = \{1, \ldots, n\}$. We assume a self-loop at each node in all these digraphs since a node can communicate with itself instantaneously. An infinite sequence of digraphs $\mathbb{G} = (\mathbb{G}(t))_{t\geqslant 1}$ is called a *dynamic graph* and the set of dynamic graphs of size n is denoted $\mathcal{G}_n$.

An *algorithm* $\mathcal{A}$ is given by a set $\mathcal{Q}$ of local states, a set of messages $\mathcal{M}$, a sending function $\sigma : \mathcal{Q} \to \mathcal{M}$, and a transition function $\tau : \mathcal{Q} \times \mathcal{M}^{\oplus} \to \mathcal{Q}$, where $\mathcal{M}^{\oplus}$ is the set of finite multisets over $\mathcal{M}$. Every state in $\mathcal{Q}$ is a possible initial state (self-stabilization model).

An *execution of* $\mathcal{A}$ with the dynamic graph $\mathbb{G}$ then proceeds as follows: In round $(t = 1, 2 \ldots)$, each node applies the sending function σ to its current state to generate the message to be broadcasted, then it receives the messages sent by its incoming neighbors in the digraph $\mathbb{G}(t)$, and finally applies the transition function τ to its current state and the list of messages it has just received to go to a next state. It is entirely determined by the collection of initial states and the dynamic graph $\mathbb{G}$.

Given an execution of $\mathcal{A}$, the value of any local variable x_i at the end of round t is denoted by $x_i(t)$, and $x_i(0)$ is the initial value of x_i.

2.2 Dynamic graphs and probability measure

Let us first recall that the *product* of two digraphs $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$, denoted $G_1 \circ G_2$, is the digraph with the set of nodes V and with an edge (i, j) if there exists $k \in V$ such that $(i, k) \in E_1$ and $(k, j) \in E_2$. For any dynamic graph $\mathbb{G}$ and any integers $t' \geqslant t \geqslant 1$, we let

$$\mathbb{G}(t : t') \overset{\text{def}}{=} \mathbb{G}(t) \circ \cdots \circ \mathbb{G}(t').$$

By convention, $\mathbb{G}(t : t) = \mathbb{G}(t)$, and when $0 < t' < t$, $\mathbb{G}(t : t')$ is the digraph with only a self-loop at each node. The set of i's in-neighbors in $\mathbb{G}(t : t')$ is denoted by $\mathrm{In}_i(t : t')$, and simply by $\mathrm{In}_i(t)$ when $t' = t$. Every edge (i, j) in $\mathbb{G}(t : t')$ corresponds to a *path in the round interval* $[t, t']$: there exist $t' - t + 2$ nodes $i = k_0, k_1, \ldots, k_{t'-t+1} = j$ such that (k_r, k_{r+1}) is an edge of $\mathbb{G}(t + r)$ for each $r = 0, \ldots, t' - t$.

For a fixed $n \in \mathbb{N}^+$, and a triple $i \in [n]$, $t \in \mathbb{N}$, $\delta \in \mathbb{N}^+$, we let

$$\Gamma_i^{t,\delta} \overset{\text{def}}{=} \{\mathbb{G} \in \mathcal{G}_n \mid \forall j \in [n], (i,j) \text{ is an edge of } \mathbb{G}(t+1 : t+\delta)\}.$$

If Σ_n denotes the Borel σ-algebra on $\mathcal{G}_n$, then $(\mathcal{G}_n, \Sigma_n)$ is a measurable space. Then, we consider a probability measure on $(\mathcal{G}_n, \Sigma_n)$, denoted $\Pr_n$, or simply Pr. The pair $(\mathcal{G}_n, \Pr_n)$ is called a *probabilistic communication network* of size n.

By analogy with the terminology used in game theory (e.g., see [7]), we say that Pr is *memoryless* if the random variables $\mathbb{G}(1), \mathbb{G}(2), \mathbb{G}(3), \dots$ are mutually independent.

Each execution of an algorithm $\mathcal{A}$ in a system with n nodes is characterized by an initial global state in $\mathcal{Q}^n$ and a communication graph in $\mathcal{G}_n$. Hence, the set of executions of $\mathcal{A}$ starting at $q \in \mathcal{Q}^n$, denoted $\mathcal{E}_q(\mathcal{A})$, is isomorphic to $\mathcal{G}_n$. Thus Pr induces a probabilistic measure on $\mathcal{E}_q(\mathcal{A})$, which will also be denoted by Pr as no confusion can arise.

2.3 The mod *P*-synchronization problem

Let P and n be two positive integers, and let $\mathcal{A}$ be an algorithm where each node i maintains an *integer* variable C_i, called the clock of i. An execution of $\mathcal{A}$, over a network of n agents, is said *to achieve mod P-synchronization in τ rounds* if

$$\exists c \in \mathbb{N}, \forall t \geqslant \tau, \forall i \in [n],\ C_i(t) \equiv_P t + c,$$

in which case the network is said to be *synchronized* (for *mod P-synchronized*) *from round τ*, i.e.,

$$\forall t \geqslant \tau, \forall i, j \in [n], \quad C_i(t) \equiv_P C_j(t).$$

Then we say that the algorithm $\mathcal{A}$ with n agents *solves the mod P-synchronization problem in τ rounds with probability* p if for every initial state $q \in \mathcal{Q}^n$ of $\mathcal{A}$, the measure of the set of executions of $\mathcal{A}$ achieving mod P-synchronization in τ rounds is at least equal to p.

2.4 Probabilistic diameters

Let us fix a global state $q \in \mathcal{Q}^n$. For any real $p \in [0,1]$ and any integer $k \in [n]$, we define the *probabilistic order k diameter* as the minimum number of rounds required for k arbitrary nodes to communicate with all the nodes in the network with probability at least p. Formally, we let

$$\hat{D}^{(k)}(p) \overset{\text{def}}{=} \inf\{\delta \in \mathbb{N}^+ \mid \inf_{i_1,\cdots,i_k \in [n], t \in \mathbb{N}} \Pr(\Gamma_{i_1}^{t,\delta} \cap \cdots \cap \Gamma_{i_k}^{t,\delta}) \geqslant p\}.$$

Clearly, we have that $\hat{D}^{(1)}(p) \leqslant \cdots \leqslant \hat{D}^{(n)}(p)$, with equalities when $p = 1$. As a matter of fact, the probabilistic proof of the SAP_g algorithm that we develop in the following section only involves the probabilistic diameters $\hat{D}^{(1)}(p)$ and $\hat{D}^{(2)}(p)$.

As an example, let us consider the memoryless probability measure Pr on $\mathcal{G}_2$ defined by

$$\Pr(\mathbb{G}(t) = G_1) = \Pr(\mathbb{G}(t) = G_2) = \frac{1}{2},$$

where G_1 and G_2 are the two-node digraphs defined in Figure 1. For any round t and any positive integer δ, we have:

$$\Pr(\Gamma_1^{t,\delta}) = 1 - \Pr\left(\bigcap_{d=1}^{\delta} (\mathbb{G}(t+d) = G_2)\right) = 1 - \prod_{\ell=1}^{\delta} \Pr(\mathbb{G}(t+d) = G_2) = 1 - 2^{-\delta}.$$

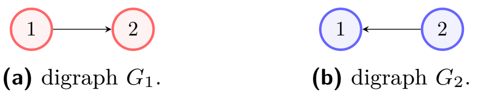


Figure 1 Two digraphs.

Similarly, $\Pr(\Gamma_2^{t,\delta}) = 1 - 2^{-\delta}$. Moreover,

$$\begin{aligned}\Pr(\Gamma_1^{t,\delta} \cap \Gamma_2^{t,\delta}) &= 1 - \Pr(\overline{\Gamma_1^{t,\delta}} \cup \overline{\Gamma_2^{t,\delta}}) \\ &= 1 - \Pr\left(\bigcap_{d=1}^{\delta} (\mathbb{G}(t+d) = G_2) \cup \bigcap_{d=1}^{\delta} (\mathbb{G}(t+d) = G_1)\right) \\ &= 1 - 2^{-\delta+1}.\end{aligned}$$

Using the definition of the probabilistic diameters and the two equations above, we obtain the values of $\hat{D}^{(1)}(p)$ and $\hat{D}^{(2)}(p)$ in our example:

$$\hat{D}^{(1)}(p) = \inf\{\delta \in \mathbb{N}^+ \mid 1 - 2^{-\delta} \geqslant p\} = \lceil -\log_2(1-p) \rceil \text{ and}$$
$$\hat{D}^{(2)}(p) = \inf\{\delta \in \mathbb{N}^+ \mid 1 - 2^{-\delta+1} \geqslant p\} = 1 + \lceil -\log_2(1-p) \rceil.$$

This simple example shows why it is not appropriate to use the parameter $D(p)$ simply defined by:

$$D(p) \overset{\text{def}}{=} \inf\{\delta \in \mathbb{N}^+ \mid \Pr(D_{\mathbb{G}} \leqslant \delta) \geqslant p\},$$

where $D_{\mathbb{G}} = \inf\{\delta \in \mathbb{N}^+ \mid \forall i \in [n], \forall t \in \mathbb{N}, \mathbb{G} \in \Gamma_i^{t,\delta}\}$ is the *dynamic diameter* of the dynamic graph $\mathbb{G}$ [11]. Indeed, for each node $i \in \{1,2\}$, we have:

$$\Pr(D_{\mathbb{G}} \leqslant \delta) \leqslant \Pr\left(\bigcap_{\ell=0}^{\infty} \Gamma_i^{\ell\delta,\delta}\right) = \prod_{\ell=0}^{\infty} \Pr(\Gamma_i^{\ell\delta,\delta}) = 0,$$

and thus $D(p)$ is infinite if p is positive. In other words, the dynamic diameter of almost all dynamic graphs is infinite in this example, while $\hat{D}^{(1)}(p)$ is finite when $p < 1$.

We now state some general properties on the probabilistic diameters $\hat{D}^{(1)}(p)$ and $\hat{D}^{(2)}(p)$.

▶ **Lemma 1.** *For any memoryless probability measure and all real numbers $p \in \left[\frac{1}{2}, 1\right]$, if $\hat{D}^{(1)}(p)$ is finite, then $\hat{D}^{(2)}(p)$ is finite and $\hat{D}^{(2)}(p) \leqslant 2\hat{D}^{(1)}(p)$.*

Proof. Because of the self-loops, the digraphs $\mathbb{G}(t+1 : t+\delta)$ and $\mathbb{G}(t+\delta+1 : t+2\delta)$ are both subgraphs of $\mathbb{G}(t+1 : t+2\delta)$, and hence $\Gamma_i^{t,\delta} \cup \Gamma_i^{t+\delta,\delta} \subseteq \Gamma_i^{t,2\delta}$. It follows that:

$$\begin{aligned}\Pr\left(\Gamma_i^{t,2\hat{D}^{(1)}(p)} \cap \Gamma_j^{t,2\hat{D}^{(1)}(p)}\right) &\geqslant 1 - \Pr\left(\overline{\Gamma_i^{t,2\hat{D}^{(1)}(p)}}\right) - \Pr\left(\overline{\Gamma_j^{t,2\hat{D}^{(1)}(p)}}\right) \\ &\geqslant 1 - \Pr\left(\overline{\Gamma_i^{t,\hat{D}^{(1)}(p)}} \cap \overline{\Gamma_i^{t+\hat{D}^{(1)}(p),\hat{D}^{(1)}(p)}}\right) \\ &\quad - \Pr\left(\overline{\Gamma_j^{t,\hat{D}^{(1)}(p)}} \cap \overline{\Gamma_j^{t+\hat{D}^{(1)}(p),\hat{D}^{(1)}(p)}}\right) \\ &\geqslant 1 - 2(1-p)^2.\end{aligned}$$

The second inequality holds because of the above-proved inclusion, and the third one because of the memoryless assumption. If $p \in \left[\frac{1}{2}, 1\right]$, then $1 - 2(1-p)^2 \geqslant p$ and $\hat{D}^{(2)}(p) \leqslant 2\hat{D}^{(1)}(p)$. ◀

Using a similar proof, it is possible to show that, for all $p \in [\frac{1}{2}, 1]$, for all ℓ_1 and $\ell_2 \leqslant \ell_1$ such that $\ell_1 + \ell_2 \leqslant n$, if $\hat{D}^{(\ell_1)}(p)$ and $\hat{D}^{(\ell_2)}(p)$ are finite, then $\hat{D}^{(\ell_1+\ell_2)}(p)$ is finite and $\hat{D}^{(\ell_1+\ell_2)}(p) \leqslant 2\hat{D}^{(\ell_1)}(p)$. Therefore, by induction on $\ell \leqslant n$, if $\hat{D}^{(1)}(p)$ is finite, then all $\hat{D}^{(\ell)}(p)$ are finite and $\hat{D}^{(\ell)}(p) \leqslant 2^{\lceil \log_2 \ell \rceil} \hat{D}^{(1)}(p)$. Finally, we prove the following finiteness result for $\hat{D}^{(1)}(p)$.

▶ **Lemma 2.** *For any memoryless probability measure, if $\hat{D}^{(1)}(p_0)$ is finite for some $p_0 \in (0, 1]$, then $\hat{D}^{(1)}(p)$ is finite for all real numbers $p \in [0, 1)$.*

Proof. For every node i, for every integers $t \geqslant 0$ and $\ell > 0$, we have

$$\begin{aligned}
\Pr\left(\Gamma_i^{t,\ell\hat{D}^{(1)}(p_0)}\right) &\geqslant \Pr\left(\bigcup_{h=0}^{\ell-1} \Gamma_i^{t+h\hat{D}^{(1)}(p_0),\hat{D}^{(1)}(p_0)}\right) \\
&= 1 - \prod_{h=0}^{\ell-1} \Pr\left(\overline{\Gamma_i^{t+h\hat{D}^{(1)}(p_0),\hat{D}^{(1)}(p_0)}}\right) \\
&\geqslant 1 - (1-p_0)^{\ell}.
\end{aligned}$$

The first inequality holds because of the self-loops, as explained in the proof of Lemma 1. The second one is due to the memoryless assumption on the Pr probability measure.

If p_0 is positive, then $\lim_{\ell\to\infty} 1 - (1-p_0)^{\ell} = 1$. Thus, for any real number p less than one, there exists some integer ℓ_0 such that $\Pr(\Gamma_i^{t,\ell_0\hat{D}^{(1)}(p_0)}) \geqslant p$, which implies that $\hat{D}^{(1)}(p)$ is finite and $\hat{D}^{(1)}(p) \leqslant \ell_0 \hat{D}^{(1)}(p)$. ◀

Since all $\hat{D}^{(\ell)}$ are non-decreasing, Lemmas 1 and 2 imply that if $\hat{D}^{(1)}(p_0)$ is finite for some $p_0 \in (0, 1]$, then all probabilistic diameters are finite for all $p \in [0, 1)$, in which case the network $(\mathcal{G}_n, \Pr_n)$ is said to be *strongly connected with high probability.* This notion is linked to the notion of dynamic diameter. Assume that a dynamic graph $\mathbb{G}$ has a finite dynamic diameter $D_{\mathbb{G}}$. Let Pr be the probability measure such that $\Pr(\{\mathbb{G}\}) = 1$, then for all real numbers $p \in (0, 1]$,

$$D_{\mathbb{G}} = \hat{D}^{(1)}(p) = \hat{D}^{(2)}(p) = \cdots = \hat{D}^{(n)}(p).$$

3 The *SAP* algorithm

We present the self-stabilizing *SAP* algorithm designed in [12] for the mod P-synchronization problem in dynamic networks with a finite dynamic diameter, and recall its basic properties.

3.1 Description of the algorithm

A typical approach to solve the mod P-synchronization problem consists in the following algorithm: at each round, each node sends its own variable $C_i \in \{0, \dots, P-1\}$ and applies the following update rule:

$$C_i \leftarrow \left[\min_{j \in \mathrm{In}_i} C_j + 1\right]_P,$$

where In_i denotes the current set of i's incoming neighbors, and $[c]_P$ is the remainder of the Euclidean division of c by P. Unfortunately, this naive algorithm does not work[2] when $\hat{D}^{(1)}(p)$ is too large compared to the period P. To overcome this problem, the *SAP* algorithm uses self-adaptive periods and the basic fact that for any positive integer M, we have

[2] see Theorem 4.13 in [1].

$$[\,[c]_{PM}\,]_P = [c]_P.$$

More precisely, each node i uses two *integer* variables M_i and C_i, and computes the clock value C_i not modulo P, but rather modulo the time-varying period PM_i. The variable M_i is used as a guess to find a large enough multiple of P so to make the clocks eventually stabilized. Until synchronization, the variables M_i increase so that the shortest period PM_i eventually becomes large enough compared to the largest clock value in the network. In the rest of this paper, $\mathcal{S}_t$ denotes the set of executions in which the system is synchronized in round t. Once all clocks are congruent modulo P, they remain congruent forever, meaning that $\mathcal{S}_t \subseteq \mathcal{S}_{t+1}$. The update rule for M_i is parametrized by a function $g : \mathbb{N} \to \mathbb{N}$. The corresponding algorithm is denoted SAP_g, and its code is given below. Line 5 in the pseudo-code implies that $C_i(t) < PM_i(t)$, and for the sake of simplicity, we assume that this inequality also holds initially, that is, $C_i(0) < PM_i(0)$.

Algorithm 1 Pseudo-code of node i in the SAP_g algorithm.

```
Variables:
 1: $C_i \in \mathbb{N}$;
 2: $M_i \in \mathbb{N}^+$;
In each round do:
 3: send $\langle C_i, M_i \rangle$ to all
 4: receive $\langle C_{j1}, M_{j1} \rangle, \langle C_{j2}, M_{j2} \rangle, \ldots$ from the set $\mathrm{In}_i$ of incoming neighbours
 5: $C_i \leftarrow \left[\min_{j \in \mathrm{In}_i} C_j + 1\right]_{PM_i}$
 6: $M_i \leftarrow \max_{j \in \mathrm{In}_i} M_j$
 7: if $C_j \not\equiv_P C_{j'}$ for some $j, j' \in \mathrm{In}_i$ then
 8:     $M_i \leftarrow g(M_i)$
 9: end if
```

In the rest of the paper, the function g is supposed to be a non-decreasing and *inflationary* function, i.e., $x < g(x)$ for every positive integer x. Therefore, each M_i variable is non-decreasing. If ℓ is a positive integer, g^ℓ denotes the ℓ-th iterate of g. For every positive real number x, we let

$$g^*(x) \overset{\text{def}}{=} \inf\{\ell \in \mathbb{N}^+ \mid g^\ell(1) \geqslant x\}.$$

Since g is inflationary, $g^*(x)$ is finite for all integers x, and $g^*(x) \leqslant x$. The algorithm is parametrized by such a function g, and the corresponding algorithm will be denoted SAP_g.

3.2 Properties of *SAP*'s executions

Let us consider an execution ϵ of the SAP_g algorithm over a network of size n, with the dynamic graph $\mathbb{G}$. We start with three basic properties of ϵ which directly come from the pseudo-code.

▶ **Lemma 3.** *If (i, j) is an arc in $\mathbb{G}(s : t)$, then $C_j(t) \leqslant C_i(s-1) + t - s + 1$.*

▶ **Lemma 4.** *If (i, j) is an arc in $\mathbb{G}(s : t)$, then one of the following statements is true:*
1. $C_j(t) \equiv_P C_i(s-1) + t - s + 1$;
2. $M_j(t) \geqslant g(M_i(s-1))$.

▶ **Lemma 5.** *Let d be a positive integer. If $C_i(t) + d \leqslant PM_i(t)$ holds for all nodes i, then all the clocks C_i are greater than 0 in the round interval $[t+1, t+d-1]$.*

For the probabilistic correctness proof of SAP_g, we will use another property of its executions, stated in the lemma below, which is a refinement of an analogous property established in the deterministic case under the condition of a finite dynamic diameter [12]. The proof is given in the Appendix.

▶ **Lemma 6.** *Let d be any positive integer, and k be a node such that $C_k(t) = \min_{j\in[n]} C_j(t)$. If the execution ϵ belongs to $\Gamma_k^{t,d}$ and all the clocks C_i are greater than 0 in the round interval $[t+1, t+d-1]$, then the network is synchronized in round $t+d$.*

4 Probabilistic correctness of *SAP*

Our approach for the correctness proof of the SAP_g algorithm relies on a fundamental probabilistic hyperproperty relating the adaptive mechanism for the periods in SAP_g to the order one probabilistic diameter of the network.

We fix some integer n, some real $p \in (0,1)$ and some initial state $q \in \mathcal{Q}^n$ of SAP_g. We consider a memoryless probability measure Pr on $(\mathcal{G}_n, \Sigma_n)$, and so on the set $\mathcal{E}_q$ of SAP_g's executions starting in q. We assume that the probabilistic network $(\mathcal{G}_n, \mathrm{Pr}_n)$ is strongly connected w.h.p., and we let

$$t_0 \overset{\text{def}}{=} \hat{D}^{(2)}(p) \left\lceil \frac{\log\left((1-p)^{-1}\right)}{p} \left(\sqrt{g^*\left(\frac{2\hat{D}^{(1)}(p)}{P}\right)} + \sqrt{2}\right)^2 \right\rceil \tag{1}$$

which is finite since g is inflationary. For all positive integers t, we consider the random variable $M(t) \overset{\text{def}}{=} \min_{i\in[n]} M_i(t)$.

▶ **Lemma 7.** *For every real number $p \in (0,1)$, we have*

$$\mathrm{Pr}\left(\left(M(t_0) \geqslant \frac{2\hat{D}^{(1)}(p)}{P}\right) \cup \mathcal{S}_{t_0}\right) \geqslant p. \tag{2}$$

Proof. For ease of notation, we let $\bar{g} = g^*\left(\frac{2\hat{D}^{(1)}(p)}{P}\right)$ and $\ell_0 = t_0/\hat{D}^{(2)}(p)$. In the first part of the proof, we construct a family of independent random variables B_t that all follow a Bernoulli distribution. In each execution in $\mathcal{E}_q$ that is not synchronized at round t, there exist two nodes $i_1(t)$ and $i_2(t)$ such that

$$C_{i_1(t)}(t) \not\equiv_P C_{i_2(t)}(t). \tag{3}$$

Then i_1 and i_2 can be viewed as two random variables that map any execution of SAP_g to a sequence of type $\mathbb{N} \to [n]$. Let B_t be the random variable equal to 1 on $\Gamma_{i_1(t)}^{t,\hat{D}^{(2)}(p)} \cap \Gamma_{i_2(t)}^{t,\hat{D}^{(2)}(p)}$ and equal to 0, otherwise. By definition of $\hat{D}^{(2)}(p)$, each B_t follows a Bernoulli distribution whose parameter $\mathrm{Pr}(B_t = 1)$ is greater than or equal to p. Since Pr is memoryless, the random variable

$$B \overset{\text{def}}{=} \sum_{\ell=0}^{\ell_0-1} B_{\ell\hat{D}^{(2)}(p)}$$

is a sum of independent Bernoulli variables.

We now show that in all executions in $\mathcal{E}_q$ that are not synchronized in round $\ell_0\hat{D}^{(2)}(p)$, it holds that

$$M(\ell_0\hat{D}^{(2)}(p)) \geqslant g^B(1). \tag{4}$$

For that, we fix such an execution and prove by induction on ℓ_0 that Eq. (4) holds in this execution.

1. Base case: $\ell_0 = 0$. Then we have $B = 0$, and so $M(\ell_0 \hat{D}^{(2)}(p)) \geqslant 1 = g^B(1)$ as needed.
2. Inductive case: Assume that

$$M(\ell_0 \hat{D}^{(2)}(p)) \geqslant g^{\sum_{t=0}^{\ell_0-1} B_{t\hat{D}^{(2)}(p)}}(1)$$

holds for for some $\ell_0 \in \mathbb{N}$ and that the system is not synchronized in round $(\ell_0+1)\hat{D}^{(2)}(p)$. Then, the nodes $i_1 = i_1(\ell_0 \hat{D}^{(2)}(p))$ and $i_2 = i_2(\ell_0 \hat{D}^{(2)}(p))$ satisfy Eq. (3). Therefore, for every node i, there exists some $x \in \{1, 2\}$ such that

$$C_i((\ell_0+1)\hat{D}^{(2)}(p)) \not\equiv_P C_{i_x}(\ell_0 \hat{D}^{(2)}(p)) + \hat{D}^{(2)}(p). \tag{5}$$

If $B_{\ell_0 \hat{D}^{(2)}(p)} = 0$, then the inductive case immediately follows. Otherwise, $B_{\ell_0 \hat{D}^{(2)}(p)} = 1$, and so the digraph $\mathbb{G}(\ell_0 \hat{D}^{(2)}(p) + 1 : (\ell_0+1)\hat{D}^{(2)}(p))$ contains all the arcs of the form (i_1, i) and (i_2, i). Then for every node i, it holds that

$$M_i((\ell_0+1)\hat{D}^{(2)}(p)) \geqslant g(M_{i_x}(\ell_0 \hat{D}^{(2)}(p))) \geqslant g(M(\ell_0 \hat{D}^{(2)}(p))). \tag{6}$$

The first inequality holds by Lemma 4 and Eq. (5), and the second one because g is non-decreasing. Using the induction hypothesis, we get

$$M((\ell_0+1)\hat{D}^{(2)}(p)) \geqslant g\left(g^{\sum_{t=0}^{\ell_0-1} B_{t\hat{D}^{(2)}(p)}}(1)\right) = g^{\sum_{t=0}^{\ell_0} B_{t\hat{D}^{(2)}(p)}}(1).$$

We now let $x_0 = \frac{1}{p}\Big(\bar{g} + \log(1-p)^{-1} + \sqrt{2\bar{g}\log(1-p)^{-1} + \log^2(1-p)^{-1}}\Big)$, and easily check that

$$\ell_0 \geqslant \frac{1}{p}(\sqrt{\bar{g}} + \sqrt{2\log(1-p)^{-1}})^2 \geqslant \frac{1}{p}(\bar{g} + 2\log(1-p)^{-1} + 2\sqrt{2\bar{g}\log(1-p)^{-1}}) \geqslant x_0 > 0. \tag{7}$$

Moreover, x_0 satisfies

$$-\frac{x_0 p}{2}\left(1 - \frac{\bar{g}}{x_0 p}\right)^2 = \log(1-p). \tag{8}$$

We obtain

$$\begin{aligned}
\Pr\left(\left(M(t_0) \geqslant \frac{2\hat{D}^{(1)}(p)}{P}\right) \cup \mathcal{S}_{t_0}\right) &\geqslant \Pr\left(B \geqslant g^*\Big(\frac{2\hat{D}^{(1)}(p)}{P}\Big)\right)\\
&\geqslant 1 - \Pr\left(B \leqslant \frac{\bar{g}}{x_0 p}\,\mathbb{E}(B)\right)\\
&\geqslant 1 - e^{-\frac{\mathbb{E}(B)}{2}\left(1-\frac{\bar{g}}{x_0 p}\right)^2}\\
&\geqslant 1 - e^{-\frac{x_0 p}{2}\left(1-\frac{\bar{g}}{x_0 p}\right)^2}\\
&= p.
\end{aligned}$$

The first inequalities comes from Eq. (4). The second and the fourth inequalities hold because, by definition of B and Eq. (7), we have $\mathbb{E}(B) \geqslant \ell_0 p \geqslant x_0 p$. The third inequality is a Chernoff bound [15] applied to B, which is a sum of independent Bernoulli variables. The last equality is by Eq. (8). ◀

Combined with the basic properties of the SAP_g's executions stated in the previous section, Lemma 7 allows us to show our main result:

▶ **Theorem 8.** *If g is a non-decreasing and inflationary function, then the SAP_g algorithm solves the mod P-synchronization problem in any probabilistic network that is strongly connected w.h.p. More precisely, for all $p \in (0,1)$, nodes synchronize within*

$$\hat{D}^{(2)}(p) \left\lceil \frac{\log\left((1-p)^{-1}\right)}{p} \left(\sqrt{g^*\left(\frac{2\hat{D}^{(1)}(p)}{P}\right)} + \sqrt{2} \right)^2 \right\rceil + 3\hat{D}^{(1)}(p)$$

rounds with probability p^4, if $\hat{D}^{(1)}(p)$ and $\hat{D}^{(2)}(p)$ denote the order one and two probabilistic diameters of the network.

Proof. For ease of notation, we let $\hat{D}^{(1)} = \hat{D}^{(1)}(p)$. We first define four random variables:

1. Let i_0 be any node satisfying $C_{i_0}(t_0) = \min_{i\in[n]} C_i(t_0)$.
2. Let t_1 be the smallest integer greater than or equal to t_0, such that at least one node holds a clock equal to 0 in round t_1 if it exists, or is equal to infinity otherwise.
3. If t_1 is finite, let i_1 be any node such that $C_{i_1}(t_1) = 0$. Otherwise, let i_1 be an arbitrary node.
4. If t_1 is finite, let i_2 be any node satisfying $C_{i_2}(t_1 + \hat{D}^{(1)}) = \min_{i\in[n]} C_i(t_1 + \hat{D}^{(1)})$. Otherwise, let i_2 be an arbitrary node.

Then we define the events E and E' as

$$E \overset{\text{def}}{=} \{\epsilon \in \mathcal{G}_n \mid t_1 < t_0 + \hat{D}^{(1)}\} \quad \text{and} \quad E' \overset{\text{def}}{=} \{\epsilon \in \mathcal{G}_n \mid M(t_0) \geqslant \tfrac{2\hat{D}^{(1)}}{P}\}.$$

By Lemma 6, we have $\Gamma_{i_0}^{t_0,\hat{D}^{(1)}} \cap \overline{E} \subseteq \mathcal{S}_{t_0+\hat{D}^{(1)}}$. Since $\mathcal{S}_{t_0+\hat{D}^{(1)}} \subseteq \mathcal{S}_{t_0+3\hat{D}^{(1)}}$, it follows that

$$\Pr\left(\mathcal{S}_{t_0+3\hat{D}^{(1)}} \mid \Gamma_{i_0}^{t_0,\hat{D}^{(1)}} \cap \overline{E} \cap (E' \cup \mathcal{S}_{t_0})\right) = 1. \tag{9}$$

In any execution belonging to E, t_1 is finite and $C_{i_1}(t_1) = 0$. Therefore, in any execution in $E' \cap E \cap \Gamma_{i_1}^{t_1,\hat{D}^{(1)}}$, every variable M_i satisfies

$$PM_i(t_1+\hat{D}^{(1)}) \geqslant PM(t_1+\hat{D}^{(1)}) \geqslant PM(t_0) \geqslant 2\hat{D}^{(1)} = C_{i_1}(t_1)+2\hat{D}^{(1)} \geqslant C_i(t_1+\hat{D}^{(1)})+\hat{D}^{(1)}.$$

The first and third inequalities above are by definition of M and E', respectively. The second one holds because M is non-decreasing, and the last one comes from Lemma 3 and the fact that the execution is in $\Gamma_{i_1}^{t_1,\hat{D}^{(1)}}$. Lemma 5 then applies, and Lemma 6 shows that

$$E' \cap E \cap \Gamma_{i_1}^{t_1,\hat{D}^{(1)}} \cap \Gamma_{i_2}^{t_1+\hat{D}^{(1)},\hat{D}^{(1)}} \subseteq \mathcal{S}_{t_1+2\hat{D}^{(1)}}.$$

Since the random variable t_1 is greater than t_0, we get $\mathcal{S}_{t_0} \subseteq \mathcal{S}_{t_1+2\hat{D}^{(1)}}$, and so

$$\Pr\left(\mathcal{S}_{t_1+2\hat{D}^{(1)}} \mid (E' \cup \mathcal{S}_{t_0}) \cap E \cap \Gamma_{i_1}^{t_1,\hat{D}^{(1)}} \cap \Gamma_{i_2}^{t_1+\hat{D}^{(1)},2\hat{D}^{(1)}} \cap \Gamma_{i_0}^{t_0,\hat{D}^{(1)}}\right) = 1. \tag{10}$$

We are now in position to bound the probability $\Pr(\mathcal{S}_{t_0+3\hat{D}^{(1)}})$ from below. For the sake of readability, the conditional probability given the event $(E' \cup \mathcal{S}_{t_0}) \cap \Gamma_{i_0}^{t_0,\hat{D}^{(1)}}$ is now denoted $\Pr'$. Then we have

$$\begin{aligned}
\Pr(\mathcal{S}_{t_0+3\hat{D}^{(1)}}) &\geqslant \Pr(\mathcal{S}_{t_0+3\hat{D}^{(1)}} \cap \Gamma_{i_0}^{t_0,\hat{D}^{(1)}} \cap (E' \cup \mathcal{S}_{t_0})) \\
&= \Pr'(\mathcal{S}_{t_0+3\hat{D}^{(1)}}) \times \Pr(\Gamma_{i_0}^{t_0,\hat{D}^{(1)}} \mid E' \cup \mathcal{S}_{t_0}) \times \Pr(E' \cup \mathcal{S}_{t_0}) \\
&\geqslant p^2 \Pr'(\mathcal{S}_{t_0+3\hat{D}^{(1)}}) \\
&= p^2 \Pr'(\mathcal{S}_{t_0+3\hat{D}^{(1)}} \mid E) \Pr'(E) + p^2 \Pr'(\mathcal{S}_{t_0+3\hat{D}^{(1)}} \mid \overline{E}) \Pr'(\overline{E}) \\
&\geqslant p^2 \Pr'(\mathcal{S}_{t_1+2\hat{D}^{(1)}} \mid E) \Pr'(E) + p^2 \Pr'(\overline{E}) \\
&\geqslant p^2 \Pr'(\mathcal{S}_{t_1+2\hat{D}^{(1)}} \cap \Gamma_{i_1}^{t_1,\hat{D}^{(1)}} \cap \Gamma_{i_2}^{t_1+\hat{D}^{(1)},\hat{D}^{(1)}} \mid E) \Pr'(E) + p^2 \Pr'(\overline{E}) \\
&\geqslant p^2 \Pr'(\mathcal{S}_{t_1+2\hat{D}^{(1)}} \mid E \cap \Gamma_{i_1}^{t_1,\hat{D}^{(1)}} \cap \Gamma_{i_2}^{t_1+\hat{D}^{(1)},\hat{D}^{(1)}}) \\
&\quad \times \Pr'(\Gamma_{i_2}^{t_1+\hat{D}^{(1)},\hat{D}^{(1)}} \mid E \cap \Gamma_{i_1}^{t_1,\hat{D}^{(1)}}) \\
&\quad \times \Pr'(\Gamma_{i_1}^{t_1,\hat{D}^{(1)}} \mid E) \Pr'(E) + p^2 \Pr'(\overline{E}) \\
&\geqslant p^4 \Pr'(E) + p^2 \Pr'(\overline{E}) \\
&\geqslant p^4
\end{aligned}$$

Lemma 7 and the fact that Pr is memoryless are used in the second inequality. The third inequality is based on Eq. (9) and the fact that, in any execution in E, it holds that $\mathcal{S}_{t_1+2\hat{D}^{(1)}} \subseteq \mathcal{S}_{t_0+3\hat{D}^{(1)}}$. Sixth inequality relies on Eq. (10) and the fact that Pr is memoryless. ◀

As in the deterministic analysis of SAP_g, the above bound on its stabilization time provides an upper bound on its space complexity, namely each node uses at most

$$\log_2 P + 2\log_2 \left(g^{t_0+3\hat{D}^{(1)}(p)}(\overline{M}_0)\right)$$

bits with probability p^4, if t_0 is defined by Eq. (1) and $\overline{M}_0 = \max_{i\in[n]} M_i(0)$. The time bound and the space bound thus depend respectively on the functions g^* and g, leading to a time-space trade-off for choosing g: the faster g grows, the lower the synchronization time is, and the higher its space complexity is.

5 The *SAP* algorithm in the Application to push-based models

The previous section demonstrates that the notion of probabilistic diameter is a powerful tool for the probabilistic analysis of distributed algorithms. This section applies our general result in Theorem 8 to a probablistic communication model, called the PUSH model [26]. This popular model in rumor spreading consists in the following: Given a base network G of size n, in each round t, each node randomly selects one of its outgoing neighbors in G with equal probability to send its round t message. This communication strategy yields a probablity measure on $\mathcal{G}_n$ which is clearly memoryless.

The PUSH model has been extensively studied with various base networks. In this section, we use two seminal works for this model: First, Feige et al. [24] introduced the notion of *almost sure rumor coverage time*, denoted $\mathcal{T}(G)$, and provided a general upper bound on this parameter. Interestingly, $\mathcal{T}(G)$ is equal to our probabilistic order one diameter for the specific value $p = 1 - 1/n$, namely $\hat{D}^{(1)}(1 - 1/n)$. Second, Pittel [33] refined this general bound on $\mathcal{T}(G)$ in the particular case where G is fully-connected.

5.1 The push model in a general symmetric network

In the PUSH communication model, Feige et al. [24] showed that a rumor reaches the n nodes of a symmetric and connected network within $12n\log_2 n$ rounds with probability $1 - 1/n$. In other words, the order one probabilistic diameter satisfies:

$$\hat{D}^{(1)}(1-1/n) \leqslant 12n\log_2 n.$$

Using Lemma 1 and the inequalities $(1-1/n)^4 \geqslant 1-4/n$ and $g^* \geqslant 1$, Theorem 8 then yields the following result.

▶ **Corollary 9.** *Let g be any non-decreasing and inflationary function. For the* PUSH *model in a general symmetric connected network with n nodes, the SAP_g algorithm achieves mod P-synchronization within $324\, n\, (\log_2 n)^2 g^*(24\, P^{-1}\, n\log_2 n)$ rounds with probability $1-\frac{4}{n}$.*

In the context of Corollary 9, Table 1 provides the probabilistic time and space complexities of SAP_g for two different choices of g, namely $g = x \mapsto x+1$ and $g = x \mapsto 2x$ (recall the notation $\overline{M}_0 = \max_{i\in[n]} M_i(0)$). It illustrates the general space-time trade-off that we have just pointed out, at the end of Section 4.

Table 1 The SAP_g algorithm for the PUSH model in a symmetric network of size n.

g	stabilization time	space complexity
$g = x \mapsto x+1$	$O\left(n^2\log^3 n\right)$	$O\left(\log\left(\overline{M}_0 + n\right)\right)$
$g = x \mapsto 2x$	$O\left(n\log^3 n\right)$	$O\left(\log \overline{M}_0 + n\log^3 n\right)$

5.2 The push model in fully-connected networks

In the particular case of fully-connected networks, Frieze and Grimmett [26] improved the above upper bound on the time complexity of rumor spreading in the PUSH model. Their bound as well as its refinement by Pittel [33] are asymptotic in the sense that they hold for sufficiently large networks. This is why our new bound on the stabilization time of the SAP_g algorithm in the particular case of a fully-connected network will be proved to hold only in sufficiently large networks.

Let us briefly recall the main result in [33]: Pittel first defines the random variable S_n as

$$S_n(\mathbb{G}) \overset{\text{def}}{=} \inf\{\delta \in \mathbb{N} \mid \mathbb{G} \in \Gamma_{i_0}^{0,\delta}\},$$

where $\mathbb{G} \in \mathcal{G}_n$, and i_0 is a fixed node. Since the network is fully-connected, all nodes play the same role and S_n does not actually depend on the choice of the origin node i_0.

▶ **Theorem 10** (Theorem 1 in [33]). *If $\omega : \mathbb{N} \to \mathbb{N}$ tends to infinity, then*

$$\lim_{n\to\infty} \Pr_n(|S_n - \log n - \log_2 n| \leqslant \omega(n)) = 1.$$

It follows that for every $p \in [0,1)$ and every such function ω, there exists a positive integer $N_p(\omega)$ such that for all integers $n \geqslant N_p(\omega)$, it holds that

$$\Pr_n\left(S_n - \log n - \log_2 n \leqslant \omega(n)\right) \geqslant \Pr_n\left(|S_n - \log n - \log_2 n| \leqslant \omega(n)\right) \geqslant p. \tag{11}$$

Here, log denotes the natural logarithm. In the PUSH model, all random variables $\mathbb{G}(t)$ are identically distributed. Hence, for all nodes i, and all non-negative integers t and δ, we have $\Pr(\Gamma_i^{t,\delta}) = \Pr(\Gamma_{i_0}^{0,\delta})$, and thus

$$\hat{D}^{(1)}(p) = \inf\left\{\delta \in \mathbb{N} \mid \Pr(\Gamma_{i_0}^{0,\delta}) \geqslant p\right\}.$$

Denoting by N_p the integer $N_p(\log)$ defined in Eq. (11), we obtain that for all integers $n \geqslant N_p$,

$$\hat{D}^{(1)}(p) \leqslant \log_2 n + 2\log n. \tag{12}$$

▶ **Corollary 11.** *Let g be a non-decreasing inflationary function. For any real number $p \in [\frac{1}{2}, 1)$ and any integer $n \geqslant N_p$, the SAP_g algorithm achieves mod P-synchronization within $81 \log(1-p)^{-1} (\log_2 n)\, g^* \left(6\, P^{-1} \log_2 n\right)$ rounds with probability p^4 in the fully-connected graph of size n and the communication* PUSH *model.*

As a complement to Eq. (12), we now compute the value of $\hat{D}^{(1)}(p)$ in small fully-connected networks, and thus obtain an approximation of N_p. For that, we fix a node $i_0 \in [n]$ and an integer $t_0 \in \mathbb{N}$, and we define the random variable $R_n(t)$ by

$$R_n(t) = \Big|\{j \in [n] \mid (i_0, j) \text{ is an arc of } \mathbb{G}(t_0+1 : t_0+t)\}\Big|.$$

Observe that for the PUSH model in a fully-connected graph, the probability distribution of $R_n(t)$ does not depend on the choices of i_0 and t_0. Moreover, the probability $\Pr_n$ is perfectly described by the sequence of the random variables $R_n(1), R_n(2), \ldots$

▶ **Lemma 12.** *Let $a, b \in \{0, \ldots, n\}$. If $a \leqslant b \leqslant 2a$, then*

$$\Pr_n(R_n(t+1) = b \mid R_n(t) = a) = \frac{1}{n^a} \sum_{\ell=b-a}^{a} \binom{a}{\ell}\binom{n-a}{a-\ell} \left\{ \begin{matrix} \ell \\ b-a \end{matrix} \right\} a^{a-\ell} (b-a)!$$

where $\left\{ \begin{smallmatrix} a \\ b \end{smallmatrix} \right\}$ is the Stirling number of the second kind. Otherwise, $\Pr_n(R_n(t+1){=}b \,|\, R_n(t) = a)$ is null.

Proof. We denote by A and B the two sets of nodes that are the targets of an arc whose source is i_0 in the digraphs $\mathbb{G}(1:t)$ and $\mathbb{G}(1:t+1)$, respectively. Thus a node j belongs to B if and only if there exists an arc from A to j in $\mathbb{G}(t+1)$.

In round $t+1$, each node in A picks one node uniformly, among all nodes. Then the total number of draws is n^a. Since each draw is equiprobable, we only have to count the number of favorable draws, that is, the draws such that $|B| = b$. Let ℓ_0 be the number of nodes in A that pick a node in $[n] \setminus A$ in round $t+1$; we have

$$\Pr_n(|B| = b \mid |A| = a) = \sum_{\ell=0}^{a} \Pr_n(|B| = b \cap \ell_0 = \ell \mid |A| = a).$$

We now fix some $\ell_0 \in \{0, \cdots, a\}$, and sample ℓ_0 nodes among the a nodes in A. For that, there are $\binom{a}{\ell_0}$ possibilities. Moreover, we partition the set $[n] \setminus A$ into two parts: $B \setminus A$, of size $b-a$ and $[n] \setminus B$. The number of possible partitionings is $\binom{n-a}{b-a}$. Then there are exactly $\left\{ \begin{smallmatrix} \ell_0 \\ b-a \end{smallmatrix} \right\} (b-a)!$ surjective mappings from the previously chosen set of ℓ_0 nodes in A into the set $B \setminus A$ [36]. Finally, $a - \ell_0$ nodes in A pick a node belonging to A in round $t+1$. There are $a^{a-\ell_0}$ possibilities. Gathering all mentioned terms, and removing terms in which $\left\{ \begin{smallmatrix} \ell_0 \\ b-a \end{smallmatrix} \right\} = 0$, we obtain the final expression of the lemma. ◀

Interestingly, a different expression of $\Pr_n(|B| = b \mid |A| = a)$ was used by Pittel in [33]. Lemma 12 shows that $(R_n(t))_{t\geqslant 1}$ is a Markov process, and provides an effective and efficient way for computing the probability distribution of each random variable $R_n(t)$ as well as the value of $\hat{D}^{(1)}(p)$. Figure 2 reports our results: The straight line represents the theoretical bound provided by Eq. (12) as a function of the logarithm of the network size $\log n$. For each $p \in \{0.5,\ 0.95,\ 0.99\}$, the values of $\hat{D}^{(1)}(p)$ provided by Lemma 12 are denoted by dots.

Figure 2 yields an estimation of N_p: Choosing $p = 0.5$, all the values of $\hat{D}^{(1)}(0.5)$ that we have computed are smaller that the bound provided by Eq. (12). This suggests that Corollary 11 holds for all n, that is, $N_{0.5} = 1$. Similarly, Figure 2 suggests that $N_{0.95} = 59$. By contrast, an estimate for $N_{0.99}$ would require to compute $\hat{D}^{(1)}(0.99)$ for larger values of n.

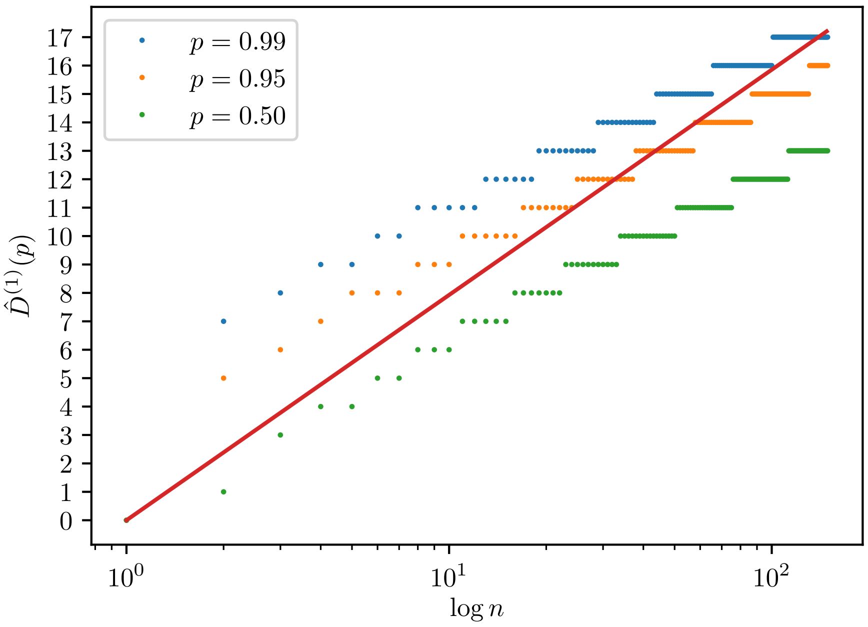


Figure 2 Some values of $\hat{D}^{(1)}(p)$ in the PUSH model, in a fully-connected network of size n.

6 Concluding Remarks

This paper provides a general solution to the mod P-synchronization problem in general probabilistic communication models. Our proof is based on two basic assumptions on the probabilistic network, making it applicable to a broad range of models. The case of the PUSH model for a general symmetric network that we have examined at the end of the paper, provides an example of probabilistic networks for which no clock synchronization algorithms have been yet devised.

Our paper extends the findings of [12], which proved the correctness of SAP_g in a wide class of dynamic networks, including networks that have an infinite dynamic diameter. The probabilistic study developed in this paper significantly enlarges the scope of correctness for SAP_g, and demonstrates the versatility of this algorithm.

References

1 Karine Altisen, Stéphane Devismes, Swan Dubois, and Franck Petit. Introduction to distributed self-stabilizing algorithms. *Synthesis Lectures on Distributed Computing Theory*, 8(1):1–165, 2019.

2 Dana Angluin, James Aspnes, David Eisenstat, and Eric Ruppert. The computational power of population protocols. *Distributed Computing*, 20(4):279–304, 2007. `doi:10.1007/s00446-007-0040-2`.

3 Anish Arora, Shlomi Dolev, and Mohamed G. Gouda. Maintaining digital clocks in step. *Parallel Processing Letters*, 1:11–18, 1991.

4 Paul Bastide, George Giakkoupis, and Hayk Saribekyan. Self-stabilizing clock synchronization with 1-bit messages. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA, 2021*, pages 2154–2173, 2021.

5 Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the Second Symposium on Principles of Distributed Computing*, pages 27–30, 1983.

6 Philip. A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

7 Henrik Björklund, Sven Sandberg, and Sergei Vorobyov. Memoryless determinacy of parity and mean payoff games: a simple proof. *Theoretical Computer Science*, 310(1-3):365–378, 2004.

8 Lucas Boczkowski, Amos Korman, and Emanuele Natale. Minimizing message size in stochastic communication patterns: fast self-stabilizing protocols with 3 bits. *Distributed Comput.*, 32(3):173–191, 2019.

9 Paolo Boldi and Sebastiano Vigna. Universal dynamic synchronous self-stabilization. *Distributed Computing*, 15(3):137–153, 2002.

10 Christian Boulinier, Franck Petit, and Vincent Villain. Synchronous vs. asynchronous unison. *Algorithmica*, 51(1):61–80, 2008.

11 Bernadette Charron-Bost and Shlomo Moran. The firing squad problem revisited. *Theoretical Computer Science*, 793:100–112, 2019.

12 Bernadette Charron-Bost and Louis Penet de Monterno. Self-Stabilizing Clock Synchronization in Dynamic Networks. In *26th International Conference on Principles of Distributed Systems (OPODIS 2022)*, volume 253 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 28:1–28:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023.

13 Bernadette Charron-Bost and Louis Penet de Monterno. Impossibility of self-stabilizing synchronization with bounded memory. ., 2024.

14 Bernadette Charron-Bost and André Schiper. The Heard-Of model: computing in distributed systems with benign faults. *Distributed Computing*, 22(1):49–71, 2009.

15 Herman Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *The Annals of Mathematical Statistics*, pages 493–507, 1952.

16 Michael R Clarkson and Fred B Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

17 Alejandro Cornejo and Fabian Kuhn. Deploying wireless networks with beeps. In *24th International Symposium on Distributed Computing, DISC 2010*, volume 6343 of *Lecture Notes on Computer Science*, pages 148–162. Springer, 2010.

18 Shlomi Dolev. Possible and impossible self-stabilizing digital clock synchronization in general graphs. *Real Time Syst.*, 12(1):95–107, 1997.

19 Shlomi Dolev and Jennifer L. Welch. Self-stabilizing clock synchronization in the presence of byzantine faults. *J. ACM*, 51(5):780–799, 2004.

20 Cynthia Dwork, Nancy A. Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

21 Shimon Even and Sergio Rajsbaum. Unison, canon, and sluggish clocks in networks controlled by a synchronizer. *Mathematical systems theory*, 28(5):421–435, 1995.

22 Shimon Even and Sergio Rajsbaum. Unison, canon, and sluggish clocks in networks controlled by a synchronizer. *Math. Syst. Theory*, 28(5):421–435, 1995.

23 Rui Fan and Nancy Lynch. Gradient clock synchronization. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 320–327, 2004.

24 Uriel Feige, David Peleg, Prabhakar Raghavan, and Eli Upfal. Randomized broadcast in networks. *Random Structures and Algorithms*, 1(4):447–460, 1990.

25 Michael Feldmann, Ardalan Khazraei, and Christian Scheideler. Time- and space-optimal discrete clock synchronization in the beeping model. In *32nd ACM Symposium on Parallelism in Algorithms and Architectures, SPAA*, pages 223–233. ACM, 2020.

26 Alan M Frieze and Geoffrey R Grimmett. The shortest-path problem for graphs with random arc-lengths. *Discrete Applied Mathematics*, 10(1):57–77, 1985.

27 Mohamed Gouda and Ted Herman. Stabilizing unison. *Inf. Process. Lett.*, 35(4):171–175, 1990.

28 Ted Herman and Sukumar Ghosh. Stabilizing phase-clocks. *Information Processing Letters*, 54(5):259–265, 1995.

29 Ali Jadbabaie. Natural algorithms in a networked world: technical perspective. *Commun. ACM*, 55(12):100, 2012.

30 Ronald Kempe, Joseph Y. Dobra, and Moshe Y. Gehrke. Gossip-based computation of aggregate information. In *Proceeding of the 44th IEEE Symposium on Foundations of Computer Science, FOCS*, pages 482–491, Cambridge, MA, USA, 2003.

31 Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

32 Christoph Lenzen, Thomas Locher, and Roger Wattenhofer. Tight bounds for clock synchronization. *Journal of the ACM (JACM)*, 57(2):1–42, 2010.

33 Boris Pittel. On spreading a rumor. *SIAM Journal on Applied Mathematics*, 47(1):213–223, 1987.

34 TK Srikanth and Sam Toueg. Optimal clock synchronization. *Journal of the ACM (JACM)*, 34(3):626–645, 1987.

35 Steven H. Strogatz. From kuramoto to crawford: exploring the onset of synchronization in populations of coupled oscillators. *Physica D*, 143(1-4):1–20, 2000.

36 Horst Wegner. Stirling numbers of the second kind and bonferroni's inequalities. *Elemente der Mathematik*, 60(3):124–129, 2005.

A Extra proofs

First, we state the following lemma, which is a reformulation of Lemma 2 in [12]. We fix any execution ϵ of SAP_g.

▶ **Lemma 13.** *If the clock value of the agent i is greater than 0 at round t, then it is equal to*

$$C_i(t) = 1 + \min_{j \in \mathrm{In}_i(t)} C_j(t-1).$$

▶ **Lemma 5.** *Let d be a positive integer. If $C_i(t) + d \leqslant PM_i(t)$ holds for all nodes i, then all the clocks C_i are greater than 0 in the round interval $[t+1, t+d-1]$.*

Proof. Let i be any node, and let $\ell \in [d-1]$. We have

$$1 + \min_{j \in \mathrm{In}_i(t+\ell)} C_j(t+\ell-1) \leqslant 1 + C_i(t+\ell-1) \leqslant \ell + C_i(t) < PM_i(t) \leqslant PM_i(t+\ell-1).$$

The first inequality is due to the self-loop at node i in $\mathbb{G}(t+\ell)$, the second one is a consequence of the self-loop and Lemma 3, the third inequality is the assumption of the lemma. The fourth one comes from the fact that M_i is non-decreasing. It follows from line 5 that $C_i(t+\ell) \neq 0$. ◀

▶ **Lemma 6.** *Let d be any positive integer, and k be a node such that $C_k(t) = \min_{j\in[n]} C_j(t)$. If the execution ϵ belongs to $\Gamma_k^{t,d}$ and all the clocks C_i are greater than 0 in the round interval $[t+1, t+d-1]$, then the network is synchronized in round $t+d$.*

Proof. We fix an execution ϵ and a positive integer d. First, we prove by induction on $\ell \in [d-1]$ that

$$\forall i \in [n], \quad C_i(t+\ell) = \ell + \min_{j \in \mathrm{In}_i(t+1:t+\ell)} C_j(t). \tag{13}$$

1. The base case $\ell = 1$ is an immediate consequence of Lemma 13.
2. Inductive step: let us assume that Eq. (13) holds for some ℓ with $1 \leqslant \ell < d - 1$. For every node i in $[n]$, we have

$$\begin{aligned} C_i(t+\ell+1) &= 1 + \min_{j \in \mathrm{In}_i(t+\ell+1)} C_j(t+\ell) \\ &= 1 + \ell + \min_{j \in \mathrm{In}_i(t+\ell+1)} \left(\min_{j' \in \mathrm{In}_i(t+1:t+\ell)} C_{j'}(t) \right) \\ &= 1 + \ell + \min_{j \in \mathrm{In}_i(t+1:t+\ell+1)} C_j(t). \end{aligned}$$

   The first equality is a direct consequence of Lemma 13, the second one is by inductive hypothesis, and the third one is due to the fact that $\mathbb{G}(t+1 : t+\ell+1) = \mathbb{G}(t+1 : t+\ell) \circ \mathbb{G}(t+\ell+1)$.

This completes the proof of Eq (13) for every integer $\ell \in [d-1]$.

Then for each node i, we get

$$\begin{aligned} C_i(t+d) &= \left[1 + \min_{j \in \mathrm{In}_i(t+d)} C_j(t+d-1) \right]_{PM_i(t+d-1)} \\ &= \left[d + \min_{j \in \mathrm{In}_i(t+1:t+d)} C_j(t) \right]_{PM_i(t+d-1)} \\ &= [d + C_k(t)]_{PM_i(t+d-1)} . \end{aligned}$$

The second equality comes from a reasoning similar to the inductive case above, using Eq (13) at round $t+d-1$. It follows that all the counters $C_i(t+d)$ are equal modulo P, i.e., the system is synchronized in round $t+d$. ◀

Every Bit Counts in Consensus

Pierre Civit
Sorbonne University, Paris, France

Seth Gilbert
National University of Singapore, Singapore

Rachid Guerraoui
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Jovan Komatovic
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Matteo Monti
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Manuel Vidigueira
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Abstract

Consensus enables n processes to agree on a common valid L-bit value, despite $t < n/3$ processes being faulty and acting arbitrarily. A long line of work has been dedicated to improving the worst-case communication complexity of consensus in partial synchrony. This has recently culminated in the worst-case *word* complexity of $O(n^2)$. However, the worst-case *bit* complexity of the best solution is still $O(n^2L + n^2\kappa)$ (where κ is the security parameter), far from the $\Omega(nL + n^2)$ lower bound. The gap is significant given the practical use of consensus primitives, where values typically consist of batches of large size ($L > n$).

This paper shows how to narrow the aforementioned gap. Namely, we present a new algorithm, DARE (Disperse, Agree, REtrieve), that improves upon the $O(n^2L)$ term via a novel dispersal primitive. DARE achieves $O(n^{1.5}L + n^{2.5}\kappa)$ bit complexity, an effective $\sqrt{n}$-factor improvement over the state-of-the-art (when $L > n\kappa$). Moreover, we show that employing heavier cryptographic primitives, namely STARK proofs, allows us to devise DARE-Stark, a version of DARE which achieves the near-optimal bit complexity of $O(nL + n^2 poly(\kappa))$. Both DARE and DARE-Stark achieve optimal $O(n)$ worst-case latency.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Byzantine consensus, Bit complexity, Latency

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.13

Related Version The full version of this paper, which includes detailed proofs, is available online [37].
Full Version: `https://arxiv.org/abs/2306.00431`

Funding *Seth Gilbert*: Supported in part by the Singapore MOE Tier 2 grant MOE-T2EP20122-0014.
Manuel Vidigueira: Supported in part by the FNS (#200021_215383).

1 Introduction

Byzantine consensus [65] is a fundamental primitive in distributed computing. It has recently risen to prominence due to its use in blockchains [70, 22, 4, 32, 5, 41, 39] and various forms of state machine replication (SMR) [8, 29, 64, 1, 11, 63, 87, 71, 74]. At the same time, the performance of these applications has become directly tied to the performance of consensus and its efficient use of network resources. Specifically, the key limitation on blockchain transaction rates today is network throughput [42, 86, 27]. This has sparked a large demand for research into Byzantine consensus algorithms with better communication complexity guarantees.

Consensus operates among n processes: each process proposes its value, and all processes eventually agree on a common valid L-bit decision. A process can either be correct or faulty: correct processes follow the prescribed protocol, while faulty processes (up to $t < n/3$) can behave arbitrarily. Consensus satisfies the following properties:

- *Agreement:* No two correct processes decide different values.
- *Termination:* All correct processes eventually decide.
- *(External) Validity:* If a correct process decides a value v, then $\mathsf{valid}(v) = \mathit{true}$.

Here, $\mathsf{valid}(\cdot)$ is any predefined logical predicate that indicates whether or not a value is valid.[1]

This paper focuses on improving the worst-case bit complexity of deterministic Byzantine consensus in standard partial synchrony [52]. The worst-case lower bound is $\Omega(nL + n^2)$ exchanged bits. This considers all bits sent by correct processes from the moment the network becomes synchronous, i.e., GST (the number of messages sent by correct processes before GST is unbounded due to asynchrony [85]). The nL term comes from the fact that all n processes must receive the decided value at least once, while the n^2 term is implied by the seminal Dolev-Reischuk lower bound [48, 85] on the number of messages. Recently, a long line of work has culminated in Byzantine consensus algorithms which achieve optimal $O(n^2)$ worst-case *word* complexity, where a word is any constant number of values, signatures or hashes [36, 66]. However, to the best of our knowledge, no existing algorithm beats the $O(n^2L + n^2\kappa)$ bound on the worst-case bit complexity, where κ denotes the security parameter (e.g., the number of bits per hash or signature). The n^2L term presents a linear gap with respect to the lower bound.

Does this gap matter? In practice, yes. In many cases, consensus protocols are used to agree on a large batch of inputs [77, 88, 27, 42, 86]. For example, a block in a blockchain amalgamates many transactions. Alternatively, imagine that n parties each propose a value, and the protocol agrees on a set of these values. (This is often known as vector consensus [14, 15, 50, 80, 49, 40].) Typically, the hope is that by batching values/transactions, we can improve the total throughput of the system. Unfortunately, with current consensus protocols, larger batches do not necessarily yield better performance when applied directly [46]. This does not mean that batches are necessarily ineffective. In fact, a recent line of work has achieved significant practical improvements to consensus throughput by entirely focusing on the efficient dissemination of large batches (i.e., large values), so-called "mempool" protocols [42, 86, 27]. While these solutions work only optimistically (they perform well in periods of synchrony and without faults), they show that a holistic focus on *bandwidth* usage is fundamental (i.e., bit complexity, and not just word complexity).

1.1 Contributions

We introduce DARE (Disperse, Agree, REtrieve), a new Byzantine consensus algorithm for partial synchrony with worst-case $O(n^{1.5}L + n^{2.5}\kappa)$ bit complexity and optimal worst-case $O(n)$ latency. Moreover, by enriching DARE with heavier cryptographic primitives, namely

[1] For traditional notions of validity, admissible values depend on the proposals of correct processes, e.g., if all correct processes start with value v, then v is the only admissible decision. In this paper, we focus on external validity [25], with the observation that any other validity condition can be achieved by reduction (as shown in [38]).

STARK proofs, we close the gap near-optimally using only $O(nL + n^2 poly(\kappa))$ bits. Notice that, if you think of L as a batch of n transactions of size s, the average communication cost of agreeing on a single transaction is only $\tilde{O}(ns)$ bits – the same as a best-effort (unsafe) broadcast [24] of that transaction!

To the best of our knowledge, DARE is the first partially synchronous algorithm to achieve $o(n^2L)$ bit complexity and $O(n)$ latency. The main idea behind DARE is to separate the problem of agreeing from the problem of retrieving an agreed-upon value (see §1.2 for more details). Figure 1 places DARE in the context of efficient consensus algorithms.

Protocol	Model	Cryptography	Complexity	
			$\mathbb{E}$[Bits][†]	$\mathbb{E}$[Latency][†]
ABC [25][‡]	Async	PKI, TS [68]	$O(n^2L + n^2\kappa + n^3)$	$O(1)$
VABA [6]	Async	above	$O(n^2L + n^2\kappa)$	$O(1)$
Dumbo-MVBA [69]	Async	above + ECC [19]	$O(nL + n^2\kappa)$	$O(1)$
			Bits	Latency
PBFT [30, 17]	PSync	PKI	$O(n^2L + n^4\kappa)$	$O(n)$
HotStuff [89]	PSync	above + TS	$O(n^2L + n^3\kappa)$	$O(n)$
Quad [36, 66]	PSync	above	$O(n^2L + n^2\kappa)$	$O(n)$
DARE	PSync	above + ECC	$O(n^{1.5}L + n^{2.5}\kappa)$	$O(n)$
DARE-Stark	PSync	above + STARK	$O(nL + n^2\kappa)$	$O(n)$

Figure 1 Performance of various consensus algorithms with L-bit values and κ-bit security parameter.
[†] For asynchronous algorithms, we show the complexity in expectation instead of the worst-case (which is unbounded for deterministic safety guarantees due to the FLP impossibility result [54]).
[‡] Threshold Signatures (TS) are used to directly improve the original algorithm.

1.2 Technical Overview

The "curse" of GST. To understand the problem that DARE solves, we must first understand why existing algorithms suffer from an $O(n^2L)$ term. "Leader-based" algorithms (such as the state-of-the-art [75, 36, 66]) solve consensus by organizing processes into a rotating sequence of *views*, each with a different leader. A view's leader broadcasts its value v and drives other processes to decide it. If all correct processes are timely and the leader is correct, v is decided.

The main issue is that, if synchrony is only guaranteed *eventually* (partial synchrony [52]), a view might fail to reach agreement even if its leader is correct: the leader could just be slow (i.e., not yet synchronous). The inability to distinguish the two scenarios forces protocols to change views even if the current leader is merely "suspected" of being faulty. Since there can be up to t faulty leaders, there must be at least $t + 1$ different views. However, this comes at the risk of sending unnecessary messages if the suspicion proves false, which is what happens in the worst case.

Suppose that, before GST (i.e., the point in time the system becomes synchronous), the first t leaders are correct, but "go to sleep" (slow down) immediately before broadcasting their values, and receive no more messages until GST + δ due to asynchrony (δ is the maximum message delay after GST). Once GST is reached, all t processes wake up and broadcast their value, for a total of $O(tnL) = O(n^2L)$ exchanged bits; this can happen before they have a chance to receive even a single message! This attack can be considered a "curse" of GST: the *adversarial shift* of correct processes in time creates a (seemingly unavoidable) situation where $\Omega(n^2)$ messages are sent at GST (which in this case include L bits each, for a total of $\Omega(n^2L)$). Figure 2 illustrates the attack.

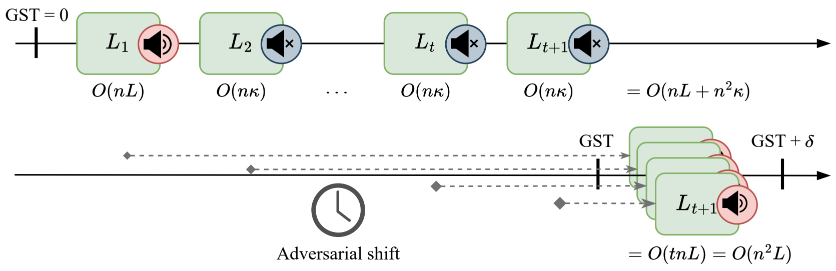


Figure 2 The *adversarial shift* attack on $t + 1$ leaders. The first line shows how leaders are optimistically ordered in time by the protocol to avoid redundant broadcasts (the blue speaker circle represents an *avoided* redundant broadcast). The second line shows how leaders can slow down before GST and overlap at GST, making redundant broadcasts (seem) unavoidable.

DARE: Disperse, Agree, REtrieve. In a nutshell, DARE follows three phases:

1. **Dispersal**: Processes attempt to disperse their values and obtain a *proof of dispersal* for any value. This proof guarantees that the value is both (1) valid, and (2) retrievable.
2. **Agreement**: Processes propose a *hash* of the value accompanied by its proof of dispersal to a Byzantine consensus algorithm for small L (e.g., $O(\kappa)$).
3. **Retrieval**: Using the decided hash, processes retrieve the corresponding value. The proof of dispersal ensures Retrieval will succeed and output a valid value.

This architecture is inspired by randomized asynchronous Byzantine algorithms [6, 69] which work with *expected* bit complexity (the worst-case is unbounded in asynchrony [54]). As these algorithms work in expectation, they can rely on randomness to retrieve a value ($\neq \bot$) that is valid after an expected constant number of tries. However, in order to achieve the same effect (i.e., a constant number of retrievals) in the worst case in partial synchrony, DARE must guarantee that the Retrieval protocol *always* outputs a valid value ($\neq \bot$) *a priori*, which shifts the difficulty of the problem almost entirely to the Dispersal phase.

Dispersal. To obtain a proof of dispersal, a natural solution is for the leader to broadcast the value v. Correct processes check the validity of v (i.e., if $\mathsf{valid}(v) = \mathit{true}$), store v for the Retrieval protocol, and produce a partial signature attesting to these two facts. The leader combines the partial signatures into a $(2t + 1, n)$-threshold signature (the proof of dispersal), which is sufficient to prove that DARE's Retrieval protocol [44] will output a valid value after the Agreement phase.

However, if leaders use best-effort broadcast [24] (i.e., simultaneously send the value to all other processes), they are still vulnerable to an *adversarial shift* causing $O(n^2L)$ communication. Instead, we do the following. First, we use a *view synchronizer* [78, 20, 21] to group leaders into *views* in a rotating sequence. A view has $\sqrt{n}$ leaders and a sequence has $\sqrt{n}$ views. Leaders of the current view can concurrently broadcast their values while messages of other views are ignored. Second, instead of broadcasting the value simultaneously to all processes, a leader broadcasts the value to different subgroups of $\sqrt{n}$ processes in intervals of δ time (i.e., broadcast to the first subgroup, wait δ time, broadcast to the second subgroup, ...) until all processes have received the value. Neither idea individually is enough to improve over the $O(n^2L)$ term. However, when they are combined, it becomes possible to balance the communication cost of the synchronizer ($O(n^{2.5}\kappa)$ bits), the maximum cost of an *adversarial shift* attack ($O(n^{1.5}L)$ bits), and the broadcast rate to achieve the improved $O(n^{1.5}L + n^{2.5}\kappa)$ bit complexity with asymptotically optimal $O(\delta n)$ latency as shown in Figure 3.

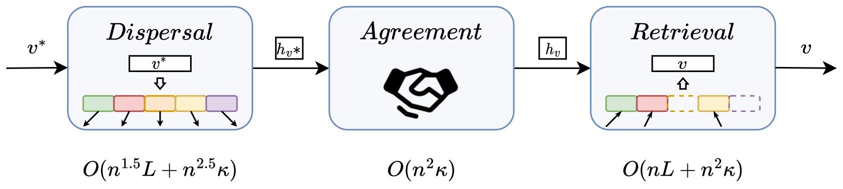


Figure 3 Overview of DARE (Disperse, Agree, REtrieve).

DARE-Stark. As we explained, the main cost of the Dispersal phase is associated with obtaining a dispersal proof that a value is valid. Specifically, it comes from the cost of having to send the entire value (L bits) in a single message.

With Succinct Transparent ARguments of Knowledge (STARKs), we can entirely avoid sending the value in a single message. STARKs allow a process to compute a proof ($O(poly(\kappa))$ bits) of a statement on some value without having to share that value. As an example, a process P_i can send $\langle h, \sigma_{\mathsf{STARK}}\rangle$ to a process P_j, which can use σ_{STARK} to verify the statement "$\exists v : \mathsf{valid}(v) = true \land \mathsf{hash}(v) = h$", all without P_j ever receiving v. As we detail in §6, by carefully crafting a more complex statement, we can modify DARE's Dispersal and Retrieval phases to function with at most $O(poly(\kappa))$ bit-sized messages, obtaining DARE-Stark. This yields the overall near-optimal bit complexity of $O(nL + n^2poly(\kappa))$. Currently, the main drawback of STARKs is their size and computation time in practice[2], which we hope will improve in the future.

Roadmap. We discuss related work in §2. In §3, we define the system model. We give an overview of DARE in §4. In §5, we detail our Dispersal protocol. We go over DARE-Stark in §6. Lastly, we conclude the paper in §7. Detailed proofs are relegated to the full version of the paper.

2 Related Work

We address the communication complexity of deterministic authenticated Byzantine consensus [65, 25] in partially synchronous distributed systems [52] for large inputs. Here, we discuss existing results in closely related contexts, and provide a brief overview of techniques, tools and building blocks which are often employed to tackle Byzantine consensus.[3]

Asynchrony. In the asynchronous setting, Byzantine agreement is commonly known as Multi-valued Validated Byzantine Agreement, or MVBA [25]. Due to the FLP impossibility result [54], deterministic Byzantine agreement is unsolvable in asynchrony (which implies

[2] The associated constants hidden by the "big O" notation result in computation in the order of seconds, proofs in the hundreds of KB, and memory usage several times greater [53].
[3] We use "consensus" and "agreement" interchangeably.

unbounded worst-case complexity). Hence, asynchronous MVBA solutions focus on expected complexity. This line of work was revitalized by HoneyBadgerBFT [73], the first practical fully asynchronous MVBA implementation. Like most other modern asynchronous MVBA protocols, it leverages randomization via a common coin [76], and it terminates in expected $O(\log n)$ time with an expected bit complexity of $O(n^2L + n^3\kappa \log n)$. [6] improves this to $O(1)$ expected time and $O(n^2L + n^2\kappa)$ expected bits, which is asymptotically optimal with $L, \kappa \in O(1)$. Their result is later extended by [69] to large values, improving the complexity to $O(nL+n^2\kappa)$ expected bits. This matches the best known lower bound [85, 3, 48], assuming $\kappa \in O(1)$.

Extension protocols [79, 56, 57, 58]. An extension protocol optimizes for long inputs via a reduction to the same problem with small inputs (considered an oracle). Using extension protocols, several state-of-the-art results were achieved in the authenticated and unauthenticated models, both in synchronous and fully asynchronous settings for Byzantine consensus, Byzantine broadcast and reliable broadcast [79]. Applying the extension protocol of [79] to [75], synchronous Byzantine agreement can be implemented with optimal resiliency ($t < n/2$) and a bit complexity of $O(nL + n^2\kappa)$. Interestingly, it has been demonstrated that synchronous Byzantine agreement can be implemented with a bit complexity of $O(n(L + \mathit{poly}(\kappa)))$ using randomization [18]. The Dolev-Reischuk bound [48] is not violated in this case since the implementation tolerates a negligible (with κ) probability of failure, whereas the bound holds for deterministic protocols. In asynchrony, by applying the (asynchronous) extension protocol of [79] to [6], the same asymptotic result as [69] is achieved, solving asynchronous MVBA with an expected bit complexity of $O(nL + n^2\kappa)$.

Unconditionally secure Byzantine agreement with large inputs has been addressed by [33, 34] under synchrony and [67] under asynchrony, assuming a common coin (implementable via unconditionally-secure Asynchronous Verifiable Secret Sharing [35]). Despite [61] utilizing erasure codes to alleviate leader bottleneck, and the theoretical construction of [38] with exponential latency, there is, to the best of our knowledge, no viable extension protocol for Byzantine agreement in partial synchrony achieving results similar to ours ($o(n^2L)$).

Error correction. Coding techniques, such as erasure codes [19, 60, 10] or error-correction codes [82, 14], appear in state-of-the-art implementations of various distributed tasks: Asynchronous Verifiable Secret Sharing (AVSS) against a computationally bounded [44, 90, 84] or unbounded [35] adversary, Random Beacon [43], Atomic Broadcast in both the asynchronous [55, 62] and partially synchronous [28] settings, Information-Theoretic (IT) Asynchronous State Machine Replication (SMR) [51], Gradecast in synchrony and Reliable Broadcast in asynchrony [2], Asynchronous Distributed Key Generation (ADKG) [44, 45], Asynchronous Verifiable Information Dispersal (AVID) [9], Byzantine Storage [47, 12, 59], and MVBA [69, 79]. Coding techniques are often used to reduce the worst-case complexity by allowing a group of processes to balance and share the cost of sending a value to an individual (potentially faulty) node and are also used in combination with other techniques, such as commitment schemes [31, 69].

We now list several problems related to or used in solving Byzantine agreement.

Asynchronous Common Subset (ACS). The goal in ACS [14, 15, 50] (also known as Vector Consensus [80, 49, 40]) is to agree on a subset of $n - t$ proposals. When considering a generalization of the validity property, this problem represents the strongest variant of consensus [38]. Atomic Broadcast can be trivially reduced to ACS [49, 25, 73]. There are well-known simple asynchronous constructions that allow for the reduction of ACS to either (1) Reliable Broadcast and Binary Byzantine Agreement [15], or (2) MVBA [25] in the authenticated setting, where the validation predicate requires the output to be a vector of signed inputs from at least $n - t$ parties. The first reduction enables the implementation of ACS with a cubic bit complexity, using the broadcast of [2]. The second reduction could be improved further with a more efficient underlying MVBA protocol, such as DARE-Stark.

Asynchronous Verifiable Information Dispersal (AVID). AVID [26] is a form of "retrievable" broadcast that allows the dissemination of a value while providing a cryptographic proof that it can be retrieved. This primitive can be implemented with a total dispersal cost of $O(L + n^2\kappa)$ bits exchanged and a retrieval cost of $O(L + n\kappa)$ per node, relying only on the existence of collision-resistant hash functions [9]. AVID is similar to our Dispersal and Retrieval phases, but has two key differences. First, AVID's retrieval protocol only guarantees that a valid value will be retrieved if the original process dispersing the information was correct. Second, it is a broadcast protocol, having stricter delivery guarantees for each process. Concretely, if a correct process initiates the AVID protocol, it should eventually disperse its *own* value. In contrast, we only require that a correct process obtains a proof of dispersal for *some* value.

Provable Broadcast (PB) and Asynchronous Provable Dispersal Broadcast (APDB). PB [7] is a primitive used to acquire a succinct proof of external validity. It is similar to our Dispersal phase, including the algorithm itself, but without the provision of a proof of dispersal (i.e., retrievability, only offering proof of validity). The total bit complexity for n PB-broadcasts from distinct processes amounts to $O(n^2L)$. APDB [69] represents an advancement of AVID, drawing inspiration from PB. It sacrifices PB's validity guarantees to incorporate AVID's dissemination and retrieval properties. By leveraging the need to retrieve and validate a value a constant number of times in expectation, [69] attains optimal $O(nL + n^2\kappa)$ expected complexity in asynchrony. However, this approach falls short in the worst-case scenario of a partially synchronous solution, where n reconstructions would cost $\Omega(n^2L)$.

Asynchronous Data Dissemination (ADD). In ADD [44], a subset of $t+1$ correct processes initially share a common L-sized value v, and the goal is to disseminate v to all correct processes, despite the presence of up to t Byzantine processes. The approach of [44] is information-theoretically secure, tolerates up to one-third malicious nodes and has a bit complexity of $O(nL + n^2 \log n)$. (In DARE, we rely on ADD in a "closed-box" manner; see §4.)

3 Preliminaries

Processes. We consider a static set $\mathsf{Process} = \{P_1, P_2, ..., P_n\}$ of $n = 3t + 1$ processes, out of which (at most) $t > 0$ can be Byzantine and deviate arbitrarily from their prescribed protocol. A Byzantine process is said to be *faulty*; a non-faulty process is said to be *correct*. Processes communicate by exchanging messages over an authenticated point-to-point network. Furthermore, the communication network is reliable: if a correct process sends a message to a correct process, the message is eventually received. Processes have local hardware clocks. Lastly, we assume that local steps of processes take zero time, as the time needed for local computation is negligible compared to the message delays.

Partial synchrony. We consider the standard partially synchronous model [52]. For every execution, there exists an unknown Global Stabilization Time (GST) and a positive duration δ such that the message delays are bounded by δ after GST. We assume that δ is known by processes. All correct processes start executing their prescribed protocol by GST. The hardware clocks of processes may drift arbitrarily before GST, but do not drift thereafter. We underline that our algorithms require minimal changes to preserve their correctness even if δ is unknown (these modifications are specified in Appendix C.2), although their complexity might be higher.

Cryptographic primitives. Throughout the paper, $\mathsf{hash}(\cdot)$ denotes a collision-resistant hash function. The codomain of the aforementioned $\mathsf{hash}(\cdot)$ function is denoted by $\mathsf{Hash\_Value}$.

Moreover, we assume a (k, n)-threshold signature scheme [83], where $k = n - t = 2t + 1$. In this scheme, each process holds a distinct private key, and there is a single public key. Each process P_i can use its private key to produce a partial signature for a message m by invoking $\mathsf{share\_sign}_i(m)$. A set of partial signatures S for a message m from k distinct processes can be combined into a single threshold signature for m by invoking $\mathsf{combine}(S)$; a threshold signature for m proves that k processes have (partially) signed m. Furthermore, partial and threshold signatures can be verified: given a message m and a signature Σ_m, $\mathsf{verify\_sig}(m, \Sigma_m)$ returns *true* if and only if Σ_m is a valid signature for m. Where appropriate, the verifications are left implicit. We denote by $\mathsf{P\_Signature}$ and $\mathsf{T\_Signature}$ the set of partial and threshold signatures, respectively. The size of cryptographic objects (i.e., hashes, signatures) is denoted by κ; we assume that $\kappa > \log n$.[4]

Reed-Solomon codes [82]. Our algorithms rely on Reed-Solomon (RS) codes [81]. Concretely, DARE utilizes (in a "closed-box" manner) an algorithm which internally builds upon error-correcting RS codes. DARE-Stark directly uses RS erasure codes (no error correction is required).

We use $\mathsf{encode}(\cdot)$ and $\mathsf{decode}(\cdot)$ to denote RS' encoding and decoding algorithms. In a nutshell, $\mathsf{encode}(\cdot)$ takes a value v, chunks it into the coefficients of a polynomial of degree t (the maximum number of faults), and outputs n (the total number of processes) evaluations of the polynomial (RS symbols); Symbol denotes the set of RS symbols. $\mathsf{decode}(\cdot)$ takes a set of $t + 1$ RS symbols S and interpolates them into a polynomial of degree t, whose coefficients are concatenated and output.

Complexity of Byzantine consensus. Let $\mathsf{Consensus}$ be a partially synchronous Byzantine consensus algorithm, and let $\mathcal{E}(\mathsf{Consensus})$ denote the set of all possible executions. Let $\alpha \in \mathcal{E}(\mathsf{Consensus})$ be an execution, and $t_d(\alpha)$ be the first time by which all correct processes have decided in α. The bit complexity of α is the total number of bits sent by correct processes during the time period $[\text{GST}, \infty)$. The latency of α is $\max(0, t_d(\alpha) - \text{GST})$.

The *bit complexity* of $\mathsf{Consensus}$ is defined as

$$\max_{\alpha \in \mathcal{E}(\mathsf{Consensus})} \bigg\{ \text{bit complexity of } \alpha \bigg\}.$$

[4] For $\kappa \leq \log n$, $t \in O(n)$ faulty processes would have computational power exponential in κ, breaking cryptographic hardness assumptions.

Similarly, the *latency* of Consensus is defined as

$$\max_{\alpha \in \mathcal{E}(\mathsf{Consensus})} \Big\{ \text{latency of } \alpha \Big\}.$$

4 DARE

This section presents DARE (Disperse, Agree, REtrieve), which is composed of three algorithms:

1. Disperser, which disperses the proposals;
2. Agreement, which ensures agreement on the hash of a previously dispersed proposal; and
3. Retriever, which rebuilds the proposal corresponding to the agreed-upon hash.

We start by introducing the aforementioned building blocks (§4.1). Then, we show how they are composed into DARE (§4.2). Finally, we prove the correctness and complexity of DARE (§4.3).

4.1 Building Blocks: Overview

In this subsection, we formally define the three building blocks of DARE. Concretely, we define their interface and properties, as well as their complexity.

4.1.1 Disperser

Interface & properties. Disperser solves a problem similar to that of AVID [26]. In a nutshell, each correct process aims to disperse its value to all correct processes: eventually, all correct processes acquire a proof that a value with a certain hash has been successfully dispersed.

Concretely, Disperser exposes the following interface:

- **request** $\mathsf{disperse}(v \in \mathsf{Value})$: a process disperses a value v; each correct process invokes $\mathsf{disperse}(v)$ exactly once and only if $\mathsf{valid}(v) = \mathit{true}$.
- **indication** $\mathsf{acquire}(h \in \mathsf{Hash\_Value}, \Sigma_h \in \mathsf{T\_Signature})$: a process acquires a pair (h, Σ_h).

We say that a correct process *obtains* a threshold signature (resp., a value) if and only if it stores the signature (resp., the value) in its local memory. (Obtained values can later be retrieved by all correct processes using Retriever; see §4.1.3 and Algorithm 1.) Disperser ensures the following:

- *Integrity:* If a correct process acquires a hash-signature pair (h, Σ_h), then $\mathsf{verify\_sig}(h, \Sigma_h) = \mathit{true}$.
- *Termination:* Every correct process eventually acquires at least one hash-signature pair.
- *Redundancy:* Let a correct process obtain a threshold signature Σ_h such that $\mathsf{verify\_sig}(h, \Sigma_h) = \mathit{true}$, for some hash value h. Then, (at least) $t + 1$ correct processes have obtained a value v such that (1) $\mathsf{hash}(v) = h$, and (2) $\mathsf{valid}(v) = \mathit{true}$.

Note that it is not required for all correct processes to acquire the same hash value (nor the same threshold signature). Moreover, the specification allows for multiple acquired pairs.

Complexity. Disperser exchanges $O(n^{1.5}L + n^{2.5}\kappa)$ bits after GST. Moreover, it terminates in $O(n)$ time after GST.

Implementation. The details on DISPERSER's implementation are relegated to §5.

4.1.2 Agreement

Interface & properties. AGREEMENT is a Byzantine consensus algorithm.[5] In AGREEMENT, processes propose and decide pairs ($h \in \mathsf{Hash\_Value}, \Sigma_h \in \mathsf{T\_Signature}$); moreover, $\mathsf{valid}(h, \Sigma_h) \equiv \mathsf{verify\_sig}(h, \Sigma_h)$.

Complexity. AGREEMENT achieves $O(n^2\kappa)$ bit complexity and $O(n)$ latency.

Implementation. We "borrow" the implementation from [36]. In brief, AGREEMENT is a "leader-based" consensus algorithm whose computation unfolds in views. Each view has a single leader, and it employs a "leader-to-all, all-to-leader" communication pattern. AGREEMENT's safety relies on standard techniques [89, 30, 23, 66]: (1) quorum intersection (safety within a view), and (2) "locking" mechanism (safety across multiple views). As for liveness, AGREEMENT guarantees termination once all correct processes are in the same view (for "long enough" time) with a correct leader. (For full details on AGREEMENT, see [36].)

4.1.3 Retriever

Interface & properties. In RETRIEVER, each correct process starts with either (1) some value, or (2) $\bot$. Eventually, all correct processes output the same value. Formally, RETRIEVER exposes the following interface:

- **request** $\mathsf{input}(v \in \mathsf{Value} \cup \{\bot\})$: a process inputs a value or $\bot$; each correct process invokes $\mathsf{input}(\cdot)$ exactly once. Moreover, the following is assumed:
  - No two correct processes invoke $\mathsf{input}(v_1 \in \mathsf{Value})$ and $\mathsf{input}(v_2 \in \mathsf{Value})$ with $v_1 \neq v_2$.
  - At least $t+1$ correct processes invoke $\mathsf{input}(v \in \mathsf{Value})$ (i.e., $v \neq \bot$).
- **indication** $\mathsf{output}(v' \in \mathsf{Value})$: a process outputs a value v'.

The following properties are ensured:

- *Agreement:* No two correct processes output different values.
- *Validity:* Let a correct process input a value v. No correct process outputs a value $v' \neq v$.
- *Termination:* Every correct process eventually outputs a value.

Complexity. RETRIEVER exchanges $O(nL + n^2 \log n)$ bits after GST (and before every correct process outputs a value). Moreover, RETRIEVER terminates in $O(1)$ time after GST.

Implementation. RETRIEVER's implementation is "borrowed" from [44]. In summary, RETRIEVER relies on Reed-Solomon codes [82] to encode the input value $v \neq \bot$ into n symbols. Each correct process Q which inputs $v \neq \bot$ to RETRIEVER encodes v into n RS symbols $s_1, s_2, ..., s_n$. Q sends each RS symbol s_i to the process P_i. When P_i receives $t+1$ identical RS symbols s_i, P_i is sure that s_i is a "correct" symbol (i.e., it can be used to rebuild v) as it was computed by at least one correct process. At this moment, P_i broadcasts s_i. Once each correct process P receives $2t+1$ (or more) RS symbols, P tries to rebuild v (with some error-correction). (For full details on RETRIEVER, see [44].)

[5] Recall that the interface and properties of Byzantine consensus algorithms are introduced in §1.

Algorithm 1 DARE: Pseudocode (for process P_i).

```
1: Uses:
2:     ▷ bits: $O(n^{1.5}L + n^{2.5}\kappa)$, latency: $O(n)$ (see §5)
3:     Disperser, instance disperser
4:     ▷ bits: $O(n^2\kappa)$, latency: $O(n)$ (see [36])
5:     Agreement, instance agreement
6:     ▷ bits: $O(nL + n^2 \log n)$, latency: $O(1)$ (see [44])
7:     Retriever, instance retriever
8: upon propose($v_i \in$ Value):
9:     invoke disperser.disperse($v_i$)
10: upon disperser.acquire($h_i \in$ Hash_Value, $\Sigma_i \in$ T_Signature):
11:     invoke agreement.propose($h_i, \Sigma_i$)
12: upon agreement.decide($h \in$ Hash_Value, $\Sigma_h \in$ T_Signature):        ▷ $P_i$ obtains $\Sigma_h$
13:     $v \leftarrow$ an obtained value such that hash($v$) $= h$ (if such a value was not obtained, $v = \bot$)
14:     invoke retriever.input($v$)
15: upon retriever.output(Value $v'$):
16:     trigger decide($v'$)
```

4.2 Pseudocode

Algorithm 1 gives DARE's pseudocode. We explain it from the perspective of a correct process P_i. An execution of DARE consists of three phases (each of which corresponds to one building block):

1. *Dispersal:* Process P_i disperses its proposal v_i using Disperser (line 9). Eventually, P_i acquires a hash-signature pair (h_i, Σ_i) (line 10) due to the termination property of Disperser.
2. *Agreement:* Process P_i proposes the previously acquired hash-signature pair (h_i, Σ_i) to Agreement (line 11). As Agreement satisfies termination and agreement, all correct processes eventually agree on a hash-signature pair (h, Σ_h) (line 12).
3. *Retrieval:* Once process P_i decides (h, Σ_h) from Agreement, it checks whether it has previously obtained a value v with hash$(v) = h$ (line 13). If it has, P_i inputs v to Retriever; otherwise, P_i inputs $\bot$ (line 14). The required preconditions for Retriever are met:
   - No two correct processes input different non-$\bot$ values to Retriever as hash$(\cdot)$ is collision-resistant.
   - At least $(t + 1)$ correct processes input a value (and not $\bot$) to Retriever. Indeed, as Σ_h is obtained by a correct process, $t + 1$ correct processes have obtained a value $v \neq \bot$ with hash$(v) = h$ (due to redundancy of Disperser), and all of these processes input v.

   Therefore, all correct processes (including P_i) eventually output the same value v' from Retriever (due to the termination property of Retriever; line 15), which represents the decision of DARE (line 16). Note that $v' = v \neq \bot$ due to the validity of Retriever.

4.3 Proof of Correctness & Complexity

We start by proving the correctness of DARE.

▶ **Theorem 1.** *DARE is correct.*

Proof. Every correct process starts the dispersal of its proposal (line 9). Due to the termination property of Disperser, every correct process eventually acquires a hash-signature pair (line 10). Hence, every correct process eventually proposes to Agreement (line 11), which implies that every correct process eventually decides the same hash-signature pair (h, Σ_h) from AGREEMENT (line 12) due to the agreement and termination properties of AGREEMENT.

As (h, Σ_h) is decided by all correct processes, at least $t+1$ correct processes P_i have obtained a value v such that (1) $\mathsf{hash}(v) = h$, and (2) $\mathsf{valid}(v) = \mathit{true}$ (due to the redundancy property of DISPERSER). Therefore, all of these correct processes input v to RETRIEVER (line 14). Moreover, no correct process inputs a different value (as $\mathsf{hash}(\cdot)$ is collision-resistant). Thus, the conditions required by RETRIEVER are met, which implies that all correct processes eventually output the same valid value (namely, v) from RETRIEVER (line 15), and decide it (line 16). ◀

Next, we prove the complexity of DARE.

▶ **Theorem 2.** *DARE achieves $O(n^{1.5}L + n^{2.5}\kappa)$ bit complexity and $O(n)$ latency.*

Proof. As DARE is a sequential composition of its building blocks, its complexity is the sum of the complexities of (1) DISPERSER, (2) AGREEMENT, and (3) RETRIEVER. Hence, the bit complexity is

$$\underbrace{O(n^{1.5}L + n^{2.5}\kappa)}_{\text{DISPERSER}} + \underbrace{O(n^2\kappa)}_{\text{AGREEMENT}} + \underbrace{O(nL + n^2\log n)}_{\text{RETRIEVER}} = O(n^{1.5}L + n^{2.5}\kappa).$$

Similarly, the latency is $O(n)$. ◀

5 Disperser: Implementation & Analysis

This section focuses on DISPERSER. Namely, we present its implementation (§5.1), and (informally) analyze its correctness and worst-case complexity (§5.2). Formal proofs can be found in the full version of the paper [37]. An analysis of the good (common) case can be found in Appendix C.1.

5.1 Implementation

DISPERSER's pseudocode is given in Algorithm 2. In essence, each execution unfolds in *views*, where each view has X *leaders* ($0 < X \leq n$ is a generic parameter); the set of all views is denoted by View. Given a view V, $\mathsf{leaders}(V)$ denotes the X-sized set of leaders of the view V. In each view, a leader disperses its value to Y-sized groups of processes ($0 < Y \leq n$ is a generic parameter) at a time (line 14), with a δ-waiting step in between (line 15). Before we thoroughly explain the pseudocode, we introduce SYNC, DISPERSER's view synchronization [36, 89, 66] algorithm.

Sync. Its responsibility is to bring all correct processes to the same view with a correct leader for (at least) $\Delta = \delta\frac{n}{Y} + 3\delta$ time. Precisely, SYNC exposes the following interface:

- **indication** $\mathsf{advance}(V \in \mathsf{View})$: a process enters a new view V.

SYNC guarantees *eventual synchronization*: there exists a time $\tau_{sync} \geq \text{GST}$ (*synchronization time*) such that (1) all correct processes are in the same view V_{sync} (*synchronization view*) from time τ_{sync} to (at least) time $\tau_{sync} + \Delta$, and (2) V_{sync} has a correct leader. We denote by V^*_{sync} the smallest synchronization view, whereas τ^*_{sync} denotes the first synchronization time. Similarly, V_{max} denotes the greatest view entered by a correct process before GST.[6]

[6] When such a view does not exist, $V_{max} = 0$

The implementation of Sync (see Appendix A) is highly inspired by RareSync, a view synchronization algorithm introduced in [36]. In essence, when a process enters a new view, it stays in the view for $O(\Delta) = O(\frac{n}{Y})$ time. Once it wishes to proceed to the next view, the process engages in an "all-to-all" communication step (which exchanges $O(n^2\kappa)$ bits); this step signals the end of the current view, and the beginning of the next one. Throughout views, leaders are rotated in a round-robin manner: each process is a leader for exactly one view in any sequence of $\frac{n}{X}$ consecutive views. As $O(\frac{n}{X})$ views (after GST) are required to reach a correct leader, Sync exchanges $O(\frac{n}{X}) \cdot O(n^2\kappa) = O(\frac{n^3\kappa}{X})$ bits (before synchronization, i.e., before $\tau^*_{sync} + \Delta$); since each view takes $O(\frac{n}{Y})$ time, synchronization is ensured within $O(\frac{n}{X}) \cdot O(\frac{n}{Y}) = O(\frac{n^2}{XY})$ time.

Disperser relies on the following properties of Sync (along with eventual synchronization):

- *Monotonicity:* Any correct process enters monotonically increasing views.
- *Stabilization:* Any correct process enters a view $V \geq V_{max}$ by time $\text{GST} + 3\delta$.
- *Limited entrance:* In the time period $[\text{GST}, \text{GST} + 3\delta)$, any correct process enters $O(1)$ views.
- *Overlapping:* For any view $V > V_{max}$, all correct processes overlap in V for (at least) Δ time.
- *Limited synchronization view:* $V^*_{sync} - V_{max} = O(\frac{n}{X})$.
- *Complexity:* Sync exchanges $O(\frac{n^3\kappa}{X})$ bits during the time period $[\text{GST}, \tau^*_{sync} + \Delta]$, and it synchronizes all correct processes within $O(\frac{n^2}{XY})$ time after GST ($\tau^*_{sync} + \Delta - \text{GST} = O(\frac{n^2}{XY})$).

The aforementioned properties of Sync are formally proven in Appendix A.

Algorithm description. Correct processes transit through views based on Sync's indications (line 10): when a correct process receives advance(V) from Sync, it stops participating in the previous view and starts participating in V.

Once a correct leader P_l enters a view V, it disperses its proposal via dispersal messages. As already mentioned, P_l sends its proposal to Y-sized groups of processes (line 14) with a δ-waiting step in between (line 15). When a correct (non-leader) process P_i (which participates in the view V) receives a dispersal message from P_l, P_i checks whether the dispersed value is valid (line 17). If it is, P_i partially signs the hash of the value, and sends it back to P_l (line 20). When P_l collects $2t + 1$ ack messages, it (1) creates a threshold signature for the hash of its proposal (line 24), and (2) broadcasts the signature (along with the hash of its proposal) to all processes via a confirm message (line 25). Finally, when P_l (or any other correct process) receives a confirm message (line 27), it (1) acquires the received hash-signature pair (line 28), (2) disseminates the pair to "help" the other processes (line 29), and (3) stops executing Disperser (line 30).

5.2 Analysis

Correctness. Once all correct processes synchronize in the view V^*_{sync} (the smallest synchronization view), all correct processes acquire a hash-signature pair. Indeed, $\Delta = \delta\frac{n}{Y} + 3\delta$ time is sufficient for a correct leader $P_l \in \text{leaders}(V^*_{sync})$ to (1) disperse its proposal $\mathit{proposal}_l$ to all processes (line 14), (2) collect $2t + 1$ partial signatures for $h = \text{hash}(\mathit{proposal}_l)$ (line 23), and (3) disseminate a threshold signature for h (line 25). When a correct process receives the aforementioned threshold signature (line 27), it acquires the hash-signature pair (line 28) and stops executing Disperser (line 30).

Algorithm 2 DISPERSER: Pseudocode (for process P_i).

```
1:  Uses:
2:      SYNC, instance $sync$                ▷ ensures a $\Delta = \delta\frac{n}{Y} + 3\delta$ overlap in a view with a correct leader
3:  upon init:
4:      Value $proposal_i \leftarrow \bot$
5:      Integer $received\_acks_i \leftarrow 0$
6:      Map(Hash_Value → Value) $obtained\_values_i \leftarrow$ empty
7:  upon disperse(Value $v_i$):
8:      $proposal_i \leftarrow v_i$
9:      start $sync$
10: upon $sync$.advance(View $V$):           ▷ $P_i$ stops participating in the previous view
11:     ▷ First part of the view
12:     if $P_i \in$ leaders($V$):
13:         for Integer $k \leftarrow 1$ to $\frac{n}{Y}$:
14:             send ⟨DISPERSAL, $proposal_i$⟩ to $P_{(k-1)Y+1}, P_{(k-1)Y+2}, ..., P_{kY}$
15:             wait $\delta$ time
16:     every process:
17:         upon reception of ⟨DISPERSAL, Value $v_j$⟩ from process $P_j \in$ leaders($V$) and valid($v_j$) $= true$:
18:             Hash_Value $h \leftarrow$ hash($v_j$)
19:             $obtained\_values_i[h] \leftarrow v_j$
20:             send ⟨ACK, share_sign$_i(h)$⟩ to $P_j$
21:     ▷ Second part of the view
22:     if $P_i \in$ leaders($V$):
23:         upon exists Hash_Value $h$ such that ⟨ACK, $h$, ·⟩ has been received from $2t+1$ processes:
24:             T_Signature $\Sigma_h \leftarrow$ combine({P_Signature $sig$ | $sig$ is received in the ACK messages})
25:             broadcast ⟨CONFIRM, $h$, $\Sigma_h$⟩
26:     every process:
27:         upon reception of ⟨CONFIRM, Hash_Value $h$, T_Signature $\Sigma_h$⟩ and verify_sig($h, \Sigma_h$) $= true$:
28:             trigger acquire($h, \Sigma_h$)
29:             broadcast ⟨CONFIRM, $h$, $\Sigma_h$⟩
30:             stop executing DISPERSER (and SYNC)
```

Complexity. DISPERSER terminates once all correct processes are synchronized in a view with a correct leader. The synchronization is ensured in $O(\frac{n^2}{XY})$ time after GST (as $\tau^*_{sync} + \Delta - \text{GST} = O(\frac{n^2}{XY})$). Hence, DISPERSER terminates in $O(\frac{n^2}{XY})$ time after GST.

Let us analyze the number of bits DISPERSER exchanges. Any execution of DISPERSER can be separated into two post-GST periods: (1) *unsynchronized*, from GST until GST + 3δ, and (2) *synchronized*, from GST + 3δ until $\tau^*_{sync} + \Delta$. First, we study the number of bits correct processes send via DISPERSAL, ACK and CONFIRM message in the aforementioned periods:

- Unsynchronized period: Due to the δ-waiting step (line 15), each correct process sends DISPERSAL messages (line 14) to (at most) $3 = O(1)$ Y-sized groups. Hence, each correct process sends $O(1) \cdot O(Y) \cdot L = O(YL)$ bits through DISPERSAL messages.

  Due to the limited entrance property of SYNC, each correct process enters $O(1)$ views during the unsynchronized period. In each view, each correct process sends (at most) $O(X)$ ACK messages (one to each leader; line 20) and $O(n)$ CONFIRM messages (line 25). As each ACK and CONFIRM message carries κ bits, all correct processes send

$$n \cdot \big(\underbrace{O(YL)}_{\text{DISPERSAL}} + \underbrace{O(X\kappa)}_{\text{ACK}} + \underbrace{O(n\kappa)}_{\text{CONFIRM}} \big)$$
$$= O(nYL + n^2\kappa) \text{ bits via DISPERSAL, ACK and CONFIRM messages.}$$

- Synchronized period: Recall that all correct processes acquire a hash-signature pair (and stop executing DISPERSER) by time $\tau^*_{sync} + \Delta$, and they do so in the view V^*_{sync}. As correct processes enter monotonically increasing views, no correct process enters a view greater than V^*_{sync}.
  By the stabilization property of SYNC, each correct process enters a view $V \geq V_{max}$ by time GST $+ 3\delta$. Moreover, until $\tau^*_{sync} + \Delta$, each correct process enters (at most) $O(\frac{n}{X})$ views (due to the limited synchronization view and monotonicity properties of SYNC). Importantly, no correct leader exists in any view V with $V_{max} < V < V^*_{sync}$; otherwise, $V = V^*_{sync}$ as processes overlap for Δ time in V (due to the overlapping property of SYNC). Hence, for each view V with $V_{max} < V < V^*_{sync}$, all correct processes send $O(nX\kappa)$ bits (all through ACK messages; line 20). In V_{max} and V^*_{sync}, all correct processes send (1) $2 \cdot O(XnL)$ bits through DISPERSAL messages (line 14), (2) $2 \cdot O(nX\kappa)$ bits through ACK messages (line 20), and (3) $2 \cdot O(Xn\kappa)$ bits through CONFIRM messages (line 25). Therefore, all correct processes send
  $$\underbrace{O(\frac{n}{X})}_{V^*_{sync} - V_{max}} \cdot \underbrace{O(nX\kappa)}_{\text{ACK}} + \underbrace{O(XnL)}_{\text{DISPERSAL in } V_{max} \text{ and } V^*_{sync}} + \underbrace{O(nX\kappa)}_{\text{ACK in } V_{max} \text{ and } V^*_{sync}} + \underbrace{O(Xn\kappa)}_{\text{CONFIRM in } V_{max} \text{ and } V^*_{sync}}$$
  $= O(nXL + n^2\kappa)$ bits via DISPERSAL, ACK and CONFIRM messages.

We cannot neglect the complexity of SYNC, which exchanges $O(\frac{n^3\kappa}{X})$ bits during the time period $[\text{GST}, \tau^*_{sync} + \Delta]$. Hence, the total number of bits DISPERSER exchanges is

$$\underbrace{O(nYL + n^2\kappa)}_{\text{unsynchronized period}} + \underbrace{O(nXL + n^2\kappa)}_{\text{synchronized period}} + \underbrace{O(\frac{n^3\kappa}{X})}_{\text{SYNC}} = O(nYL + nXL + \frac{n^3\kappa}{X}).$$

With $X = Y = \sqrt{n}$, DISPERSER terminates in optimal $O(n)$ time, and exchanges $O(n^{1.5}L + n^{2.5}\kappa)$ bits. Our analysis is illustrated in Figure 4.

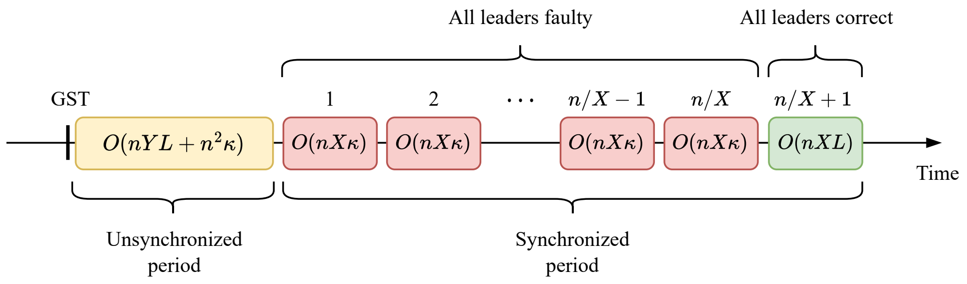


Figure 4 Illustration of DISPERSER's bit complexity.

6 DARE-Stark

In this section, we present DARE-STARK, a variant of DARE which relies on STARK proofs. Importantly, DARE-STARK achieves $O(nL + n^2 poly(\kappa))$ bit complexity, nearly tight to the $\Omega(nL + n^2)$ lower bound, while preserving optimal $O(n)$ latency.

First, we revisit DISPERSER, pinpointing its complexity on proving RS encoding (§6.1). We then present DARE-STARK, which uses STARKs for provable RS encoding, thus improving on DARE's complexity (§6.2). For a brief overview on STARKs, a cryptographic primitive providing succinct proofs of knowledge, we invite the interested reader to Appendix B.

6.1 Revisiting DARE: What Causes Disperser's Complexity?

Recall that DISPERSER exchanges $O(n^{1.5}L + n^{2.5}\kappa)$ bits. This is due to a fundamental requirement of RETRIEVER: at least $t+1$ correct processes must have obtained the value v by the time AGREEMENT decides $h = \mathsf{hash}(v)$. RETRIEVER leverages this requirement to prove the correct encoding of RS symbols. In brief (as explained in §4.1.3): (1) every correct process P that obtained $v \neq \bot$ encodes it in n RS symbols $s_1, \ldots, s_n$; (2) P sends each s_i to P_i; (3) upon receiving $t+1$ identical copies of s_i, P_i can trust s_i to be the i-th RS symbol for v (note that s_i can be trusted only because it was produced by at least one correct process – nothing else proves s_i's relationship to v!); (4) every correct process P_i disseminates s_i, enabling the reconstruction of v by means of error-correcting decoding. In summary, DARE bottlenecks on DISPERSER, and DISPERSER's complexity is owed to the need to prove the correct encoding of RS symbols in RETRIEVER. Succinct arguments of knowledge (such as STARKs), however, allow to publicly prove the relationship between an RS symbol and the value it encodes, eliminating the need to disperse the entire value to $t+1$ correct processes – a dispersal of provably correct RS symbols suffices. DARE-STARK builds upon this idea.

6.2 Implementation

Provably correct encoding. At its core, DARE-STARK uses STARKs to attest the correct RS encoding of values. For every $i \in [1, n]$, we define $\mathsf{shard}_i(\cdot)$ by

$$\mathsf{shard}_i(v \in \mathsf{Value}) = \begin{cases} \big(\mathsf{hash}(v), \mathsf{encode}_i(v)\big), & \text{if and only if } \mathsf{valid}(v) = \mathit{true} \\ \bot, & \text{otherwise,} \end{cases} \tag{1}$$

where $\mathsf{encode}_i(v)$ represents the i-th RS symbol obtained from $\mathsf{encode}(v)$ (see §3). We use $\mathsf{proof}_i(v)$ to denote the STARK proving the correct computation of $\mathsf{shard}_i(v)$. The design and security of DARE-STARK rests on the following theorem.

▶ **Theorem 3.** *Let $i_1, \ldots, i_{t+1}$ be distinct indices in $[1, n]$. Let h be a hash, let $s_1, \ldots, s_{t+1}$ be RS symbols, let $stark_1, \ldots, stark_{t+1}$ be STARK proofs such that, for every $k \in [1, t+1]$, $stark_k$ proves knowledge of some (undisclosed) v_k such that $\mathsf{shard}_{i_k}(v_k) = (h, s_k)$. We have that*

$$v = \mathsf{decode}(\{s_1, \ldots, s_k\})$$

satisfies $\mathsf{valid}(v) = \mathit{true}$ and $\mathsf{hash}(v) = h$.

Proof. For all k, by the correctness of $stark_k$ and Equation (1), we have that (1) $h = \mathsf{hash}(v_k)$, (2) $s_k = \mathsf{encode}_{i_k}(v_k)$, and (3) $\mathsf{valid}(v_k) = \mathit{true}$. By the collision-resistance of $\mathsf{hash}(\cdot)$, for all k, k', we have $v_k = v'_k$. By the definition of $\mathsf{encode}(\cdot)$ and $\mathsf{decode}(\cdot)$, we then have

$$v = \mathsf{decode}(\{s_1, \ldots, s_k\}) = v_1 = \ldots = v_{t+1},$$

which implies that $\mathsf{valid}(v) = \mathit{true}$ and $\mathsf{hash}(v) = h$. ◀

Algorithm description. The pseudocode of DARE-STARK is presented in Algorithm 3 from the perspective of a correct process P_i. Similarly to DARE, DARE-STARK unfolds in three phases:

1. *Dispersal:* Upon proposing a value v_i (line 9), P_i sends (line 14) to each process P_k (1) $(h_k, s_k) = \mathsf{shard}_k(v_i)$ (computed at line 12), and (2) $stark_k = \mathsf{proof}_k(v_i)$ (computed at line 13). In doing so (see Theorem 3), P_i proves to P_k that $h_k = \mathsf{hash}(v_i)$ is the hash of a

Algorithm 3 DARE-Stark: Pseudocode (for process P_i).

```
1:  Uses:
2:      Agreement, instance agreement
3:  upon init:
4:      Hash_Value $proposed\_hash_i \leftarrow \bot$
5:      Map(Hash_Value → (Symbol, STARK)) $proposal\_shards_i \leftarrow$ empty
6:      Map(Hash_Value → Set(Symbol)) $decision\_symbols_i \leftarrow$ empty
7:      Bool $decided_i \leftarrow false$
8:  ▷ Dispersal
9:  upon propose(Value $v_i$):
10:     $proposed\_hash_i \leftarrow \mathsf{hash}(v_i)$
11:     for Integer $k \leftarrow 1$ to $n$:
12:         (Hash $h_k$, Symbol $s_k$) $\leftarrow \mathsf{shard}_k(v_i)$
13:         STARK $stark_k \leftarrow \mathsf{proof}_k(v_i)$
14:         send ⟨DISPERSAL, $h_k, s_k, stark_k$⟩ to $P_k$
15: upon reception of ⟨DISPERSAL, Hash_Value $h$, Symbol $s$, STARK $stark$⟩ from process $P_j$ and $stark$
    proves $\mathsf{shard}_i(?) = (h, s)$:
16:     $proposal\_shards_i[h] \leftarrow (s, stark)$
17:     send ⟨ACK, $\mathsf{share\_sign}_i(h)$⟩ to $P_j$
18: ▷ Agreement
19: upon ⟨ACK, P_Signature $sig$⟩ is received from $2t + 1$ processes:
20:     T_Signature $\Sigma \leftarrow \mathsf{combine}(\{sig \mid sig$ is received in the ACK messages$\})$
21:     invoke $agreement$.propose($proposed\_hash_i, \Sigma$)
22: upon $agreement$.decide(Hash_Value $h$, T_Signature $\Sigma$) with $proposal\_shards_i[h] \neq \bot$:
23:     (Symbol $s$, STARK $stark$) $\leftarrow proposal\_shards_i[h]$
24:     broadcast ⟨RETRIEVE, $h, s, stark$⟩
25: ▷ Retrieval
26: upon reception of ⟨RETRIEVE, Hash_Value $h$, Symbol $s$, STARK $stark$⟩ from process $P_j$ and $stark$
    proves $\mathsf{shard}_j(?) = (h, s)$:
27:     $decision\_symbols_i[h] \leftarrow decision\_symbols_i[h] \cup \{s\}$
28: upon (1) exists Hash_Value $h$ such that $decision\_symbols_i[h]$ has $t + 1$ elements, and (2) $decided_i =$
    $false$:
29:     $decided_i \leftarrow true$
30:     trigger decide(decode($decision\_symbols_i[h]$))
```

valid proposal, whose k-th RS symbol is $\mathsf{encode}_k(v_i)$. P_k checks $stark_k$ against (h_k, s_k) (line 15), stores $(s_k, stark_k)$ (line 16), and sends a partial signature for h_k back to P_i (line 17).

2. *Agreement:* Having collected a threshold signature Σ for $\mathsf{hash}(v_i)$ (line 20), P_i proposes $(\mathsf{hash}(v_i), \Sigma)$ to Agreement (line 21).
3. *Retrieval:* Upon deciding a hash h from Agreement (line 22), P_i broadcasts (if available) the i-th RS symbol for h, along with the relevant proof (line 24). Upon receiving $t + 1$ symbols S for the same hash (line 28), P_i decides $\mathsf{decode}(S)$ (line 30).

Analysis. Upon proposing a value v_i (line 9), a correct process P_i sends $\mathsf{shard}_k(v_i)$ and $\mathsf{proof}_k(v_i)$ to each process P_k (line 14). Checking $\mathsf{proof}_k(v_i)$ against $\mathsf{shard}_k(v_i)$ (line 15), P_k confirms having received the k-th RS symbol for v_i (note that this does not require the transmission of v_i, just $\mathsf{hash}(v_i)$). As $2t + 1$ processes are correct, P_i is guaranteed to eventually gather a $(2t + 1)$-threshold signature Σ for $\mathsf{hash}(v_i)$ (line 20). Upon doing so, P_i proposes $(\mathsf{hash}(v_i), \Sigma)$ to Agreement (line 21). Since every correct process eventually proposes a value to Agreement, every correct process eventually decides some hash h from Agreement (line 22). Because $2t + 1$ processes signed h, at least $t + 1$ correct processes (without loss of generality, $P_1, \ldots, P_{t+1}$) received a correctly encoded RS-symbol for h. More precisely, for every $k \in [1, t + 1]$, P_k received and stored the k-th RS symbol encoded from the pre-image v of h. Upon deciding from AGREEMENT, each process P_k broadcasts its RS symbol, along with the relevant proof (line 24). Because at most t processes are faulty, no correct process receives $t+1$ RS symbols pertaining to a hash other than h. As $P_1, \dots, P_{t+1}$ all broadcast their symbols and proofs, eventually every correct process collects $t+1$ (provably correct) RS symbols S pertaining to h (line 28), and decides $\mathsf{decode}(S)$ (line 30). By Theorem 3, every correct process eventually decides the same valid value v (with $h = \mathsf{hash}(v)$).

Concerning bit complexity, throughout an execution of DARE-STARK, a correct process engages once in AGREEMENT (which exchanges $O(n^2\kappa)$ bits in total) and sends: (1) n DISPERSAL messages, each of size $O(\frac{L}{n} + poly(\kappa))$, (2) n ACK messages, each of size $O(\kappa)$, and (3) n RETRIEVE messages, each of size $O(\frac{L}{n} + poly(\kappa))$. Therefore, the bit complexity of DARE-STARK is $O(nL + n^2 poly(\kappa))$. As for the latency, it is $O(n)$ (due to the linear latency of AGREEMENT).

7 Concluding Remarks

This paper introduces DARE (Disperse, Agree, REtrieve), the first partially synchronous Byzantine agreement algorithm on values of L bits with better than $O(n^2 L)$ bit complexity and sub-exponential latency. DARE achieves $O(n^{1.5}L + n^{2.5}\kappa)$ bit complexity (κ is the security parameter) and optimal $O(n)$ latency, which is an effective $\sqrt{n}$ factor bit-improvement for $L \geq n\kappa$ (typical in practice). DARE achieves its complexity in two steps. First, DARE decomposes problem of agreeing on large values (L bits) into three sub-problems: (1) value dispersal, (2) validated agreement on small values ($O(\kappa)$), and (3) value retrieval. (DARE effectively acts as an extension protocol for Byzantine agreement.) Second, DARE's novel dispersal algorithm solves the main challenge, value dispersal, using only $O(n^{1.5}L)$ bits and linear latency.

Moreover, we prove that the lower bound of $\Omega(nL + n^2)$ is near-tight by matching it near-optimally with DARE-STARK, a modified version of DARE using STARK proofs that reaches $O(nL + n^2 poly(\kappa))$ bits and maintains optimal $O(n)$ latency. We hope DARE-STARK motivates research into more efficient STARK schemes in the future, which currently have large hidden constants affecting their practical use.

References

1 Michael Abd-El-Malek, Gregory R Ganger, Garth R Goodson, Michael K Reiter, and Jay J Wylie. Fault-scalable byzantine fault-tolerant services. *ACM SIGOPS Operating Systems Review*, 39(5):59–74, 2005.

2 Ittai Abraham and Gilad Asharov. Gradecast in synchrony and reliable broadcast in asynchrony with optimal resilience, efficiency, and unconditional security. In Alessia Milani and Philipp Woelfel, editors, *PODC '22: ACM Symposium on Principles of Distributed Computing, Salerno, Italy, July 25 - 29, 2022*, pages 392–398. ACM, 2022. doi:10.1145/3519270.3538451.

3 Ittai Abraham, T-H. Hubert Chan, Danny Dolev, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. Communication Complexity of Byzantine Agreement, Revisited. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, pages 317–326, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3293611.3331629.

4 Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Alexander Spiegelman. Solida: A blockchain protocol based on reconfigurable byzantine consensus. *arXiv preprint arXiv:1612.02916*, 2016.

5 Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Alexander Spiegelman. Solidus: An incentive-compatible cryptocurrency based on permissionless byzantine consensus. *CoRR, abs/1612.02916*, 2016.

6 Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically Optimal Validated Asynchronous Byzantine Agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 337–346, 2019.

7 Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically Optimal Validated Asynchronous Byzantine Agreement. *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, pages 337–346, 2019.

8 Atul Adya, William J Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R Douceur, Jon Howell, Jacob R Lorch, Marvin Theimer, and Roger P Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. *ACM SIGOPS Operating Systems Review*, 36(SI):1–14, 2002.

9 Nicolas Alhaddad, Sourav Das, Sisi Duan, Ling Ren, Mayank Varia, Zhuolun Xiang, and Haibin Zhang. Brief announcement: Asynchronous verifiable information dispersal with near-optimal communication. In Alessia Milani and Philipp Woelfel, editors, *PODC '22: ACM Symposium on Principles of Distributed Computing, Salerno, Italy, July 25 - 29, 2022*, pages 418–420. ACM, 2022. `doi:10.1145/3519270.3538476`.

10 Nicolas Alhaddad, Sisi Duan, Mayank Varia, and Haibin Zhang. Succinct Erasure Coding Proof Systems. *Cryptology ePrint Archive*, 2021.

11 Yair Amir, Claudiu Danilov, Jonathan Kirsch, John Lane, Danny Dolev, Cristina Nita-Rotaru, Josh Olsen, and David Zage. Scaling byzantine fault-tolerant replication towide area networks. In *International Conference on Dependable Systems and Networks (DSN'06)*, pages 105–114. IEEE, 2006.

12 Elli Androulaki, Christian Cachin, Dan Dobre, and Marko Vukolic. Erasure-coded byzantine storage with separate metadata. In Marcos K. Aguilera, Leonardo Querzoni, and Marc Shapiro, editors, *Principles of Distributed Systems - 18th International Conference, OPODIS 2014, Cortina d'Ampezzo, Italy, December 16-19, 2014. Proceedings*, volume 8878 of *Lecture Notes in Computer Science*, pages 76–90. Springer, 2014. `doi:10.1007/978-3-319-14472-6_6`.

13 Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 62–73, 1993.

14 Michael Ben-Or, Ran Canetti, and Oded Goldreich. Asynchronous Secure Computation. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 52–61, 1993.

15 Michael Ben-Or, Boaz Kelmer, and Tal Rabin. Asynchronous Secure Computations with Optimal Resilience. In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, pages 183–192, 1994.

16 Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *Cryptology ePrint Archive*, 2018.

17 Alysson Bessani, João Sousa, and Eduardo EP Alchieri. State Machine Replication for the Masses with BFT-SMART. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362. IEEE, 2014.

18 Amey Bhangale, Chen-Da Liu-Zhang, Julian Loss, and Kartik Nayak. Efficient adaptively-secure byzantine agreement for long messages. In Shweta Agrawal and Dongdai Lin, editors, *Advances in Cryptology - ASIACRYPT 2022 - 28th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, December 5-9, 2022, Proceedings, Part I*, volume 13791 of *Lecture Notes in Computer Science*, pages 504–525. Springer, 2022. `doi:10.1007/978-3-031-22963-3_17`.

19 Richard E Blahut. *Theory and practice of error control codes*, volume 126. Addison-Wesley Reading, 1983.

20 Manuel Bravo, Gregory Chockler, and Alexey Gotsman. Making Byzantine Consensus Live. In *34th International Symposium on Distributed Computing (DISC)*, volume 179, pages 1–17, 2020.

21 Manuel Bravo, Gregory V. Chockler, and Alexey Gotsman. Making byzantine consensus live. *Distributed Comput.*, 35(6):503–532, 2022. doi:10.1007/s00446-022-00432-y.

22 Ethan Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains.* PhD thesis, University of Guelph, 2016.

23 Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. *arXiv preprint arXiv:1807.04938*, pages 1–14, 2018. arXiv:1807.04938.

24 Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.).* Springer, 2011. doi:10.1007/978-3-642-15260-3.

25 Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and Efficient Asynchronous Broadcast Protocols. In Joe Kilian, editor, *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, volume 2139 of *Lecture Notes in Computer Science*, pages 524–541. Springer, 2001. doi:10.1007/3-540-44647-8_31.

26 Christian Cachin and Stefano Tessaro. Asynchronous Verifiable Information Dispersal. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, pages 191–201. IEEE, 2005.

27 Martina Camaioni, Rachid Guerraoui, Matteo Monti, Pierre-Louis Roman, Manuel Vidigueira, and Gauthier Voron. Chop chop: Byzantine atomic broadcast to the network limit. *arXiv preprint arXiv:2304.07081*, 2023.

28 Jan Camenisch, Manu Drijvers, Timo Hanke, Yvonne Anne Pignolet, Victor Shoup, and Dominic Williams. Internet Computer Consensus. *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, 2021:81–91, 2022. doi:10.1145/3519270.3538430.

29 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

30 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

31 Dario Catalano and Dario Fiore. Vector Commitments and Their Applications. In *Public-Key Cryptography–PKC 2013: 16th International Conference on Practice and Theory in Public-Key Cryptography, Nara, Japan, February 26–March 1, 2013. Proceedings 16*, pages 55–72. Springer, 2013.

32 Jing Chen and Silvio Micali. Algorand. *arXiv preprint arXiv:1607.01341*, 2016.

33 Jinyuan Chen. Fundamental limits of byzantine agreement. *CoRR*, abs/2009.10965, 2020. arXiv:2009.10965.

34 Jinyuan Chen. Optimal error-free multi-valued byzantine agreement. In Seth Gilbert, editor, *35th International Symposium on Distributed Computing, DISC 2021, October 4-8, 2021, Freiburg, Germany (Virtual Conference)*, volume 209 of *LIPIcs*, pages 17:1–17:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.DISC.2021.17.

35 Ashish Choudhury and Arpita Patra. On the communication efficiency of statistically secure asynchronous MPC with optimal resilience. *J. Cryptol.*, 36(2):13, 2023. doi:10.1007/s00145-023-09451-9.

36 Pierre Civit, Muhammad Ayaz Dzulfikar, Seth Gilbert, Vincent Gramoli, Rachid Guerraoui, Jovan Komatovic, and Manuel Vidigueira. Byzantine Consensus Is $\Theta(n^2)$: The Dolev-Reischuk Bound Is Tight Even in Partial Synchrony! In Christian Scheideler, editor, *36th International Symposium on Distributed Computing, DISC 2022, October 25-27, 2022, Augusta, Georgia, USA*, volume 246 of *LIPIcs*, pages 14:1–14:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.DISC.2022.14.

37 Pierre Civit, Seth Gilbert, Rachid Guerraoui, Jovan Komatovic, Matteo Monti, and Manuel Vidigueira. Every Bit Counts in Consensus. *arXiv preprint arXiv:2306.00431*, 2023.

38 Pierre Civit, Seth Gilbert, Rachid Guerraoui, Jovan Komatovic, and Manuel Vidigueira. On the Validity of Consensus. *To appear in PODC 2023*, abs/2301.04920, 2023. doi:10.48550/arXiv.2301.04920.

39 Miguel Correia. From byzantine consensus to blockchain consensus. In *Essentials of Blockchain Technology*, pages 41–80. Chapman and Hall/CRC, 2019.

40 Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. From Consensus to Atomic Broadcast: Time-Free Byzantine-Resistant Protocols without Signatures. *The Computer Journal*, 49(1):82–96, 2006.

41 Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. Dbft: Efficient leaderless byzantine consensus and its application to blockchains. In *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, pages 1–8. IEEE, 2018.

42 George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: a dag-based mempool and efficient bft consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 34–50, 2022.

43 Sourav Das, Vinith Krishnan, Irene Miriam Isaac, and Ling Ren. Spurt: Scalable distributed randomness beacon with transparent setup. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, pages 2502–2517. IEEE, 2022. doi:10.1109/SP46214.2022.9833580.

44 Sourav Das, Zhuolun Xiang, and Ling Ren. Asynchronous Data Dissemination and its Applications. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2705–2721, 2021.

45 Sourav Das, Thomas Yurek, Zhuolun Xiang, Andrew Miller, Lefteris Kokoris-Kogias, and Ling Ren. Practical asynchronous distributed key generation. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, pages 2518–2534. IEEE, 2022. doi:10.1109/SP46214.2022.9833584.

46 Christian Decker, Jochen Seidel, and Roger Wattenhofer. Bitcoin Meets Strong Consistency. In *Proceedings of the 17th International Conference on Distributed Computing and Networking*, pages 1–10, 2016.

47 Dan Dobre, Ghassan O. Karame, Wenting Li, Matthias Majuntke, Neeraj Suri, and Marko Vukolic. Proofs of writing for robust storage. *IEEE Trans. Parallel Distributed Syst.*, 30(11):2547–2566, 2019. doi:10.1109/TPDS.2019.2919285.

48 Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for Byzantine agreement. *Journal of the ACM (JACM)*, 1985.

49 Assia Doudou and André Schiper. Muteness Detectors for Consensus with Byzantine Processes. In Brian A. Coan and Yehuda Afek, editors, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC '98, Puerto Vallarta, Mexico, June 28 - July 2, 1998*, page 315. ACM, 1998. doi:10.1145/277697.277772.

50 Sisi Duan, Xin Wang, and Haibin Zhang. Practical Signature-Free Asynchronous Common Subset in Constant Time. *Cryptology ePrint Archive*, 2023.

51 Sisi Duan, Haibin Zhang, and Boxin Zhao. Waterbear: Information-theoretic asynchronous BFT made practical. *IACR Cryptol. ePrint Arch.*, page 21, 2022. URL: https://eprint.iacr.org/2022/021.

52 Cynthia Dwork, Lynch Nancy, and Larry Stockmeyer. Consensus in the Presence of Partial Synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

53 Facebook. Winterfell: A STARK prover and verifier for arbitrary computations. URL: https://github.com/facebook/winterfell#Performance.

54 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the Association for Computing Machinery,*, 32(2):374–382, 1985.

55 Adam Gągol, Damian Leśniak, Damian Straszak, and Michał Świętek. ALEPH: Efficient atomic broadcast in asynchronous networks with Byzantine nodes. *AFT 2019 - Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pages 214–228, 2019. doi:10.1145/3318041.3355467.

56 Chaya Ganesh and Arpita Patra. Broadcast extensions with optimal communication and round complexity. In George Giakkoupis, editor, *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, pages 371–380. ACM, 2016. doi:10.1145/2933057.2933082.

57 Chaya Ganesh and Arpita Patra. Optimal extension protocols for byzantine broadcast and agreement. *IACR Cryptol. ePrint Arch.*, page 63, 2017. URL: `http://eprint.iacr.org/2017/063`.

58 Chaya Ganesh and Arpita Patra. Optimal extension protocols for byzantine broadcast and agreement. *Distributed Comput.*, 34(1):59–77, 2021. `doi:10.1007/s00446-020-00384-1`.

59 James Hendricks, Gregory R. Ganger, and Michael K. Reiter. Low-overhead byzantine fault-tolerant storage. In Thomas C. Bressoud and M. Frans Kaashoek, editors, *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, pages 73–86. ACM, 2007. `doi:10.1145/1294261.1294269`.

60 James Hendricks, Gregory R. Ganger, and Michael K. Reiter. Verifying distributed erasure-coded data. In Indranil Gupta and Roger Wattenhofer, editors, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC 2007, Portland, Oregon, USA, August 12-15, 2007*, pages 139–146. ACM, 2007. `doi:10.1145/1281100.1281122`.

61 Ioannis Kaklamanis, Lei Yang, and Mohammad Alizadeh. Poster: Coded broadcast for scalable leader-based bft consensus. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 3375–3377, 2022.

62 Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is DAG. In Avery Miller, Keren Censor-Hillel, and Janne H. Korhonen, editors, *PODC '21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021*, pages 165–175. ACM, 2021. `doi:10.1145/3465084.3467905`.

63 Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 45–58, 2007.

64 Ramakrishna Kotla and Michael Dahlin. High throughput byzantine fault tolerance. In *International Conference on Dependable Systems and Networks, 2004*, pages 575–584. IEEE, 2004.

65 Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

66 Andrew Lewis-Pye. Quadratic worst-case message complexity for State Machine Replication in the partial synchrony model, 2022. `doi:10.48550/ARXIV.2201.01107`.

67 Fan Li and Jinyuan Chen. Communication-efficient signature-free asynchronous byzantine agreement. In *IEEE International Symposium on Information Theory, ISIT 2021, Melbourne, Australia, July 12-20, 2021*, pages 2864–2869. IEEE, 2021. `doi:10.1109/ISIT45174.2021.9518010`.

68 Benoît Libert, Marc Joye, and Moti Yung. Born and Raised Distributively: Fully Distributed Non-Interactive Adaptively-Secure Threshold Signatures with Short Shares. *Theoretical Computer Science*, 645:1–24, 2016.

69 Yuan Lu, Zhenliang Lu, Qiang Tang, and Guiling Wang. Dumbo-MVBA: Optimal Multi-Valued Validated Asynchronous Byzantine Agreement, Revisited. *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, pages 129–138, 2020.

70 Loi Luu, Viswesh Narayanan, Kunal Baweja, Chaodong Zheng, Seth Gilbert, and Prateek Saxena. Scp: A computationally-scalable byzantine consensus protocol for blockchains. *Cryptology ePrint Archive*, 2015.

71 Dahlia Malkhi, Kartik Nayak, and Ling Ren. Flexible byzantine fault tolerance. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, pages 1041–1053, 2019.

72 Ralph C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology — CRYPTO*, 1987.

73 Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 31–42, 2016.

74 Atsuki Momose and Ling Ren. Multi-threshold byzantine fault tolerance. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1686–1699, 2021.

75 Atsuki Momose and Ling Ren. Optimal Communication Complexity of Authenticated Byzantine Agreement. In *35th International Symposium on Distributed Computing (DISC)*, volume 209, pages 32:1–32:0. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany, 2021.

76 Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. Signature-Free Asynchronous Binary Byzantine Consensus with t < n/3, O(n2) Messages, and O(1) Expected Time. *J. ACM*, 62(4):31:1–31:21, 2015. `doi:10.1145/2785953`.

77 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized business review*, page 21260, 2008.

78 Oded Naor and Idit Keidar. Expected Linear Round Synchronization: The Missing Link for Linear Byzantine SMR. *34th International Symposium on Distributed Computing (DISC)*, 179, 2020.

79 Kartik Nayak, Ling Ren, Elaine Shi, Nitin H. Vaidya, and Zhuolun Xiang. Improved extension protocols for byzantine broadcast and agreement. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, volume 179 of *LIPIcs*, pages 28:1–28:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. `doi:10.4230/LIPIcs.DISC.2020.28`.

80 Nuno Ferreira Neves, Miguel Correia, and Paulo Verissimo. Solving Vector Consensus with a Wormhole. *IEEE Transactions on Parallel and Distributed Systems*, 16(12):1120–1131, 2005.

81 Irving S Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

82 Irving S Reed and Gustave Solomon. Polynomial odes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

83 Victor Shoup. Practical Threshold Signatures. In Bart Preneel, editor, *Advances in Cryptology - EUROCRYPT 2000, International Conference on the Theory and Application of Cryptographic Techniques, Bruges, Belgium, May 14-18, 2000, Proceeding*, volume 1807 of *Lecture Notes in Computer Science*, pages 207–220. Springer, 2000. `doi:10.1007/3-540-45539-6_15`.

84 Victor Shoup and Nigel P Smart. Lightweight asynchronous verifiable secret sharing with optimal resilience. *Cryptology ePrint Archive*, 2023.

85 Alexander Spiegelman. In Search for an Optimal Authenticated Byzantine Agreement. In Seth Gilbert, editor, *35th International Symposium on Distributed Computing (DISC 2021)*, volume 209 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 38:1–38:19, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.DISC.2021.38`.

86 Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: Dag bft protocols made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2705–2718, 2022.

87 Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. Efficient byzantine fault-tolerance. *IEEE Transactions on Computers*, 62(1):16–30, 2011.

88 Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

89 Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT Consensus with Linearity and Responsiveness. *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.

90 Thomas Yurek, Licheng Luo, Jaiden Fairoze, Aniket Kate, and Andrew Miller. hbacss: How to robustly share many secrets. In *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society, 2022. URL: `https://www.ndss-symposium.org/ndss-paper/auto-draft-245/`.

A Sync: Implementation

In this section, we provide the remaining pseudocode of DISPERSER, namely the SYNC component. For a complete formal proof of the correctness and complexity of DISPERSER, please refer to the extended version of this paper. Throughout the entire section, $X = Y = \sqrt{n}$. For simplicity, we assume that $\sqrt{n}$ is an integer.

Algorithm description. The pseudocode of SYNC is given in Algorithm 4, and it highly resembles RARESYNC [36]. Each process P_i has an access to two timers: (1) $view\_timer_i$, and (2) $dissemination\_timer_i$. A timer exposes the following interface:

- measure(Time x): After exactly x time as measured by the local clock, the timer expires (i.e., an expiration event is received by the host). As local clocks can drift before GST, timers might not be precise before GST: x time as measured by the local clock may not amount to x real time.
- cancel(): All previously invoked measure($\cdot$) methods (on that timer) are cancelled. Namely, all pending expiration events (associated with that timer) are removed from the event queue.

We now explain how SYNC works, and we do so from the perspective of a correct process P_i. When P_i starts executing SYNC (line 11), it starts measuring $view\_duration = \Delta + 2\delta$ time using $view\_timer_i$ (line 12) and enters the first view (line 13).

Once $view\_timer_i$ expires (line 14), which signals that P_i's current view has finished, P_i notifies every process of this via a VIEW-COMPLETED message (line 15). When P_i learns that $2t + 1$ processes have completed some view $V \geq view_i$ (line 16) or any process started some view $V' > view_i$ (line 22), P_i prepares to enter a new view (either $V + 1$ or V'). Namely, P_i (1) cancels $view\_timer_i$ (line 19 or line 25), (2) cancels $dissemination\_timer_i$ (line 20 or line 26), and (3) starts measuring δ time using $dissemination\_timer_i$ (line 21 or line 27). Importantly, P_i measures δ time (using $dissemination\_timer_i$) before entering a new view in order to ensure that P_i enters only $O(1)$ views during the time period $[\text{GST}, \text{GST} + 3\delta)$. Finally, once $dissemination\_timer_i$ expires (line 28), P_i enters a new view (line 31).

B STARKs

First introduced in [16], STARKs are succinct, universal, transparent arguments of knowledge. For any function f (computable in polynomial time) and any (polynomially-sized) y, a STARK can be used to prove the knowledge of some x such that $f(x) = y$. Remarkably, the size of a STARK proof is $O(poly(\kappa))$. At a very high level, a STARK proof is produced as follows: (1) the computation of $f(x)$ is unfolded on an execution trace; (2) the execution trace is (RS) over-sampled for error amplification; (3) the correct computation of f is expressed as a set of algebraic constraints over the trace symbols; (4) the trace symbols are organized in a Merkle tree [72]; (5) the tree's root is used as a seed to pseudo-randomly sample the trace symbols. The resulting collection of Merkle proofs proves that, for some known (but not revealed) x, $f(x) \neq y$ only with cryptographically low probability (negligible in κ). STARKs are non-interactive, require no trusted setup (they are transparent), and their security reduces to that of cryptographic hashes in the Random Oracle Model (ROM) [13].

C Further Analysis of DARE

In this section, we provide a brief good-case analysis of DARE and discuss how DARE can be adapted to a model with unknown δ.

Algorithm 4 Sync: Pseudocode (for process P_i).

```
1:  Uses:
2:      Timer $view\_timer_i$
3:      Timer $dissemination\_timer_i$
4:  Functions:
5:      leaders(View $V$) $\equiv \{P_{\left((V \bmod \sqrt{n})-1\right)\sqrt{n}+1}, P_{\left((V \bmod \sqrt{n})-1\right)\sqrt{n}+2}, ..., P_{\left((V \bmod \sqrt{n})-1\right)\sqrt{n}+\sqrt{n}}\}$
6:  Constants:
7:      Time $view\_duration = \Delta + 2\delta = \delta\sqrt{n} + 3\delta + 2\delta$
8:  Variables:
9:      View $view_i \leftarrow 1$
10:     T_Signature $view\_sig_i \leftarrow \bot$
11: upon init:                                              ▷ start of the algorithm
12:     $view\_timer_i$.measure($view\_duration$)
13:     trigger advance(1)
14: upon $view\_timer_i$ expires:
15:     broadcast ⟨VIEW-COMPLETED, $view_i$, share_sign$_i$($view_i$)⟩
16: upon exists View $V \geq view_i$ with $2t+1$ ⟨VIEW-COMPLETED, $V$, P_Signature $sig$⟩ received messages:
17:     $view\_sig_i \leftarrow$ combine$\big(\{sig \mid sig$ is received in the VIEW-COMPLETED messages$\}\big)$
18:     $view_i \leftarrow V + 1$
19:     $view\_timer_i$.cancel()
20:     $dissemination\_timer_i$.cancel()
21:     $dissemination\_timer_i$.measure($\delta$)
22: upon reception of ⟨ENTER-VIEW, View $V$, T_Signature $sig$⟩ with $V > view_i$:
23:     $view\_sig_i \leftarrow sig$
24:     $view_i \leftarrow V$
25:     $view\_timer_i$.cancel()
26:     $dissemination\_timer_i$.cancel()
27:     $dissemination\_timer_i$.measure($\delta$)
28: upon $dissemination\_timer_i$ expires:
29:     broadcast ⟨ENTER-VIEW, $view_i$, $view\_sig_i$⟩
30:     $view\_timer_i$.measure($view\_duration$)
31:     trigger advance($view_i$)
```

C.1 Good-Case Complexity

For the good-case complexity, we consider only executions where GST $= 0$ and where all processes behave correctly. This is sometimes also regarded as the common case since, in practice, there are usually no failures and the network behaves synchronously. Throughout the entire subsection, $X = Y = \sqrt{n}$.

In such a scenario, the good-case bit complexity of DARE is $O(n^{1.5}L + n^2\kappa)$. As all processes are correct and synchronized at the starting view, Disperser terminates after only one view. The $n^{1.5}L$ term comes from this view: the first $\sqrt{n}$ correct leaders broadcast their full L-bit proposal to all other processes. The $n^{2.5}\kappa$ term is reduced to only $n^2\kappa$ (only the CONFIRM messages sent by correct processes at line 29) since Disperser terminates after just one view.

The good-case latency of DARE is essentially the sum of the good-case latencies of the Dispersal, Agreement, and Retrieval phases:

$$\underbrace{O(\sqrt{n}\cdot\delta)}_{\text{DISPERSAL}} + \underbrace{O(\delta)}_{\text{AGREEMENT}} + \underbrace{O(\delta)}_{\text{RETRIEVAL}} = O(\sqrt{n}\cdot\delta).$$

Thus, the good-case latency of DARE is $O(\sqrt{n}\cdot\delta)$.

C.2 DARE (and DARE-Stark) with Unknown δ

To accommodate for unknown δ, two modifications to DARE are required:

- DISPERSER must accommodate for unknown δ. We can achieve this by having SYNC increase the ensured overlap with every new view (by increasing *view_duration* for every new view).
- AGREEMENT must accommodate for unknown δ. Using the same strategy as for SYNC, AGREEMENT can tolerate unknown δ. (The same modification makes DARE-STARK resilient to unknown δ.)

Efficient Collaborative Tree Exploration with Breadth-First Depth-Next

Romain Cosson
Inria, Paris, France

Laurent Massoulié
Inria, Paris, France

Laurent Viennot
Inria, Paris, France

Abstract

We study the problem of *collaborative tree exploration* introduced by Fraigniaud, Gasieniec, Kowalski, and Pelc [10] where a team of k agents is tasked to collectively go through all the edges of an unknown tree as fast as possible and return to the root. Denoting by n the total number of nodes and by D the tree depth, the $\mathcal{O}(n/\log(k) + D)$ algorithm of [10] achieves a $\mathcal{O}(k/\log(k))$ competitive ratio with respect to the cost of offline exploration which is at least $\max\{2n/k, 2D\}$. Brass, Cabrera-Mora, Gasparri, and Xiao [1] study an alternative performance criterion, the competitive overhead with respect to the cost of offline exploration, with their $2n/k + \mathcal{O}((D+k)^k)$ guarantee. In this paper, we introduce "Breadth-First Depth-Next" (BFDN), a novel and simple algorithm that performs collaborative tree exploration in $2n/k + \mathcal{O}(D^2 \log(k))$ rounds, thus outperforming [1] for all values of (n, D, k) and being order-optimal for trees of depth $D = o(\sqrt{n})$. Our analysis relies on a two-player game reflecting a problem of online resource allocation that could be of independent interest. We extend the guarantees of BFDN to: scenarios with limited memory and communication, adversarial setups where robots can be blocked, and exploration of classes of non-tree graphs. Finally, we provide a recursive version of BFDN with a runtime of $\mathcal{O}_\ell(n/k^{1/\ell} + \log(k)D^{1+1/\ell})$ for parameter $\ell \geq 1$, thereby improving performance for trees with large depth.

2012 ACM Subject Classification Theory of computation → Online algorithms; Theory of computation → Distributed algorithms; Mathematics of computing → Graph algorithms

Keywords and phrases collaborative exploration, online algorithms, trees, adversarial game, competitive analysis, robot swarms

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.14

Funding This work was supported by PRAIRIE ANR-19-P3IA-0001 and Tempogral ANR-22-CE48-0001.

Acknowledgements The authors thank the anonymous reviewers for their useful remarks and the entire Argo team at Inria for enlightening discussions. RC thanks Pierre Fraigniaud for precious advice and Maxime Cartan for his implementation of a Python demo (available at `https://github.com/Romcos/BFDN`). A brief announcement appeared in PODC' 23 [4] and some extensions are available on arXiv [3].

1 Introduction

Problem setting. A team of robots[1], initially located at the root of an unknown tree, is tasked to collectively go through all the edges of a tree as fast as possible and then return to the root. At each round, the robots move synchronously along one incident edge to reach a neighbour, thereby discovering new adjacent edges. Following [10], we consider two distinct

[1] the term "robots" is often preferred over "agents" in line with the initial work of [10].

communication models. The *complete communication* model, in which communications are unrestricted and consequently the team takes decisions in a centralized fashion. The *write-read* communication model, in which robots communicate through whiteboards that are located at all nodes and must thus take decisions in a distributed fashion.

Main results. In this paper, we present a simple and novel algorithm that achieves collaborative tree exploration with k agents in $\frac{2n}{k} + D^2(\min\{\log(k), \log(\Delta)\} + 3)$ rounds for any tree with n nodes, depth D and maximum degree Δ. This algorithm can be implemented in the complete communication model and the write-read communication model.

The algorithm is called "Breadth-First Depth-Next" (abbreviated `BFDN`) and the behaviour of the robots can be described synthetically as follows: when located at the root, a robot is sent to the highest unexplored edge (as in a breadth-first search). Upon arrival, the robot changes behaviour until it reaches the root again, it goes through unexplored edges when adjacent to one and goes up towards the root otherwise (as in a depth-first search).

Our analysis involves a simple zero-sum two-player game with balls in urns. An immediate application of this analysis is in resource allocation in the face of uncertainty. Given k workers and k (parallelizable) tasks requiring each an unknown amount of work, we show that the strategy of reassigning idle workers to the least crowded task is competitive in terms of number of times a worker will have to switch between tasks. More precisely, we show that this number is at most $k\log(k) + 2k$.

The `BFDN` algorithm is easy to implement and we provide it with extensions to more complex settings, such as i) exploration of specific classes of non-tree graphs, ii) scenarios with constrained communications and memory, and iii) setups where an adversary chooses at each time step which robots are allowed to move. Finally, in an attempt to improve dependence in the tree depth D, we propose $\texttt{BFDN}_\ell$, a recursive version of `BFDN` in the complete communication model that explores the tree in time $\mathcal{O}_\ell\left(\frac{n}{k^{1/\ell}} + \min\{\log(k), \log(\Delta)\}D^{1+1/\ell}\right)$ where $\ell \geq 1$ is some constant provided as input.

Useful context and related works. In the case of a single robot, the "Depth First Search" (`DFS`) algorithm is optimal for traversing the edges of a tree. It can be implemented both *offline* (the tree is known in advance) and *online* (edges are revealed when reached). One way to describe DFS in an online fashion is to have the robot go through an adjacent unexplored edge if possible and go up towards the root otherwise. After $2(n-1)$ rounds, where n is the number of nodes, all edges have been traversed (twice) and the robot is at the root.

In the multi-robot setting, i.e. with $k \geq 2$, traversing all the edges of a tree in an *offline* manner requires at least $\max\{2n/k, 2D\} \geq n/k + D$ synchronous rounds [7, 13]. This is because every edge has to be traversed in both directions and some robot has to reach the deepest node before returning to the root. A simple algorithm [7, 13] matches this bound up to a factor 2, with a runtime of at most $2(n/k + D)$: consider a depth-first search path from the root of length $2(n-1)$, and divide it in k segments each of length $\lceil 2(n-1)/k \rceil$, then assign one robot to reach and traverse each segment. The optimal offline k-traversal is NP-hard to compute as [10] gave a reduction from `3-PARTITION` to this problem.

To analyze the *online* problem (i.e. collective tree exploration), the literature initially focused on the *competitive ratio* which is the worst-case ratio between the cost an online algorithm and the optimal offline algorithm. For an online algorithm $\mathcal{A}_k$ using $k \geq 2$ robots, this ratio is defined up to a constant factor as $\max_{n,D\in\mathbb{N}} \max_{T\in\mathcal{T}(n,D)} \texttt{Runtime}(\mathcal{A}_k, T)/(n/k + D)$ where $\mathcal{T}(n, D)$ denotes the set of all trees with n nodes and depth D. The algorithm proposed initially by [10] `CTE` (Collective Tree Exploration) runs in $O(\frac{n}{\log k} + D)$ rounds for any tree $T \in \mathcal{T}(n, D)$ and therefore has a competitive ratio of $O(\frac{k}{\log k})$. Furthermore, it can be implemented in the write-read communication model [10]. It was later shown by [11] that the competitive analysis of `CTE` is tight as they provided a simple construction of a tree with $n = kD$ edges that `CTE` would take $\frac{Dk}{\log_2(k)}$ time-steps to explore. To date, no algorithm is known to have a better competitive ratio than `CTE`, while the best lower-bound known on the competitive ratio, for deterministic exploration algorithms, is in $\Omega(\frac{\log k}{\log \log k})$ by [9].

The limited progress on the analysis of the competitive ratio as a function of k led most subsequent works to investigate algorithms with super-linear dependence in (n, D), usually assuming complete communication [13, 1, 8, 6, 5, 11]. In this spirit, [13] derived a recursive algorithm called `Yo*` that runs in $\mathcal{O}(2^{\mathcal{O}(\sqrt{\log D \log \log k})} \log(k)(\log(k) + \log(n))(n/k + D))$ rounds. On the other hand, [1] proposed a novel analysis of `CTE` yielding a guarantee of $\frac{2n}{k} + \mathcal{O}((k+D)^k)$, displaying optimal dependence in n at the cost of large additive dependence in (k, D). The algorithm we propose with its guarantee of $\frac{2n}{k} + \mathcal{O}(D^2 \log(k))$ complements this line of work. Our guarantee yields a strict improvement over [1] for all values of (n, k, D), and improves upon `CTE` and `Yo*` for the specific range of parameters as depicted in Figure 1.

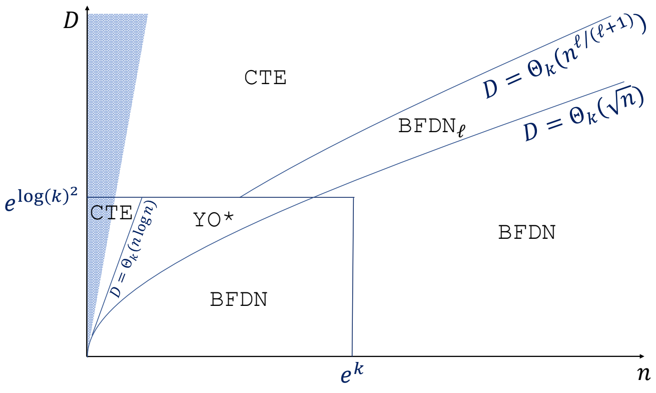


Figure 1 Regions of (n, D) where either of `CTE`, `Yo`*, `BFDN` and `BFDN`$_\ell$ has the best runtime guarantee. The runtime of algorithm Yo* was simplified to improve readability. ℓ must satisfy $\ell \leq \text{cst}(\log k / \log \log k)$. No trees defined in shaded region $n \leq D$. See Appendix A for details.

Collaborative tree exploration has also been studied under additional assumptions. For example, for trees which can be embedded in the 2-dimensional grid, [8] obtained an algorithm running in $\mathcal{O}(\sqrt{D}(\frac{n}{k} + D))$ rounds. The setting where the number of robots k is very large, specifically $k \geq Dn^c$ for some constant $c > 1$, was also investigated by [5]. Assuming global communication, their algorithm achieves exploration in $\frac{c}{c-1}D + o(D)$ rounds. Interestingly, their guarantees also apply to the challenging and less studied collaborative graph exploration problem; see also [1, 2].

Open directions. In line with [1], our work advocates for the study of the *competitive overhead* of collaborative exploration in complement to its *competitive ratio*. Recently [6] showed that (deterministic) collaborative exploration with $k = n$ requires at least $\Omega(D^2)$, implying that no algorithm can have a $\frac{2n}{k} + \mathcal{O}(D^c)$ guarantee for $c < 2$. On the other hand, a simple algorithm explores any tree in $\mathcal{O}(D^2)$ rounds as soon as $k \geq \frac{n}{D}$ [13]. In view of these results, our $\frac{2n}{k} + \mathcal{O}(D^2 \log(k))$ guarantee seems close-to-optimal. We highlight the open question of whether there exists a $\frac{2n}{k} + \mathcal{O}(D^2)$ exploration algorithm, or even a guarantee of the form $\frac{2n}{k} + \mathcal{O}(f(D))$, for some real-valued function f.

Structure of the paper. Section 2 defines algorithm BFDN and provides the main result for the complete communication setting. Section 3 analyzes a two-player zero-sum board game, an essential ingredient in our analysis of BFDN. Section 4 contains extensions of BFDN to settings with: limited communications; adversarial interruption of robots; and more general graph exploration. Finally, Section 5 provides a recursive version of BFDN that yields improved runtime guarantees when the tree depth D gets larger compared to n.

Notations. $\log(\cdot)$ refers to the natural logarithm and $\log_2(\cdot)$ to the logarithm in base 2. For an integer k we use the abbreviation $[k] = \{1, \dots, k\}$.

A tree $T = (V, E)$ is defined by its set of nodes V and edges $E \subset V \times V$; it is rooted at some specific node denoted $\texttt{root} \in V$ from which all robots start the exploration. For a node $v \in V$, $\delta(v)$ is the distance of v to the $\texttt{root}$ and $T(v)$ denotes the sub-tree of T rooted at v containing all the descendants of v. The depth of T is $D = \max_{v \in V} \delta(v)$. We will also use a notion of *partially explored tree* (defined in Section 2) that enjoys the same definitions.

2 The Breadth-First Depth-Next algorithm

Our main result on BFDN, which is described below, is the following

▶ **Theorem 1.** BFDN *achieves online exploration of any tree with k robots in at most*

$$\frac{2n}{k} + D^2(\min\{\log(\Delta), \log(k)\} + 3)$$

rounds, where Δ is the maximum degree, n is the number of nodes, and D is the depth.

Following [10], we shall start by showing the guarantee in the complete communication model, and we later present in Section 4 how BFDN can be adapted to the write-read model.

Partially explored tree. At a given exploration round, V denotes the set of *explored nodes*, i.e. nodes that have been occupied by at least one robot in the past, and E denotes the set of *discovered edges*, i.e. edges that have at least one explored endpoint. The discovered edges that have exactly one explored endpoint are called *dangling edges*. Such edges can be viewed as a pair $(u, ?)$, with $u \in V$. The *partially explored tree* or *discovered tree* $T_{\text{online}} = (V, E)$ contains all the information gathered by the robots at some point of exploration. If there are no dangling edges in T_{online}, it means that exploration is complete and that the partially explored tree equals the underlying tree $T_{\text{offline}} \in \mathcal{T}(n, D)$.

Collaborative exploration algorithm. A collaborative exploration algorithm in the complete communication model is formally defined as a function that maps a partially explored tree $T = (V, E)$ as well as the list of positions of the agents $p_1, \dots, p_k \in V^k$ and their past movements to a list of *selected edges* $e_1, \dots, e_k \in (E \cup \{\perp\})^k$ that the agents will use for their next move. Each selected edge $e_i \in E$ must be adjacent to the position p_i. Dangling edges may be selected. By convention, $e_i =\perp$ is used to indicate that agent i will not move at the next round. In pseudo-code, the routine $\texttt{SELECT}(\texttt{Robot}_i, e)$ performs the assignment $e_i \leftarrow e$. When all agents have selected a next move, the routine MOVE is applied and all agents move along their selected edge synchronously. The partially explored tree (V, E) is then updated with the new information provided by the agents that have traversed a dangling edge. Exploration always starts with all agents located at the root, $V = \{\texttt{root}\}$ and E the set of all dangling edges that are adjacent to the root. The collaborative exploration algorithm is applied iteratively. Exploration terminates when the explored tree (V, E) contains no dangling edges and when the position of all agents is back at the root. The runtime of an exploration algorithm is defined as a function of (n, D) by the number of rounds required before termination on any tree with n nodes and depth D.

Breadth-First Depth-Next Algorithm. We now provide a brief description of `BFDN`, Algorithm 1. When located at the root, a robot indexed by $i \in [k]$ and denoted $\texttt{Robot}_i$ is assigned an *anchor* $v_i \in V$ which is a node that is adjacent to at least one dangling edge. If no such node exists, the anchor is the root itself. The exact anchor assignment is specified by procedure `Reanchor` which gives the priority to nodes that are the closest to the root and that have the least number of anchored robots. $\texttt{Robot}_i$ then attains this anchor in a series of breadth-first moves performed with procedure `BF`. When the anchor is reached, the robot only makes depth-next moves with procedure `DN`, until it returns to the root. In a sequence of depth-next moves, the robot always goes through a dangling edge if one is available (i.e. adjacent and not already selected as next move by another robot), and goes one step up towards the root otherwise. This will result in a depth-first-like exploration inside $T(v_i)$ followed by a direct travel from v_i to the root. The algorithm stops when all robots are at the root and are not assigned a new anchor because there are no more dangling edges.

The reason why we ask that the robots go back all the way to the root before being reassigned a new anchor, rather than having them use a shortest path from their previous anchor to their next anchor, will become apparent when we adapt the algorithm to the distributed write-read communication setting. In that setting, the root will play the role of a central planner, gathering information on the advancement of exploration thanks to returning robots.

2.1 Analysis of `BFDN` and proof of Theorem 1

We first prove the correctness and termination of `BFDN` and then bound its runtime.

Correctness. In Algorithm 1, the do-while loop is interrupted when no robot changes position at some round (line 14). Note that the `root` is the only place where robots may stay at the same position because direction `up` is interpreted as $\perp$ at the root only (line 23). Thus all robots are at the root when the algorithm stops. Also note that the selection of direction `up` by all robots at the root implies that there are no dangling edges in the tree. Thus the tree has been entirely explored and all robots have returned. The algorithm is correct.

Termination. To prove termination, we show that while the algorithm runs, a node is discovered every $3D$ rounds at least. Since there are n nodes in the tree, the algorithm must terminate after at most $3D \times n$ rounds. Assume by contradiction that no node is discovered in a sequence of $3D$ rounds. After $2D$ rounds, all robots have attained the root because all `DF` moves are directed `up`. Then, either one robot is assigned an anchor that is adjacent to an unexplored edge which will be traversed in the coming D rounds, or the algorithm stops. In both cases we have a contradiction.

We now provide the following lemma which will be proved in Section 3.

▶ **Lemma 2.** *In an execution of* `BFDN`*, for any* $d \in \{1, \ldots, D-1\}$*, the number of calls to procedure* `Reanchor` *which return an anchor at depth* d *is at most* $k(\min\{\log(k), \log(\Delta)\} + 3)$.

Algorithm 1 BFDN "Breadth-First Depth-Next".

```
Ensure: The robots traverse all edges and return to the root.
 1: $V$ = list of explored nodes ; $E$ = list of discovered edges
 2: $v_i \leftarrow$ root  $\forall i \in \{1, \dots, k\}$                      ▷ Initialize anchors.
 3: $S_i \leftarrow [\,]$  $\forall i \in \{1, \dots, k\}$                      ▷ Initialize empty stacks.
 4: do                                                                          ▷ Round $t$.
 5:     for $i = 1$ to $k$ do                                                   ▷ Sequential decisions.
 6:         if Robot$_i$ is at root then
 7:             $v_i \leftarrow$ Reanchor$(i)$
 8:             Stack in $S_i$ the list of edges that lead to $v_i$             ▷ Reverse order.
 9:         if $S_i$ is not empty then
10:             BF$(i)$
11:         else
12:             DN$(i)$
13:     MOVE all robots on their selected edge and update $(V, E)$      ▷ Synchronous moves.
14: while some robot changes position
15:
16: procedure BF$(i)$
17:     Unstack $e \in E$ from $S_i$ and SELECT(Robot$_i, e$)
18:
19: procedure DN$(i)$
20:     if Robot$_i$ is adjacent to some dangling and unselected edge $e \in E$ then
21:         SELECT(Robot$_i, e$)
22:     else
23:         SELECT(Robot$_i$, up)              ▷ If Robot$_i$ is at the root, up is interpreted as $\perp$.
24:
25: procedure REANCHOR$(i)$
26:     $U = \{v \in V \;\; s.t. \;\; v$ is adjacent to some dangling edge with $\delta(v)$ minimal$\}$
27:     if $U \neq \emptyset$ then                                     ▷ Choose anchor of minimum load.
28:         $v_i \leftarrow \arg\min_{v \in U} n_v$  where  $\forall v \in V : n_v = \#\{j \in [k] \;\; s.t. \;\; v_j = v\}$
29:     else                                                                    ▷ The tree is explored.
30:         $v_i \leftarrow$ root
```

Time complexity. During the execution, a given Robot$_i$ anchored at v_i can spend time in two different ways (1) being idle at the root (2) moving along a selected edge. We denote by $\mathsf{T}_i^1, \mathsf{T}_i^2$ the time (number of rounds) spent by Robot$_i$ in each of these phases. We have that $\sum_{i \in [k]} (\mathsf{T}_i^1 + \mathsf{T}_i^2) = k\mathsf{T}$ where T is the total number of rounds of the algorithm as the k robots operate in parallel. We now prove a series of claims.

▶ **Claim 1.** *The total number of rounds when some robot does not move is at most $D + 1$.*

Proof of Claim 1. Recall that if a robot does not move, it must be anchored at the root and have selected direction up with procedure DN. This only occurs in two cases (1) there are no more dangling edges in the discovered tree (this happens at most D times because all robots are on their way back) (2) there are still dangling edges that are adjacent to the root, but they are all selected (this happens at most once because at the next time-step, all edges adjacent to the root will be explored). The number of time-steps when a robot may not move is thus at most $D + 1$. ◀

▶ **Claim 2.** *In the round when a dangling edge is explored for the first time, it is traversed by a single robot.*

Proof of Claim 2: All breadth-first moves (with procedure BF) are through previously explored edges because they lead from the root to a previously explored node. Thus dangling edges are only explored in depth-next moves (with procedure DN). In this procedure, two robots cannot select the same dangling edge. ◀

▶ **Claim 3.** *Consider a sequence of moves by some* $\texttt{Robot}_i$ *that starts at the root with the assignment of an anchor* v *of depth* $\delta(v) = d$ *and that ends with the return of* $\texttt{Robot}_i$ *to the root after* T_x *rounds. In this sequence,* $\texttt{Robot}_i$ *explored exactly* $(\mathsf{T}_x - 2d)/2$ *dangling edges.*

Proof of Claim 3. The sequence of moves, denoted x, has the following structure. First, $\texttt{Robot}_i$ uses a shortest path from the $\texttt{root}$ to v which takes d moves through previously explored edges. Then the robot performs moves inside $T(v)$ by going down through dangling edges if some are available and going up towards the root otherwise. Note that exactly half of the moves inside $T(v)$ must be through dangling edges as there must be as many moves down as moves up in $T(v)$. Finally, the robot goes back from v to the root in again d moves through explored edges. Exactly $(\mathsf{T}_x - 2d)/2$ dangling edges are explored in this sequence. ◀

We now assemble the claims and Lemma 2 together to bound the runtime of BFDN. Using Claim 1, we have that $\sum_i \mathsf{T}_i^1 \le k(D+1)$. Then, we write $\sum_i \mathsf{T}_i^2 = \sum_{d\le D-1}\sum_{x\in X_d} \mathsf{T}_x$ where X_d is the list of all sequences of moves x that start with the assignment of an anchor v at depth $\delta(v) = d$ to some robot and that end with the return of that robot to the root. Using Claim 2 and Claim 3, we have that $\sum_{d\le D-1}\sum_{x\in X_d}(\mathsf{T}_x - 2d)/2 \le n-1$. Consequently,

$$\sum_{i\in[k]} \mathsf{T}_i^2 \le 2(n-1) + 2\sum_{d\le D-1}\sum_{x\in X_d} d.$$

By Lemma 2, the cardinality of X_d is at most $k(\min\{\log(k), \log(\Delta)\}+3)$, for $d \in \{1, \dots, D-1\}$. Thus, $\sum_{d\le D-1}\sum_{x\in X_d} d \le \frac{D(D-1)}{2}k(\min\{\log(k), \log(\Delta)\}+3)$. Finally, using $\sum_{i\in[k]}(\mathsf{T}_i^1 + \mathsf{T}_i^2) = k\mathsf{T}$, we obtain $k\mathsf{T} \le 2(n-1) + D(D-1)k(\min\{\log(\Delta), \log(k)\}+3) + (D+1)k$, which proves that the algorithm stops after at most

$$\mathsf{T} \le \frac{2n}{k} + D^2(\min\{\log(\Delta), \log(k)\}+3)$$

steps, thus completing Theorem 1's proof.

Though it is not required for the the analysis above, we conclude this section with a final claim that provides useful intuition on the algorithm.

▶ **Claim 4.** *At all rounds, all dangling and unexplored edges, are in* $\cup_{i\in[k]}T(v_i)$.

Proof of Claim 4. Consider some dangling edge e and its explored endpoint $v \in V$. At the round when v was explored by a robot, that robot was performing a depth-next move because its anchor was at least as high as v which is still adjacent to a dangling edge. That robot cannot have left $T(v)$ before the edge e was traversed. Consequently, it is still rooted at some ancestor v_i of v, thus $e \in \cup_{i\in[k]}T(v_i)$. ◀

3 A two-player zero-sum game with balls in urns

In this section we introduce a two-player zero-sum board game that essential to the analysis of BFDN. A strategy for the player of the game is given and analyzed in Theorem 3. Its connection with BFDN is detailed in Section 3.2 where a proof of Lemma 2 is given.

3.1 Game of balls in urns

Game description. At time $t \in \mathbb{N}$, the board of the game is a list of k integers $(n_1^t, \dots, n_k^t)$ that represent the load of k urns with a total of k balls. When the game starts at $t = 0$, we have $n_i^0 = 1$ and at every instant t we have $\sum_{i \in [k]} n_i^t = k$ and $n_i^t \geq 0$. At time t, player A (the adversary) chooses a ball in an urn $a_t \in [k]$ that is not empty, i.e. such that $n_{a_t}^t \geq 1$, and then player B (the player) chooses an urn $b_t \in [k]$ and moves that ball from urn a_t to urn b_t. At the beginning of time $t+1$, the board satisfies $n_{a_t}^{t+1} = n_{a_t}^t - 1$ and $n_{b_t}^{t+1} = n_{b_t}^t + 1$.

Goal of the game. At a given time t, we denote by U_t the set of urns that have never been selected by the adversary, $U_t = \{1, \dots, k\} \setminus \{a_0, \dots, a_{t-1}\}$. The game stops when all urns in U_t contain at least Δ balls, i.e. $n_i^t \geq \Delta, \forall i \in U_t$. If $\Delta \geq k$, the game stops when all urns have been chosen, i.e. $U_t = \emptyset$. The goal of player B is to end the game as soon as possible, while the goal of the adversary is to play for as long as it can.

Strategy of the player. At time t, the player picks the urn b_t that contains the least number of balls among the urns that were never chosen by the adversary, i.e. $b_t \in \arg\min_{i \in [k] \setminus \{a_0, \dots, a_t\}} n_i^t$. For this strategy, we state the main result of this section.

▶ **Theorem 3.** *Under this strategy, the game ends after at most $k \min\{\log(\Delta), \log(k)\} + 2k$ steps.*

Interpretation of the game. While the main focus of this paper is on collective tree exploration, a more immediate application of the above result is in resource allocation in the face of uncertainty. Given k workers and k (parallelizable) tasks of unknown length, our analysis shows that the 'best' way to reassign idle workers online is to reassign them to the unfinished task which has the least number of workers working on it. Using this simple rule, the number of times a worker changes task is at most $\log(k) + 2$ times the optimum (which is of order k) irrespective of the individual task lengths.

Proof. The set U_t does not increase with time. We denote its cardinality $u_t = |U_t|$. Denoting $N_t = \sum_{i \in U_t} n_i^t$ the total number of balls in urns of U_t, the possible number of balls for an urn of U_t lies in $\{\lceil \frac{N_t}{u_t} \rceil, \lfloor \frac{N_t}{u_t} \rfloor\}$. The game thus stops as soon as $\frac{N_t}{u_t} \geq \Delta$ and the quantity $x_t := \Delta u_t - N_t$, must thus be positive as long as the game lasts. We distinguish two options for the adversary at any step t:

(a) The adversary chooses an urn a_t that it previously chose ($a_t \notin U_t$). In this case, $u_{t+1} = u_t$ and $N_{t+1} = N_t + 1$. Note that this option is available to the adversary only if some ball lies outside of U_t, i.e. if $N_t \leq k - 1$.

(b) The adversary chooses an urn a_t that it has never chosen before ($a_t \in U_t$). In this case, $u_{t+1} = u_t - 1$ and $N_{t+1} = N_t - n_{a_t}^t + 1$.

We now will establish that the adversary always prefer option (a) to option (b). For parameters $u, N \in \{0, \dots, k\}$, we denote by $R(N, u)$ the largest number of steps that the game may still last after player B's move led to a configuration where $N_t = N$ and $u_t = u$ at any time t. Note that by the discussion above, this value is the same for all such configurations of the game. Clearly, $\Delta u - N \leq 0 \Rightarrow R(N, u) = 0$. Besides, in view of the options (a) and (b) just listed, one has the following, assuming $\Delta u - N > 0$:

$$N < k \Rightarrow R(N, u) = 1 + \max \begin{cases} R(N+1, u), \\ R(N - \lceil N/u \rceil + 1, u - 1), \\ R(N - \lfloor N/u \rfloor + 1, u - 1). \end{cases} \tag{1}$$

$$N = k \Rightarrow R(N,u) = 1 + \max \begin{cases} R(N - \lceil N/u \rceil + 1, u-1), \\ R(N - \lfloor N/u \rfloor + 1, u-1). \end{cases} \tag{2}$$

We now establish the following,

▶ **Lemma 4.** *For any $(u, N) \in \{0, \ldots, k\}$, it holds that:*
i) *Function $M \to R(M,u)$ is non-increasing, and*
ii) *The maximum in (1) for $N < k$ is always achieved by $R(N+1,u)$.*

Proof. For $u = 0$, $R(M,u) \equiv 0$ and there is nothing to prove. Assume that the two properties i) and ii) hold for $v = u - 1 \geq 0$. We will show that ii) holds for u. Consider $N < k$. By the monotonicity assumption i),

$$R(N - \lceil N/u \rceil + 1, u-1) \geq R(N - \lfloor N/u \rfloor + 1, u-1).$$

Assume thus that the adversary moves first to configuration $(N - \lceil N/u \rceil + 1, u-1)$. By assumption ii) at rank v, its next best move is to configuration $(N - \lceil N/u \rceil + 2, u-1)$. If alternatively the adversary had made a first move to $(N+1, u)$, it could then move to $(N + 1 - \lceil (N+1)/u \rceil + 1, u-1)$. Now by the monotonicity assumption ii) this can only improve the adversary's reward if $N - \lceil N/u \rceil + 2 \geq N + 1 - \lceil (N+1)/u \rceil + 1$, which is obviously true. We have thus established ii) at rank u. Monotonicity i) at rank u readily follows, since we now have that $R(N+1,u) = R(N,u) - 1$ if $\Delta u - N > 0$. ◀

From the lemma above, we conclude that a strategic adversary always prefer option (a) over option (b) when it is available. Playing option (b) grants the adversary a budget to choose option (a) for another $\lceil \frac{N_t}{u_t} \rceil - 1$ time steps. In such game, u_t is thus decremented by 1 every $\lceil \frac{k}{u_t} \rceil$ steps. The game stops if $u_t \leq \frac{k}{\Delta}$, thus right after $u_t = \lceil \frac{k}{\Delta} \rceil$. Assuming $\Delta \leq k$, the game then lasts a total time of at most $\lceil \frac{k}{k} \rceil + \lceil \frac{k}{k-1} \rceil + \cdots + \lceil \frac{k}{\lceil k/\Delta \rceil} \rceil \leq \sum_{h=\lceil k/\Delta \rceil}^{k} \left(\frac{k}{h} + 1\right) \leq k \sum_{h \geq k/\Delta+1}^{k} \frac{1}{h} + 2k \leq k \int_{k/\Delta}^{k} \frac{dx}{x} + 2k \leq k(\log(k) - \log(k/\Delta)) + 2k = k\log(\Delta) + 2k$. Instead assuming $k < \Delta$, the game will stop after $u_t = 1$ and the sum is thus bounded by $k \int_1^k \frac{dx}{x} + 2k \leq k\log(k) + 2k$. Overall, the game ends in at most $k \min\{\log(\Delta), \log(k)\} + 2k$ steps. ◀

3.2 Connection to BFDN

We start by giving some intuition to connect the game above to BFDN and then provide a proof of Lemma 2. The general picture is that balls of the game will correspond to robots exploring the tree whereas urns of the game will correspond to the anchors at the working depth d, i.e. the minimum depth of a dangling edge. Note that in BFDN, procedure Reanchor applies the strategy for the player of the game described above, by reassigning the current robot to the anchor of smallest load within set U, which is defined line 26 of Algorithm 1 by,

$$U = \{v \in V \;\; s.t. \;\; v \text{ is adjacent to some dangling edge and } \delta(v) = d\}. \tag{3}$$

▶ **Lemma 2** (Restated). *In an execution of* BFDN*, for any $d \in \{1, \ldots, D-1\}$, the number of calls to procedure* Reanchor *returning a node at depth d is at most $k(\min\{\log(k), \log(\Delta)\} + 3)$.*

Proof. We start the proof of the lemma by the following claim on BFDN.

▶ **Claim 5.** *At some round, if all anchors are at depth at most $d-1$, all nodes v explored at depth d are in either of these (non-exclusive) situations: their sub-tree $T(v)$ is entirely explored, or their sub-tree $T(v)$ hosts exactly one robot.*

Proof of Claim 5. Consider an explored node v at depth d that contains a dangling edge in its sub-tree $T(v)$. We show that $T(v)$ hosts one robot. The dangling edge must have an explored endpoint $v' \in T(v)$ that was attained by a robot performing depth-next moves. This robot cannot have left $T(v') \subset T(v)$ because v' is still adjacent to a dangling edge, thus that robot is still in $T(v)$. At most one robot is in $T(v)$ because v can only have been attained by a single robot, since all anchors are at depth $d-1$ or above. ◀

We now provide a reduction of the analysis of `BFDN` to the urns and balls game. We fix some depth $d \geq 1$ and bound the number N_d of times a robot is reanchored at depth d. We denote by U_0 the set U, defined by (3), in the first round when it consists of nodes at depth d. Since all anchors were at depth less than $k-1$ before that round, using Claim 5 we have that $|U_0| \leq k$ (in fact, $|U_0| \leq k-1$ because at least one robot must be at the root). Since all edges at depth less than $d-1$ are explored, we note that U_0 contains all nodes which are possible candidates for anchors at depth d and that $U \subset U_0$ for as long as it concerns nodes at depth d. For each candidate anchor in U_0, we formally re-anchor the robot exploring the corresponding sub-tree to this anchor. This does not change the algorithm's evolution because there are no more dangling edges at depth less than d so all robots head back directly to the root when they have finished explored below the associated candidate anchor.

We then increment counter c at every call of the procedure `Reanchor`, with possibly multiple increments within a single round. For counter value c, we denote by $a_c \in U_0$ the vertex to which the robot was previously anchored, and by $b_c \in U$ the vertex to which it is anchored next. Note that all nodes in $\{a_1, \dots, a_c\}$ can no longer be adjacent to a dangling edge. We stop the increment the last time a robot is anchored at depth d, which happens when there does not remain any node at depth d that is adjacent to some dangling edge.

Consider the number of calls C when for each node in U_0, either a robot returning from it has reached the root, or at least Δ robots are anchored at it. Then C is the duration of a run of the previous two-player game, initialized with one urn containing $k-u$ balls and u urns each containing one ball, where $u = |U_0| \in \{0, \dots, k-1\}$ and where player B implements the balancing strategy. Indeed the re-anchoring strategy of `BFDN` balances the numbers of robots assigned per anchor. A direct adaptation of our analysis also holds for this modified initial condition of the game, yielding the upper bound on C of $k(\min\{\log\Delta, \log k\} + 2)$. Once C assignments at depth d were made, at least Δ robots are assigned to nodes at depth d that are still adjacent to a dangling edge. In the subsequent d rounds `BFDN` can anchor each robot at most one last time before there is no more dangling edge at depth d. This yields the announced bound of $k(\min(\log(k), \log(\Delta)) + 3)$ on N_d. ◀

4 Extensions of `BFDN` to alternative settings

We now consider three settings where a `BFDN` strategy enjoys non-trivial runtime guarantees.

4.1 Restricted memory and communications

In this section, we study a setting where robots are allowed to communicate with a central planner only when they are located at the root and where they have access to $\Delta + D\log(\Delta)$ bits of internal memory. This setting encompasses the write-read communication model of [10] as detailed in Remark 5. Formally, we precise the setting as follows. At every node, the *ports*, which are defined as the endpoints of the adjacent edges, are numbered from 0 to $\Delta - 1$ where Δ is the maximum degree. A node v at depth $d \leq D$ is identified by the sequence of ports that leads to it from the root with $d\log_2(\Delta)$ bits. For every node distinct from the root, we assume that port number 0 leads to the root. As before, robots operate in rounds. All robots arriving at the root at some round t have their memory read and stored by the planner along with their identifier. The planner can then perform any computation and update the memory of the robots. All robots arriving at some node v distinct from the root at some round t can observe the list of all ports at v from which a robot has returned (these will be called "finished ports") and are given two choices: `SELECT` a port number as next move, or use a local routine `PARTITION`(v) enjoying the following properties,

- No two robots calling `PARTITION`(v) will ever be sent to the same port $j \geq 1$.
- If a robot calling `PARTITION`(v) at round t is sent to port $j \geq 0$, it means that `PARTITION`(v) has previously sent a robot to all ports $j' \geq j$ at round t or before.

In this model, `BFDN` is implemented as follows. In a stack of d port numbers (each represented by $\log_2(\Delta)$ bits) the central planner assigns to `Robot`$_i$ an anchor v_i at depth d that it will reach by unstacking port numbers and applying routine `SELECT`. When the robot reaches this node, the stack is empty and the robot will make consecutive calls to routine `PARTITION` that will eventually lead it back to the root. We ask that `Robot`$_i$ stores the finished port numbers of v_i using its additional Δ bits of memory. This information will be used by the central planner to update its candidates for future anchors, i.e. the value of the set U, as specified by Algorithm 2 below.

▶ Remark 5. The present model encompasses the classical write-read communication model of [10] where robots with unbounded memory communicate by synchronously writing and then synchronously reading information on whiteboards (of infinite size) located at each node of the tree. In this model, the information gathered at the root allows each robot located at the root to emulate the decision taken by the central planner regarding its next anchor assignment. Furthermore, since robots can log their passages at any node (see [10]) the local procedure `PARTITION` can easily be implemented, and the assumption that robots access the list of adjacent port number from which no robot has returned is granted.

▶ **Proposition 6.** *In this restricted communication model, the version of* `BFDN` *described above achieves tree exploration in at most* $\frac{2n}{k} + D^2(\min\{\log(k), \log(\Delta)\} + 3)$ *rounds.*

Proof. We note that the algorithm described above is the same as Algorithm 1, with a minor difference in the definition of U in procedure `Reanchor` line 26, which must now be computed using only information gathered at the root (see Algorithm 2 for details). Informally, U now denotes the set of all nodes at working depth d which *could* be adjacent to a dangling edge, given information collected at the `root`.

The key observation is that a candidate anchor v can be withdrawn from U as soon as a robot which had been anchored at v returns to the root. Consider again the urns-in-balls assignment rule $b_c = \arg\min_{v \in U \setminus \{a_1,\dots,a_c\}} n_v^c$, where n_v^c denotes the number of robots anchored at v upon increment c, but where nodes in U remain eligible as anchors until some robot has returned to the root from them. The proof of Theorem 3 entails that, for such a modified assignment rule, a robot will have returned from all nodes of U after at most $k(\min\{\log(k), \log(\Delta)\} + 3)$ reassignments, after which the root knows that there can be no more dangling edges at depth d.

Algorithm 2 below precises how the central planner uses information gathered by returning robots to update its knowledge of eligible anchors at the working depth d. Denoting the list of all possible anchors at depth d by A and the list of anchors at depth d from which a robot has returned by R, the planner implements `Reanchor` with set $U = A \setminus R$. When $A \setminus R = \emptyset$, a robot has returned from all anchors at depth d and d is incremented. The planner keeps track of $U' = A' \setminus R'$, which contains the children of A that may be adjacent to a dangling edge, or equivalently the ports of A that are not known to be finished. This update is performed using the memory of the returning robots. ◀

Algorithm 2 `BFDN` "Breadth-First Depth-Next" (central planner at the root).

```
Require: At most $k$ robots arriving at the root at some round.
Ensure: Assigns a node $v$, represented by a sequence of port numbers, to each robot.
 1: $d$ = working depth ;
 2: $A$ = list of anchors at depth $d$ ;
 3: $R$ = nodes of $A$ from which a robot has returned ;
 4: $A'$ = list of children of nodes in $A$ ;
 5: $R'$ = nodes of $A'$ from which a robot has returned ;
 6: Read memory of returning robots and update $R, A', R'$.
 7: if $A \setminus R = \emptyset$ then
 8:     if $A' \setminus R' = \emptyset$ then
 9:         Exploration is finished and robots wait at the root.
10:     else
11:         $d \leftarrow d + 1$
12:         $A \leftarrow A' \setminus R'$                         ▷ contains at most $k$ elements.
13:         $R, A', R' \leftarrow \emptyset$
14: Reanchor the robots to nodes of minimum load in $A \setminus R$, such that after this operation
    the numbers of robots per anchor differ by at most one.
```

4.2 Adversarial robot break-downs

So far we assumed that all robots traverse exactly one edge per time-step. We relax this assumption in the present section, assuming instead that some adversary decides at each time-step and for each robot whether the robot actually moves, or instead incurs a break-down, being stalled at its current location. Our aim remains to to explore the tree in as few moves as possible. However we no longer require that the robots return to the root at the end of exploration, because the adversary could decide to break-down some robot indefinitely.

Formally, at each round $t \in \mathbb{N}$, robot i is allowed to make a move if some variable $M_{ti} = 1$ whereas it is blocked at its current position if $M_{ti} = 0$. For this adversarial model, we assume that $\mathbb{M} = (M_{ti})_{t\in\mathbb{N}, i\in[k]}$ is an arbitrary sequence of binary values that takes only a finite number of 1 (allowed moves). We denote the average distance travelled by the robots $A(\mathbb{M})$ which equals $A(\mathbb{M}) = \frac{1}{k}\sum_{t\in\mathbb{N}}\sum_{i\in[k]} M_{ti}$.

For this setting, we consider `BFDN` as specified in Algorithm 1, with the minor modification that at each round t the only robots taking part in the assignment process are those which are allowed to move. More precisely, we replace the `for` loop of Algorithm 1 (**for** $i \in \{1, \dots, k\}$ **do**) with an iteration over all robots that may move (**for** $i \in \{i : M_{ti} = 1\}$ **do**). This modification is introduced to ensure that when multiple robots are at the same location, blocked robots do not prevent unblocked robots from traversing dangling edges.

▶ **Proposition 7.** *For any sequence of allowed moves $\mathbb{M} \in \{0,1\}^{\mathbb{N}\times[k]}$ satisfying $A(\mathbb{M}) \geq \frac{2n}{k} + D^2(\log(k) + 3)$ all edges of the tree will be visited by the above variant of* `BFDN`.

Proof. Again, the proof is very similar to that of Theorem 1 and all claims `1-5` all naturally adapt to this setting. As an example, we adapt the third claim as follows.

▶ **Claim 3** (Restated). *Consider a sequence of moves by some* $\texttt{Robot}_i$ *that starts at the root with the assignment of an anchor* v *of depth* $\delta(v) = d$ *and that ends with the return of* $\texttt{Robot}_i$ *to the root after* T_x *allowed moves of* $\texttt{Robot}_i$*. In this sequence,* $\texttt{Robot}_i$ *has explored exactly* $(\mathsf{T}_x - 2d)/2$ *dangling edges.*

The adversarial nature of the urns and balls game of Section 3 makes it applicable to the present setup, and Lemma 2 straightforwardly holds except for the $\log(\Delta)$ guarantee. Indeed, the adversary could choose to block all robots at a specific anchor until all k robots reach that anchor, which happens after at most $k(\log(k) + 3)$ anchor assignments. ◀

▶ Remark 8. Other adversarial settings could be considered, for instance with an adversary that observes the moves that the robots have selected before choosing which robots to block. Another extension of interest would consist in relaxing the slotted time assumption to consider instead continuous time evolution, which could capture more realistic scenarios.

4.3 Collaborative exploration of non-tree graphs

The algorithm BFDN described above can be executed on any graph if it undergoes a minor modification: that any robot traversing on a dangling edge and arriving on a node explored earlier by another robot should go back from where it came and "close" the corresponding edge (this edge will never be used again). A similar technique was already proposed by [1] to adapt the algorithm of [10] to graphs. Unfortunately, without further assumption, the guarantees of BFDN do not generalize to graphs with n *edges* and *radius* D, where the radius is defined as the maximum distance between a node and the origin of the robots.

We therefore make the additional assumption that at any given node, a robot knows its distance to the origin in the underlying graph. Though restrictive, this assumption holds in some contexts of interest. It is for instance satisfied for the exploration of grid graphs with rectangular obstacles considered in [12] because the distance of any node with coordinates $(i, j) \in \mathbb{N}^2$ to the origin is equal to the so-called Manhattan distance $i + j$.

In that context, consider the following variant of BFDN: a robot traversing a dangling edge e will backtrack and "close" this edge if either of these two conditions is satisfied: (1) e led to a node that is already explored (2) e led to a node that is not strictly further to the origin than its first endpoint. In the case of (2), the node that is reached by the edge over which the respective robot backtracks is not considered as explored.

▶ **Proposition 9.** *Given a graph* $G = (V, E)$ *with* n *edges, diameter* D *and maximum degree* Δ*, assuming that the* k *robots are aware at all times of their distance to the origin and implement the above variant of* BFDN*, collaborative graph exploration is completed in at most* $\frac{2n}{k} + D^2(\min\{\log(\Delta), \log(k)\} + 3)$ *rounds.*

Proof. It is clear that at the end of the execution of this algorithm, the edges which have never been closed form breadth-first tree of the graph with depth D. This tree is explored efficiently by BFDN while other edges are traversed at most twice by a single robot (or once by two robots, each coming from both endpoints, that will swap their identities). This leads to a total runtime of at most $\frac{2n}{k} + D^2(\min\{\log(\Delta), \log(k)\} + 3)$. ◀

5 Recursive Algorithms for Improved Dependence on Depth D

In this section we develop a general recursive construction of so-called *anchor-based algorithms* which, applied to BFDN, yields the following result. It can be seen as a generalization of Theorem 1 as, for $\ell = 1$, it provides the same upper-bound up to a factor 4.

▶ **Theorem 10.** *For any integer $\ell \geq 1$, `BFDN`$_\ell$, an associated recursive version of `BFDN`, explores a tree with n nodes, depth D, maximum degree Δ with k robots in $\frac{4n}{k^{1/\ell}} + 2^{\ell+1}(\ell + 1 + \min\{\log(\Delta), \log(k)/\ell\})\, D^{1+1/\ell}$ rounds.*

To describe our recursive construction we need the following definitions. Given a node v in a tree T, $P_T[v]$ denotes the path from v to the root of T, and $P_T(v) = P_T[v] \setminus \{v\}$. Given two nodes u, v in a tree T, $\mathrm{LCA}_T(u, v)$ denotes their lowest common ancestor in T. We say that a explored node is *open* as long as it has at least one dangling adjacent edge. We say that it is *closed* as soon as a robot has traversed its last dangling edge. Note that open nodes are the parents of dangling edges. We decompose the exploration of an edge into two edge events as follows. An *edge event* occurs when a robot traverses an edge from parent to child for the first time, or when a robot traverses an edge from child to parent for the first time. There are thus at most $2(n-1)$ edge events in any exploration. Edges for which only one event has occurred are said to be *half explored.*

Anchor-based algorithm. Given k robots, an activity parameter $k^* \in [k]$, and a depth d, an *anchor-based* algorithm $\mathcal{A}(k^*, k, d)$ is by definition an exploration algorithm by k robots meeting the following requirements. Each robot is in one of the two states *active* or *inactive.* Each active robot i is assigned to a node v_i of the tree called its *anchor.* The algorithm must explore the tree so as to bring anchors at depth d while maintaining a list of invariants. The full list of so-called “Anchor-based invariants” is given in Appendix B. It mainly includes a variant of Claim 4 called *Open Node Coverage* which specifies that all open nodes must always be in $\cup_{i \in A} T(v_i)$ where A is the set of active robots. Other invariants mainly specify properties of the positions of the robots with respect to the partially explored tree and ensure that we can start an execution of an anchor-based algorithm after having interrupted the execution of another anchor-based algorithm.

Initially, the algorithm starts from any partially explored tree, with all robots active and anchored at the root. Robots must be in so-called *Parallel DFS Positions*, a requirement ensuring that all invariants are initially satisfied (see Appendix B). Active robots are allowed to move and explore the tree while inactive robots must be at depth at most d and wait. We distinguish two phases in the execution of the algorithm. As long as some anchor is at depth less than d or is not closed, we say that the algorithm runs *shallow.* During this first “shallow” phase, the algorithm must have at least k^* active robots at all rounds. When all anchors are at depth d and are all closed, we say that the algorithm runs *deep.* In this second “deep” phase, it is required that all active robots trigger an edge event at each round. However, the number of active robots may get below k^* during that phase. At any round, the algorithm may turn a robot into inactive or active as long as the requirements for the two phases are met. Finally, the algorithm can terminate when all robots are inactive. The Open Node Coverage invariant implies that the tree is then completely explored (see Appendix B).

Divide depth functor. We now define the *divide depth functor* $\mathcal{D}$, a map that takes an anchor-based algorithm and transforms it into another anchor-based algorithm as follows. Given an anchor-based algorithm $\mathcal{A}(k^*, k', d')$, a number n_{team} of teams and a number n_{iter} of iterations, we construct the exploration algorithm $\mathcal{D}[\mathcal{A}(k^*, k', d'); n_{team}; n_{iter}]$ for terminating the exploration of a partially explored tree. It uses $k = n_{team} k'$ robots for exploring the tree up to depth $d = n_{iter} d'$ in n_{iter} iterations where each iteration makes anchors progress d' deeper. More precisely, the i-th iteration runs parallel instances of $\mathcal{A}(k^*, k', d')$ in at most n_{team} sub-trees rooted at nodes with depth $(i-1)d'$. We assume that the previous iteration has terminated with a set R of at most $k^* \leq n_{team}$ anchors at depth

$(i-1)d'$. Relying on the Open Node Coverage invariant, we then restrict the exploration to the sub-trees rooted in R. Robots are thus partitioned into n_{team} teams of k' robots each. Each node $r \in R$ is taken in charge by a distinct team which runs an instance $\mathcal{A}_r(k^*, k', d')$ of $\mathcal{A}(k^*, k', d')$ on $T(r)$. When $|R| < n_{team}$, all robots in unassigned teams are inactive and wait at their position until the end of the current iteration. All other teams explore in parallel their sub-trees. We interrupt all running instances simultaneously when the overall number of active robots gets below k^* so that we can use their anchors as roots in the next iteration. As any single instance has activity parameter k^* this cannot happen until all anchors are at depth d' in each sub-tree, that is depth $i \cdot d'$ in T. After n_{iter} iterations, this guarantees that all nodes up to depth d have been closed and that exploration finally continues in at most k^* sub-trees rooted at depth d. See Appendix C for a formal description of the resulting anchor-based algorithm $\mathcal{B}(k^*, k, d) = \mathcal{D}[\mathcal{A}(k^*, k', d'); n_{team}; n_{iter}]$.

We say that an anchor-based algorithm $\mathcal{A}(k^*, k, d)$ has *f-shallow efficiency* for parameter f if it triggers at least $k^*(\mathsf{T} - f)$ edge events when running shallow during T rounds where parameter f may depend on k and d. We then have the following

▶ **Proposition 11.** *Given an anchor-based algorithm $\mathcal{A}(k^*, k', d')$, integers $n_{team} \geq k^*$ and $n_{iter} \geq 1$, $\mathcal{D}[\mathcal{A}(k^*, k', d'); n_{team}; n_{iter}]$ is correct and it is an anchor-based exploration algorithm $\mathcal{B}(k^*, k, d)$ for $k = n_{team}k'$ robots with depth $d = n_{iter}d'$. If moreover $\mathcal{A}(k^*, k', d')$ has f'-shallow efficiency, then $\mathcal{D}[\mathcal{A}(k^*, k', d'); n_{team}; n_{iter}]$ has f-shallow efficiency with $f = n_{iter}f' + n_{iter}^2 d' = n_{iter}(f' + d)$.*

Its proof is deferred to Appendix C. The reason for f-shallow efficiency is the following. Consider the i-th iteration of $\mathcal{D}_{\mathcal{A},k',d'}(k^*, k, d)$. Moving robots towards their associated root takes $2(i-1)d'$ rounds. Now, count the number T^1 of rounds where at least one of the instances has not run deep. As such an instance has run shallow during T^1 rounds, it has triggered at least $k^*(\mathsf{T}^1 - f')$ edge events by f'-shallow efficiency of $\mathcal{A}(k^*, k, d)$. During the remaining T^2 rounds of the iteration, all instances run deep. As this continues as long as k^* robots or more are active, at least k^* edge events are triggered per round, that is $k^*\mathsf{T}^2$ or more in total. Letting $\mathsf{T}_i = 2(i-1)d' + \mathsf{T}^1 + \mathsf{T}^2$ denote the number of rounds spent in the ith iteration, the number of edge events triggered during that iteration is thus at least $k^*(\mathsf{T}_i - f' - 2(i-1)d')$. The algorithm runs shallow during the n_{iter} iterations which last overall $\mathsf{T} = \sum_{i=1}^{n_{iter}} \mathsf{T}_i$. By summation, we get that it then triggers at least $k^*(\mathsf{T} - n_{iter}f' - n_{iter}^2 d')$ edge events as $\sum_{i=1}^{n_{iter}}(i-1) < n_{iter}^2/2$.

BFDN. Our first candidate for applying the divide depth functor is the following variant of Algorithm 1, denoted, $\texttt{BFDN}_1(k, k, d)$, where the procedure `Reanchor` is modified for assigning anchors at depth at most d. Precisely, we replace Line 26 with:

$U = \{v \in V \;\; s.t. \; v$ is adjacent to some unexplored edge and $\delta(v)$ is minimal and $\delta(v) \leq d\}$.

Note that this modification implies that when there are no more dangling edges at depth at most d, robots start to be anchored to the root and are then considered as inactive. Note that according to Claim 5 for depth $d+1$, there still remains exactly one robot in each sub-tree rooted at depth $d+1$ which is not entirely explored. These robots remain active until they have completely explored their sub-tree. $\texttt{BFDN}_1(k, k, d)$ thus terminates only when the tree has been fully explored. We also slightly modify the anchoring of robots: when a robot i is anchored at v_i it might happen that there are no more dangling edges at depth $\delta(v_i)$ or less thanks to the exploration of other robots. If this happens when $v_i \in P(u_i)$ and $\delta(v_i) < d$, we re-anchor robot i at the children of v_i in $P[u_i]$. This modification does not change the movements of robot i as it is then in a sequence of depth-next moves and will go up when reaching v_i anyway. However, this modification will ensure the preservation of the Partial Exploration invariant defined in Appendix B. It also implies that when there are no more dangling edges at depth at most d, all anchors are then at depth d.

One can then easily check that $\texttt{BFDN}_1(k,k,d)$ is an anchor-based algorithm. For example, the Open Node Coverage invariant is shown as Claim 4; see Appendix B for more details. We also note that $\texttt{BFDN}_1(k,k,d)$ has $c_1(k)d^2$-shallow efficiency where $c_1(k) = \min\{\log\Delta, \log k\}+2$. Indeed, $\texttt{BFDN}_1(k,k,d)$ runs exactly as Algorithm 1 as long as there are dangling edges at depth at most d, that is as long as the algorithm is running shallow. If this phase lasts T rounds, it triggers at least $k(\mathsf{T} - c_1(k)d^2)$ edge events. The proof is similar to that of Theorem 1 using Lemma 2 with the slight subtlety that we count edge events. The reason is that when starting from a partially explored tree where robots are in Parallel DFS Positions, the moves when robots go up still trigger edge events although no new edge may be discovered.

The $\texttt{BFDN}_\ell(k^*, k, d)$ anchor-based algorithm. We construct recursively a series of algorithms $\texttt{BFDN}_\ell(k^{1/\ell}, k, d)$ for $\ell \geq 1$ as follows. Assuming that k and d are both ℓ-th powers of integers, we define for $\ell \geq 2$ the algorithm $\texttt{BFDN}_\ell(k^*, k, d) := \mathcal{D}[\texttt{BFDN}_{\ell-1}(k^*, k/n_{team}, d/n_{iter}); n_{team}; n_{iter}]$ with $k^* = n_{team} = k^{1/\ell}$ and $n_{iter} = d^{1/\ell}$. We let $k' = k/n_{team} = k^{(\ell-1)/\ell}$ and $d' = d/n_{iter} = d^{(\ell-1)/\ell}$ denote the parameters used for $\texttt{BFDN}_{\ell-1}$. Note that k' and d' are both $(\ell-1)$-th powers of integers and recursive calls all have integer-valued parameters. The activity parameter of instances $\texttt{BFDN}_{\ell-1}(k^*, k', d')$ indeed satisfies $(k')^{1/(\ell-1)} = k^{1/\ell} = k^*$. As we use $n_{team} = k^*$, we indeed respect the constraint $k^* \leq n_{team}$. We can bound its shallow efficiency according to the following statement:

▶ **Lemma 12.** *Given an integer $\ell \geq 2$, two integers k and d that are both ℓth powers of integers, $\texttt{BFDN}_\ell(k^{1/\ell}, k, d)$ is $c_\ell(k)d^{1+1/\ell}$-shallow efficient with $c_\ell(k) = c_1(k^{1/\ell}) + \ell - 1$.*

Proof. As $\texttt{BFDN}_1(k^{1/\ell}, k^{1/\ell}, d^{1/\ell})$ is $c_1(k^{1/\ell})d^{2/\ell}$-shallow efficient, by induction (Proposition 11) $\texttt{BFDN}_j(k^{1/\ell}, k^{j/\ell}, d^{j/\ell})$ is $(c_1(k^{1/\ell})+j-1)d^{(j+1)/\ell}$-shallow efficient for $j = 2, \dots, \ell$. ◀

▶ **Definition 13** (of $\texttt{BFDN}_\ell$). *If k is the ℓ-th power of an integer, consider the sequence of depths $d_j = 2^{j\ell}$ for $j = 1, 2, \dots$ Algorithm $\texttt{BFDN}_\ell$ consists in running $\texttt{BFDN}_\ell(k^{1/\ell}, k, d_1)$, interrupting it right after its last iteration (without running deep further), then running $\texttt{BFDN}_\ell(k^{1/\ell}, k, d_2)$ with the current robot positions and anchor assignments until its last iteration finishes, and so on. When running $\texttt{BFDN}_\ell(k^{1/\ell}, k, d_j)$ with $j = \lceil \frac{\log_2 D}{\ell} \rceil$, all anchors reach depth D and the algorithm terminates. If k is not an integer to the power ℓ, we use $K = \lfloor k^{1/\ell} \rfloor^\ell \leq k$.*

Poof of Theorem 10. Assume first that k is the ℓ-th power of some integer. In a run of $\texttt{BFDN}_\ell$, denote by T_j the number of rounds that the call to $\texttt{BFDN}_\ell(k^{1/\ell}, k, d_j)$ lasts. This call triggers at least $k^{1/\ell}(\mathsf{T}_j - c_\ell(k)d_j^{1+1/\ell})$ edge events by applying Lemma 12. We can thus bound the overall running time $\mathsf{T} = \sum_{j=1}^{\lceil (\log_2 D)/\ell \rceil} \mathsf{T}_j$ by summing over all calls: $2n \geq k^{1/\ell}\left(\mathsf{T} - c_\ell(k)\sum_{j=1}^{\lceil (\log_2 D)/\ell \rceil} d_j^{1+1/\ell}\right)$. As we have $\sum_{j=1}^{\lceil (\log_2 D)/\ell \rceil} d_j^{1+1/\ell} = \sum_{j=1}^{\lceil (\log_2 D)/\ell \rceil} 2^{(\ell+1)j} \leq \frac{2^{(\ell+1)((\log_2 D)/\ell+2)}-1}{2^{\ell+1}-1} \leq 2^{\ell+1}D^{1+1/\ell}$, we obtain $\mathsf{T} \leq \frac{2n}{k^{1/\ell}} + 2^{\ell+1}c_\ell(k)D^{1+1/\ell}$. For arbitrary k, with $K = \lfloor k^{1/\ell} \rfloor^\ell$, using $K^{1/\ell} \geq k^{1/\ell}/2$, we obtain a time bound of $\mathsf{T} \leq \frac{4n}{k^{1/\ell}} + 2^{\ell+1}(\ell - 1 + c_1(k^{1/\ell}))D^{1+1/\ell}$, yielding the runtime bound announced in Theorem 10 since $c_1(k^{1/\ell}) = 2 + \min\{\log(\Delta), \log(k)/\ell\}$. ◀

References

1 Peter Brass, Flavio Cabrera-Mora, Andrea Gasparri, and Jizhong Xiao. Multirobot tree and graph exploration. *IEEE Trans. Robotics*, 27(4):707–717, 2011. doi:10.1109/TRO.2011.2121170.

2 Peter Brass, Ivo Vigan, and Ning Xu. Improved analysis of a multirobot graph exploration strategy. In *13th International Conference on Control Automation Robotics & Vision, ICARCV 2014, Singapore, December 10-12, 2014*, pages 1906–1910. IEEE, 2014. doi:10.1109/ICARCV.2014.7064607.

3 Romain Cosson, Laurent Massoulié, and Laurent Viennot. Breadth-first depth-next: Optimal collaborative exploration of trees with low diameter. *arXiv preprint arXiv:2301.13307*, 2023.

4 Romain Cosson, Laurent Massoulié, and Laurent Viennot. Brief announcement: Efficient collaborative tree exploration with breadth-first depth-next. In Rotem Oshman, Alexandre Nolin, Magnús M. Halldórsson, and Alkida Balliu, editors, *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing, PODC 2023, Orlando, FL, USA, June 19-23, 2023*, pages 24–27. ACM, 2023. doi:10.1145/3583668.3594568.

5 Dariusz Dereniowski, Yann Disser, Adrian Kosowski, Dominik Pajak, and Przemyslaw Uznanski. Fast collaborative graph exploration. *Inf. Comput.*, 243:37–49, 2015. doi:10.1016/j.ic.2014.12.005.

6 Yann Disser, Frank Mousset, Andreas Noever, Nemanja Skoric, and Angelika Steger. A general lower bound for collaborative tree exploration. *Theor. Comput. Sci.*, 811:70–78, 2020. doi:10.1016/j.tcs.2018.03.006.

7 Miroslaw Dynia, Miroslaw Korzeniowski, and Christian Schindelhauer. Power-aware collective tree exploration. In Werner Grass, Bernhard Sick, and Klaus Waldschmidt, editors, *Architecture of Computing Systems - ARCS 2006, 19th International Conference, Frankfurt/Main, Germany, March 13-16, 2006, Proceedings*, volume 3894 of *Lecture Notes in Computer Science*, pages 341–351. Springer, 2006. doi:10.1007/11682127_24.

8 Miroslaw Dynia, Jaroslaw Kutylowski, Friedhelm Meyer auf der Heide, and Christian Schindelhauer. Smart robot teams exploring sparse trees. In Rastislav Kralovic and Pawel Urzyczyn, editors, *Mathematical Foundations of Computer Science 2006, 31st International Symposium, MFCS 2006, Stará Lesná, Slovakia, August 28-September 1, 2006, Proceedings*, volume 4162 of *Lecture Notes in Computer Science*, pages 327–338. Springer, 2006. doi:10.1007/11821069_29.

9 Miroslaw Dynia, Jakub Lopuszanski, and Christian Schindelhauer. Why robots need maps. In Giuseppe Prencipe and Shmuel Zaks, editors, *Structural Information and Communication Complexity, 14th International Colloquium, SIROCCO 2007, Castiglioncello, Italy, June 5-8, 2007, Proceedings*, volume 4474 of *Lecture Notes in Computer Science*, pages 41–50. Springer, 2007. doi:10.1007/978-3-540-72951-8_5.

10 Pierre Fraigniaud, Leszek Gasieniec, Dariusz R. Kowalski, and Andrzej Pelc. Collective tree exploration. *Networks*, 48(3):166–177, 2006. doi:10.1002/net.20127.

11 Yuya Higashikawa, Naoki Katoh, Stefan Langerman, and Shin-ichi Tanigawa. Online graph exploration algorithms for cycles and trees by multiple searchers. *J. Comb. Optim.*, 28(2):480–495, 2014. doi:10.1007/s10878-012-9571-y.

12 Christian Ortolf and Christian Schindelhauer. Online multi-robot exploration of grid graphs with rectangular obstacles. In Guy E. Blelloch and Maurice Herlihy, editors, *24th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '12, Pittsburgh, PA, USA, June 25-27, 2012*, pages 27–36. ACM, 2012. doi:10.1145/2312005.2312010.

13 Christian Ortolf and Christian Schindelhauer. A recursive approach to multi-robot exploration of trees. In Magnús M. Halldórsson, editor, *Structural Information and Communication Complexity - 21st International Colloquium, SIROCCO 2014, Takayama, Japan, July 23-25, 2014. Proceedings*, volume 8576 of *Lecture Notes in Computer Science*, pages 343–354. Springer, 2014. doi:10.1007/978-3-319-09620-9_26.

A Comparisons between Algorithms CTE, Yo* and BFDN

We provided in Figure 1 a picture of how BFDN compares in terms of runtime with other state-of-the art algorithms for collaborative tree exploration. The regions are defined up to multiplicative constants that only depend on k. We included in the figure only algorithms requiring no assumptions on the tree structure. Four algorithms thus appear in the figure: the original "collaborative tree exploration" CTE algorithm of [10] with runtime $\mathcal{O}(\frac{n}{\log(k)}+D)$, the recursive algorithm Yo* of [13] with runtime $\mathcal{O}(2^{\mathcal{O}(\sqrt{\log D \log\log k})}\log k(\log n+\log k)(n/k+D))$, which we reduced to smaller quantities to simplify the picture, BFDN with runtime $2n/k+D^2\log(k)$ as well as its recursive variant $\mathtt{BFDN}_\ell$.

Figure 1 highlights that BFDN is the only algorithm to outperform CTE of [10] in an unbounded range of parameters (n, D). Indeed, the other competitor, Yo*, is outperformed by CTE when $n \geq e^k$ or when $D \geq e^{\log(k)^2}$. Yet, CTE remains the most efficient algorithm for trees with small depth. We detail below the calculations that led to Figure 1.

Comparison between BFDN and CTE. Since the runtime of any collaborative tree algorithm exceeds n/k and D, it is sufficient to compare the suboptimal terms of both algorithms which are $D^2\log(k)$ and $n/\log(k)$ for BFDN and CTE respectively. It therefore turns out that BFDN is faster than CTE in the range $D^2\log(k)^2 \leq n$.

Comparison between CTE and Yo*. First, we simplified the runtime of Yo* to $\mathcal{O}(\log(n)n/k+D)$, which gives that it can outperform the $\mathcal{O}(n/\log(k)+D)$ of [10] only in the range $n \leq e^{k/\log(k)}$ which we extend to $n \leq e^k$ in the picture. After, we simplified the runtime of Yo* to $\mathcal{O}(e^{\sqrt{\log(D)}}n/k+D)$ to obtain the range $D \leq e^{\log(k)^2}$. Finally, we simplified the runtime of Yo* to $D\log(n)\log(k)$ to get that CTE outperforms Yo* for trees satisfying $D \geq \frac{n}{\log(n)}\log(k)^2$.

Comparison between BFDN and Yo*. We used the comparisons above for $e^k \leq n$ or $e^{\log(k)^2} \leq D$, and completed by the following simplification of the runtime of Yo* to $\mathcal{O}(\log(k)n/k+D)$. BFDN is thus faster than Yo* when $\log(k)D^2 \leq \log(k)n/k$, that is when $kD^2 \leq n/k$.

Comparison between $\mathtt{BFDN}_\ell$ and CTE. We note that $\mathtt{BFDN}_\ell$ may outperform CTE only if $k^{1/\ell} > \log(k)$, or equivalently if $\ell < \frac{\log(k)}{\log(\log(k))}$, which we assumed in the caption of the Figure. Under this condition, $\mathtt{BFDN}_\ell$ outperforms CTE if $2^\ell\log(k)D^{1+1/\ell} < \frac{n}{\log(k)}$. Since we have $2^\ell < k$, this condition is met if $D < \frac{1}{k\log(k)^2}n^{\ell/(\ell+1)}$.

Comparison between $\mathtt{BFDN}_\ell$ and BFDN. If $n/k > D^2$, if is clear that BFDN outperforms $\mathtt{BFDN}_\ell$. On the other hand, if $n/k^{1/\ell} < D^2$, $\mathtt{BFDN}_\ell$ outperforms BFDN.

B Formal description of Anchor-based Invariants

During the execution of an anchor-based algorithm, it is required that the partially explored tree, the set $A \subseteq [k]$ of active robots, the anchor assignment $(v_i)_{i\in A}$, and the positions $(u_i)_{i\in[k]}$ of the robots always satisfy the following invariants:

- all open nodes of the currently explored tree are in $\cup_{i\in[k]}P_T[u_i]$, (DFS Open Coverage)
- for any two robots $i \neq j$, all nodes in $P_T(\mathrm{LCA}_T(u_i, u_j))$ are closed, (Parallel Positions)

- for all active robot i such that $v_i \in P_T[u_i]$, all edges in the path from v_i to u_i are half explored, (Partial Exploration)
- for all active robot $i \in A$, $\delta(v_i) \leq d$, (Limited Anchor Depth)
- all inactive robots are located at depth at most d, (Inactive Depth)
- all open nodes of the currently explored tree are in $\cup_{i \in A} T(v_i)$, (Open Node Coverage)
- if $\exists i \in A$ such that either $\delta(v_i) < d$ or v_i is open, then at least k^* robots are active, (Shallow Activity)
- if all anchors $\{v_i : i \in A\}$ are at depth d and are close, each active robot triggers an edge event at each round. (Deep Activity)

Initially, robots are said to be in *Parallel DFS Positions* when DFS Open Coverage, Parallel Positions and Partial Exploration are all three satisfied when assuming that all robots are active and anchored at the root. One can easily check that other invariants are then also satisfied.

Properties of an anchor-based algorithm. The Open Node Coverage invariant implies that all nodes at depth less than d' are closed where $d' = \min_{i \in A} \delta(v_i)$ is the minimum depth of an anchor. The Shallow Activity invariant implies that the number of active robots may decrease below k^* only when all anchors are at depth d and consequently when all nodes up to depth d are closed. The Open Node Coverage invariant also implies that for any dangling edge adjacent to a explored node w, there exists at least one active robot i such that w is in $T(v_i)$. This implies that if all anchors are at depth d and if i is the last robot with anchor v_i, it cannot become inactive unless $T(v_i)$ has been completely explored. This indeed implies that the algorithm cannot terminate unless the full tree has been completely explored: as long as there remains an open node w, some robot i must be active with an ancestor of w as anchor. Recall that we require that the algorithm cannot terminate unless all robots are inactive.

BFDN. $\texttt{BFDN}_1(k, k, d)$ is an anchor-based algorithm. Indeed, the Open Node Coverage invariant is shown as Claim 4; the DFS Open Coverage and Partial Exploration invariants come from the similarity of `DN` moves with a DFS traversal, while the Parallel Positions invariant comes from the selection of distinct dangling edges when several robots are located at the same node. The Limited Anchor Depth and Inactive Depth invariants are satisfied by the modification of anchor selection. The Shallow Activity invariant comes from the fact that all robots are active as long as there remain some dangling edge at depth at most d. Finally, the Deep Efficiency invariant comes from Claim 5 as when the algorithm runs deep, each sub-tree at depth $d + 1$ which is not completely explored contains exactly one robot performing a DFS-like traversal of the sub-tree.

We also note that we can start $\texttt{BFDN}_1(k, k, d)$ from any partially explored tree where robots are in Parallel DFS Positions as long as each robot i, which is in a position u_i with open ancestors, gets anchored to a node v_i of $P[u_i]$ such that all nodes of $P(v_i)$ are closed. Such a situation occurs in `BFDN` when a robot is performing `DN` moves. It is thus possible to start a robot in any such situation so that it will then behave similarly as in `BFDN`. The other robots see only closed nodes and thus get to the root according to Algorithm 1 where they get re-anchored.

C Divide-depth Algorithm

Algorithm 3 Divide depth algorithm $\mathcal{D}[\mathcal{A}(k^*, k', d'); n_{team}; n_{iter}]$.

```
Require: An anchor-based exploration algorithm $\mathcal{A}(k^*, k', d')$, integers $n_{team} \geq k^*$ and
    $n_{iter} \geq 1$, a partially explored tree $T$ with $k = n_{team}k'$ robots in Parallel DFS Positions
    and such that at most $k^*$ robots are at depth greater than 0.
Ensure: All nodes are explored and closed.
 1: $R \leftarrow \{\texttt{root}(T)\}$                    ▷ Set of sub-tree roots in next iteration.
 2: $A \leftarrow \{i \in [k] : u_i \neq \texttt{root}(T)\}$      ▷ Set of robots having already progressed in $T$.
 3: All robots are active and have $\texttt{root}(T)$ as anchor.
 4: for $i = 1, \dots, d/d'$ do
 5:     ▷ Iteration $i$:
 6:     For all $r \in R$, let $k_r = |\{i \in A : v_i = r\}|$ be the number of robots having progressed
    in $T(r)$.
 7:     Partition robots into $|R|$ teams $(B_r)_{r \in R}$ of $k'$ robots each, one per node $r \in R$:
 8:         each robot $i \in A$ is assigned to $v_i$,
 9:         for all $r \in R$, $k' - k_r$ robots in $[k] \setminus A$ are assigned to $r$. ▷ We rely on $k_r \leq k'$ and
    $|R| \leq n_{team}$.
10:     All robots in team $B_r$ are assigned to anchor $r$: we set $v_i \leftarrow r$ for all $i \in B_r \setminus A$.
11:     All robots in $\cup_{r \in R} B_r \setminus A$ are turned to active, and move to their anchor in $2(i-1)d'$
    rounds.                                       ▷ Moves for rebalancing robots.
12:     All robots in $[k] \setminus \cup_{r \in R} B_r$ are turned to inactive and wait at their current position.
13:     Each team associated to $r \in R$ initializes independently an instance $\mathcal{A}_r(k^*, k', d')$ for
    exploring $T(r)$.
14:     At any round, we let $A_r$ denote the set of active robots among the team exploring
    $T(r)$.
15:     while $|\cup_{r \in R} A_r| \geq k^*$ do
16:         Run in parallel one round of all instances $\mathcal{A}_r(k^*, k', d')$ for $r \in R$.
17:     end while
18:     $A \leftarrow |\cup_{r \in R} A_r|$                       ▷ Overall set of active robots.
19:     $R \leftarrow \{v_i : i \in A\}$                  ▷ Roots of sub-trees not fully explored yet.
20: Continue running instances $\mathcal{A}_r(k^*, k', d')$ of the last iteration for all $r \in R$.  ▷ Running
    deep.
```

Proof of Proposition 11. We first check that all invariants are preserved by induction on the iteration number i. The main argument is that all anchors are at depth $i \cdot d'$ after Iteration i. We require that the DFS Open Coverage, Parallel DFS Positions and Partial Exploration invariants are satisfied by the initial positions of robots. All remaining invariants are also satisfied as the only initial anchor is at depth zero. Assume that all invariants are satisfied up to the beginning of Iteration i, and that nodes in R are at depth $(i-1)d'$.

The Inactive Depth invariant ensures that inactive robots at the end of the previous iteration are at depth $(i-1)d'$ or less, and moving them according to Line 11 can indeed be done within $2(i-1)d'$ rounds. Moreover, the Open Node Coverage invariant ensures that all nodes at depth less than $(i-1)d'$ are closed, and these movements preserve the DFS Open Coverage and Parallel Positions invariants. The Partial Exploration invariant is also preserved since these robots are not located in the sub-tree of their anchor. These $(i-1)d'$

rounds also preserve Anchor Depth and Open Node Coverage invariants as the anchors R of nodes active in the last round of the previous iteration remain their anchor, while other nodes are assigned to one of the anchors in R.

The fact that robots are initially in Parallel DFS Positions in each instance $\mathcal{A}_r(k^*, k', d')$ for $r \in R$ comes from the preservation of the DFS Open Coverage, Parallel Positions, and Partial Exploration invariants at the end of the previous round as the root r was the anchor of robots that are not located at r. Now, as all instances $\mathcal{A}_r(k^*, k', d')$ for $r \in R$ run in disjoint sub-trees, the DFS Open Coverage, Parallel Positions, Partial Exploration, Anchor Depth and Open Node Coverage invariants are also preserved during the rest of the iteration since each $\mathcal{A}_r(k^*, k', d')$ is anchor-based. Similarly, the Inactive Depth invariant is satisfied as its variant in instances $\mathcal{A}_r(k^*, k', d')$ imply that inactive nodes are at depth $(i-1)d' + d' = i \cdot d' \leq d$ at most. The Shallow Activity invariant is preserved as long as at least one instance $\mathcal{A}_r(k^*, k', d')$ is not running deep according to the Shallow Activity invariant for that instance. This means that the number of overall active robots can drop below k^* only when all instances are running deep, implying that all anchors are then at depth $(i-1)d' + d' = i \cdot d'$. Note that the Open Node Coverage invariant then implies that all open nodes are in the sub-trees rooted at the anchors of the robots that were active in the last round. The exploration can thus be reduced to these at most k^* sub-trees as claimed in the description of the divide depth functor.

Finally, the algorithm starts running deep only when all anchors are at depth d and are all closed. This can happen only towards the end of the last iteration when all instances are running deep. The reason is that if an instance is not running deep, it has at least k^* active robots by the Shallow Activity invariant and the termination condition of the inner while loop at Line 15 is not met. The Deep Activity invariant then follows from the fact that instances are running in pairwise disjoint sub-trees and all satisfy the Deep Activity invariant.

This completes the proof that $\mathcal{D}[\mathcal{A}(k^*, k', d'); n_{team}; n_{iter}]$ is correct and that it is an anchor-based exploration algorithm.

The proof for f-shallow efficiency is given in Section 5. ◀

A Topology by Geometrization for Sub-Iterated Immediate Snapshot Message Adversaries and Applications to Set-Agreement

Yannis Coutouly
Laboratoire d'Informatique et des Systèmes - Université Aix-Marseille, France
CNRS, Marseille, France

Emmanuel Godard
Laboratoire d'Informatique et des Systèmes - Université Aix-Marseille, France
CNRS, Marseille, France

Abstract

The Iterated Immediate Snapshot model (IIS) is a central model in the message adversary setting. We consider general message adversaries whose executions are arbitrary subsets of the executions of the IIS message adversary. We present a complete and explicit characterization and lower bounds for solving *set-agreement* for general sub-IIS message adversaries.

In order to have this characterization, we introduce a new topological approach for such general adversaries, closely associating executions to *geometric* simplicial complexes. This way, it is possible to define and explicitly construct a topology directly on the considered sets of executions. We believe this topology by geometrization to be of independent interest and a good candidate to investigate distributed computability in general sub-IIS message adversaries, as this could provide both simpler and more powerful ways of using topology for distributed computability.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases topological methods, geometric simplicial complex, set-agreement

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.15

Funding This work has been funded by ANR project DUCAT – ANR-20-CE48-0006.

1 Introduction

The $k-$set-agreement problem is a standard problem in distributed computing and it is known to be a good benchmark for topological approaches. The $k-$set-agreement problem is a distributed task where processes have to agree on no more than k different initial values. The *set-agreement* problem is the $k-$set agreement problem with $k+1$ processes. A review by Raynal can be found in [23]. Since the seminal works of Herlihy-Shavit, Borowsky-Gafni and Saks Zaharoglou [14, 3, 25], using topological methods has proved very fruitful for distributed computing and for distributed computability in particular. In the shared memory model, the impossibility of wait-free $k-$set agreement for more than $k+1$ processes is one of the crowning achievements of topological methods in distributed computing.

Since those first results, the topological framework has been refined to be presented in a more effective way. In particular, the Iterated Immediate Snapshot model (IIS) is a special message adversary that has been proposed as a central model to investigate distributed computability. In this paper we consider the set-agreement problem in the context of message adversaries defined as subsets of executions of the IIS message adversary.

1.1 Main Contributions

The main contribution is the first complete and explicit characterization of sub-IIS message adversaries for which set-agreement is solvable. We introduce in Section 3 a geometrization mapping *geo* that associates a point in $\mathbb{R}^N$ (with a large enough N) to any execution of IIS_n, the set of IIS executions for $n+1$ processes. The characterization of Th. 26 states that set-agreement is solvable for $\mathcal{M} \subset IIS_n$ if and only if the geometrization of $\mathcal{M}$ has a "hole", *i.e.* $geo(\mathcal{M})$ is a strict subset of the convex hull of $\mathbb{S}^n$, the simplex of dimension n. In Section 4, we describe and prove important properties of the geometrization *geo*. In particular we give a combinatorial description of the sets $geo^{-1}(x)$, for $x \in \mathbb{R}^N$, in Th.25. Interestingly, we show that these sets can have only three possible size: 1, 2 and infinite size. Together with the previous theorem, this gives a explicit and complete characterization of the subsets of the executions of the IIS message adversary for which set-agreement is solvable. We also apply our technique to derive new lower bounds for general message adversaries solving set-agreement.

The *geo* mapping is central to our characterization. The second main contribution is to show that there is a natural topological interpretation of this mapping. Using *geo*, we present in Section 3 a new topology that is defined directly on the set of IIS executions. We believe this topology by geometrization to be of independent interest and a good candidate to investigate distributed computability in general sub-IIS message adversaries, as this could provide both simpler and more powerful ways of using topology for distributed computability of any task.

In order to handle general message adversaries, we consider here simplicial complexes primarily as *geometric* simplicial complexes. The standard chromatic subdivision is the combinatorial topology representation of one round of the Immediate Snapshot model. Its simple and regular structure makes topological reasoning attractive. In this paper, we introduce a new universal algorithm and show its relationship with the standard chromatic subdivision as exposed in the geometric simplicial complex setting. This new algorithm is called the Chromatic Averaging algorithm, it averages with specific weights vectors of $\mathbb{R}^N$ at each node. Running the Chromatic Average Algorithm in the IIS_n model yields a geometric counterpart in $\mathbb{R}^N$ to any given infinite execution of IIS_n. The geometrization mapping $geo(w)$ of an execution $w \in IIS_n$ is defined as the convergence value of running the Chromatic Average Algorithm under execution w.

The topology on the set of executions is then the topology induced from the standard topology in $\mathbb{R}^N$ by the mapping $geo :$ the open sets are pre-images $geo^{-1}(\Omega)$ of the open sets Ω of R^N. The standard euclidean topology of $\mathbb{R}^N$ is simple and well understood, however, since *geo* is not injective, it is necessary to describe so-called "non-separable sets" in order to fully understand the new topology. In topology, two distinct elements x, y are said to be non-separable if for any two neighbourhoods Ω_x of x and Ω_y of y, we have $\Omega_x \cap \Omega_y \neq \emptyset$. In our setting, two executions are non-separable when they have the same image via the mapping *geo*, we call such pre-image sets *geo−classes*. Understanding those sets is central to the characterization of solvability of set-agreement. It is also central to precisely describe the properties of the geometrization topology. So we introduce first the geometrization topology and in Section 4, we investigate the *geo*−classes. In Section 5, we apply our framework to derive the characterization of computability of set-agreement and lower bounds. In the conclusion, we discuss the perspective of possible application of the geometrization topological framework to arbitrary tasks.

1.2 Related Works

In [9], the “two generals problem”, that is the consensus problem for two processes is investigated for arbitrary sub-IIS models by Godard and Perdereau. Given that consensus for two processes is actually set-agreement, the characterization of solvability of set-agreement presented here is a generalization to any number of processes of the results of [9].

One of the most advanced results toward the investigation of general sub-ISS adversary are presented in the work of Kuznetsov and Rieutord [24, 17]. Their adversaries are iterated and are related to so-called affine task. Our work consider more general sub-IIS adversaries, including non-iterated adversaries, but the distributed computability is presented only for the set-agreement task.

In [8], Gafni, Kuznetsov and Manolescu investigate models that are more general subsets of the Iterated Immediate Snapshot model, where the execution sets are closed under a specific relation. We believe our tools can provide a simpler, and less error-prone (see [9, Section 5.1]), framework to investigate distributed computability of sub-IIS models. In particular, their closure relation is nicely interpreted here as exactly the non-separability relation of the geometrization topology.

In a series of works, averaging algorithms to solve relaxed versions of the Consensus problem, including approximate Consensus, have been investigated. In [6], Charron-Bost, Függer, and Nowak have used matrix oriented approaches to show the convergence of different averaging algorithms. We use a similar stochastic matrix technique here to prove the convergence of the Chromatic Average Algorithm. In [7], Függer, Nowak and Schwarz have shown tight bounds for solving approximate and asymptotic Consensus in quite general message adversaries.

In [20], Nowak, Schmid, and Winkler propose knowledge-based topologies for all message adversaries. It is then used to characterize message adversaries that can solve Consensus. The scope of [20] is larger than the scope of this paper, however, note that contrary to those topologies, that are *implicitly* defined by indistinguishability of local knowledge, the geometrization topology here is explicitly defined and fully described by Th. 25. Recently, in [2], Attiya, Castañeda and Nowak presented a corrected version of the general characterisation of [8] in this framework. They also give as application a characterisation for set-agreement based upon terminating subdivisions [2, Thm. 4.2]. We believe the characterisation given in Thm. 26 to be more precise. An interesting open question would be to compare the geometrization topology to the knowledge-based ones defined in [20, 2].

2 Models and Definitions

2.1 Message Adversaries

We introduce and present here our notations. Let $n \in \mathbb{N}$, we consider systems with $n+1$ processes. We denote $\Pi_n = [0, .., n]$ the set of processes. Since sending a message is an asymmetric operation, we will work with directed graphs. We recall the main standard definitions in the following.

We use standard directed graph (or digraph) notations: given G, $V(G)$ is the set of vertices, $A(G) \subset V(G) \times V(G)$ is the set of arcs.

▶ **Definition 1.** *We denote by $\mathcal{G}_n$ the set of directed graphs with vertices in Π_n.*

A dynamic graph $\mathbf{G}$ *is a sequence $G_1, G_2, \cdots, G_r, \cdots$ where G_r is a directed graph with vertices in Π_n. We also denote by $\mathbf{G}(r)$ the digraph G_r. A* message adversary *is a set of dynamic graphs.*

Since that n will be mostly fixed through the paper, we use Π for the set of processes and $\mathcal{G}$ for the set of graphs with vertices Π when there is no ambiguity.

Intuitively, the graph at position r of the sequence describes whether there will be, or not, transmission of some messages sent at round r. A formal definition of an execution under a scenario will be given in Section 2.3.

We will use the standard following notations in order to describe more easily our message adversaries [21]. A sequence is seen as a word over the alphabet $\mathcal{G}$.

▶ **Definition 2.** *Given $A \subset \mathcal{G}$, A^* is the set of all finite sequences of elements of A, A^ω is the set of all infinite ones and $A^\infty = A^* \cup A^\omega$.*

Given $\mathbf{G} \in \mathcal{G}^\omega$, if $\mathbf{G} = \mathbf{HK}$, with $\mathbf{H} \in \mathcal{G}_n^*, \mathbf{K} \in \mathcal{G}_n^\omega$, we say that $\mathbf{H}$ is *a prefix* of $\mathbf{G}$, and $\mathbf{K}$ *a suffix*. $Pref(\mathbf{G})$ denotes the set of prefixes of $\mathbf{G}$. An adversary of the form A^ω is called an *oblivious adversary* or an *iterated adversary*. A word in $\mathcal{M} \subset \mathcal{G}^\omega$ is called a *communication scenario* (or *scenario* for short) of message adversary $\mathcal{M}$. Given a word $\mathbf{H} \in \mathcal{G}^*$, it is called a *partial scenario* and $len(\mathbf{H})$ is the length of this word. The prefix of $\mathbf{G}$ of length r is denoted $\mathbf{G}_{|r}$ (not to be confused with $\mathbf{G}(r)$ which is the r-th letter of $\mathbf{G}$, it the digraph at time r).

The following definitions provide a notion of causality when considering infinite word over digraphs.

▶ **Definition 3** ([5]). *Let $\mathbf{G}$ a sequence $G_1, G_2, \cdots, G_r, \cdots$. Let $p, q \in \Pi$. There is a* journey *in $\mathbf{G}$ at time r from p to q, if there exists a sequence $p_0, p_1, \ldots, p_t \in \Pi$, and a sequence $r \leq i_0 < i_1 < \cdots < i_t \in \mathbb{N}$ where we have*

- *$p_0 = p, p_t = q$,*
- *for each $0 < j \leq t$, $(p_{j-1}, p_j) \in A(G_{i_j})$*

This is denoted $p \overset{r}{\underset{\mathbf{G}}{\rightsquigarrow}} q$. We also say that p is causally influencing q from round r in $\mathbf{G}$.

2.2 Iterated Immediate Snapshot Message Adversary

We say that a graph G has the *Immediacy Property* if for all $a, b, c \in V(G)$, $(a, b), (b, c) \in A(G)$ implies that $(a, c) \in A(G)$. A graph G has the *containment Property* if for all $a, b \in V(G)$, $(a, b) \in A(G)$ or $(b, a) \in A(G)$.

▶ **Definition 4** ([12]). *We set $IS_n = \{G \in \mathcal{G}_n \mid G$ has the Immediacy and Containment properties$\}$. The Iterated Immediate Snapshot message adversary for $n+1$ processes is the message adversary $IIS_n = IS_n^\omega$.*

The Iterated Immediate Snapshot model was first introduced as a (shared) memory model and then has been shown to be equivalent to the above message adversary first as tournaments and iterated tournaments [4, 1], then as this message adversary [12, 13]. See also [22] for a survey of the reductions involved in these layered models.

We show how standard fault environments are conveniently described in our framework.

▶ **Example 5.** Consider a message passing system with $n+1$ processes where, at each round, all messages could be lost. The associated message adversary is $\mathcal{G}_n^\omega$.

▶ **Example 6.** Consider a system with two processes $\{\circ, \bullet\}$ where, at each round, only one message can be lost. The associated message adversary is $\{\circ\!\leftrightarrow\!\bullet, \circ\!\leftarrow\!\bullet, \circ\!\rightarrow\!\bullet\}^\omega$. This is IIS_1.

2.3 Execution of a Distributed Algorithm

Given a message adversary $\mathcal{M}$ and a set of initial configurations $\mathcal{I}$, we define what is an execution of a given algorithm $\mathcal{A}$ subject to $\mathcal{M}$ with initialization $\mathcal{I}$. An execution is constituted of an initialization step, and a (possibly infinite) sequence of rounds of messages exchanges and corresponding local state updates. When the initialization is clear from the context, we will use *scenario* and *execution* interchangeably.

An execution of an algorithm $\mathcal{A}$ under scenario $w \in \mathcal{M}$ and initialization $\iota \in \mathcal{I}$ is the following. This execution is denoted $\iota.w$. First, ι affects an initial state to all processes of Π.

A round is decomposed in 3 steps: sending, receiving, updating the local state. At round $r \in \mathbb{N}$, messages are sent by the processes using the `SendAll()` primitive. The fact that the corresponding receive actions, using the `Receive()` primitive, will be successful depends on $G = w(r)$, G is called the *instant graph* at round r.

Let $p, q \in \Pi$. The message sent by p is received by q on the condition that the arc $(p, q) \in A(G)$. Then, all processes update their state according to the received values and $\mathcal{A}$. Note that, it is usually assumed that p always receives its own value, that is $(p, p) \in A(G)$ for all p and G.

Let $w \in \mathcal{M}, \iota \in \mathcal{I}$. Given $u \in Pref(w)$, we denote by $\mathbf{s}_p(\iota.u)$ the state of process p at the $len(u)$-th round of the algorithm $\mathcal{A}$ under scenario w with initialization ι. This means that $\mathbf{s}_p(\iota.\varepsilon)$ represents the initial state of p in ι, where ε denotes the empty word.

A task is given by a set $\mathcal{I}$ of initial configurations, a set of output values Out and a relation Δ, the specification, between initial configurations and output configuration[1]. We say that a process *decides* when it outputs a value in Out. Finally and classically,

▶ **Definition 7.** *An algorithm $\mathcal{A}$ solves a Task $(\mathcal{I}, Out, \Delta)$ for the message adversary $\mathcal{M}$ if for any $\iota \in \mathcal{I}$, any scenario $w \in \mathcal{M}$, there exist u a prefix of w such that the states of the processes $out = (\mathbf{s}_0(\iota.u), \dots, \mathbf{s}_n(\iota.u))$ satisfy the specification of the task, ie $\iota \Delta out$.*

3 A Topology by Geometrization

In this paper we present a new topological approach for investigating distributed computability. It extends the known simplicial complexes-based known method for finite executions to infinite executions without considering infinite additional complexes like in [8]. This enables to define directly a topology on the set of executions of the standard Iterated Immediate Snapshot model IIS_n.

3.1 Combinatorial Topology Definitions

3.1.1 Geometric Simplicial Complexes

Before giving the definition of the geometrization topology in Sect. 3.2.2, we state the definition of simplicial complexes, but not first as abstract complex, as is usually done in distributed computing, but primarily as geometrical objects in $\mathbb{R}^N$. This is the reason we call this definition the geometrization topology. Intuitively we will associatea point in $\mathbb{R}^N$ to any execution via a geometrization mapping *geo*. The geometrization topology is the topology induced by geo^{-1} from the standard topology in $\mathbb{R}^N$. This also makes *geo*

[1] Note that the standard definition in the topological setting involves carrier map that we do not consider here for we will consider only one specific task, the Set Agreement problem.

continuous by definition. In the standard approach, geometric simplices are also used but they are introduced as geometric realizations of the abstract simplicial complexes. As will be seen later, when dealing with infinite complexes, the standard topology of these simplices does not enable to handle the computability of distributed tasks since we will need to define an other topology. We show that the topology on infinite complexes, as defined in standard topology textbook, is different from the one we show here to be relevant for distributed computability. Note that to be correctly interpreted, the topology we construct is on the set of infinite executions, not on the complexes corresponding to finite executions.

The following definitions are standard definitions from algebraic topology [19]. We fix an integer $N \in \mathbb{N}$ for this part. We denote $||x||$ the euclidean norm in $\mathbb{R}^N$. For a bounded subset $X \subset \mathbb{R}^n$, we denote $diam(X)$ its diameter.

▶ **Definition 8.** *Let $n \in \mathbb{N}$. A finite set $\sigma = \{x_0, \ldots, x_n\} \subset \mathbb{R}^N$ is called a* simplex *of dimension n if the vectors $\{x_1 - x_0, \ldots, x_n - x_0\}$ are linearly independent. We denote by $|\sigma|$ the convex hull of σ.*

▶ **Definition 9** ([19])**.** *A* simplicial complex *is a collection C of* simplices *such that:*
(a) *If $\sigma \in C$ and $\sigma' \subseteq \sigma$, then $\sigma' \in C$,*
(b) *If $\sigma, \tau \in C$ and $|\sigma| \cap |\tau| \neq \emptyset$ then there exists $\sigma' \in C$ such that $|\sigma| \cap |\tau| = |\sigma'|$, $\sigma' \subset \sigma, \sigma' \subset \tau$.*

We denote $\wr C \wr = \bigcup_{S \in C} |S|$, this is the *geometrization of C*.

Note that these definitions do not require complexes to be a finite collection of simplices. The simplices of dimension 0 (singleton) of C are called vertices, we denote $V(C)$ the set of vertices of C. A complex is pure of dimension n if all maximal simplices are of dimension n. In this case, a simplex of dimension $n-1$ is called a facet. The *boundary* of a simplex $\sigma = \{x_0, \ldots, x_n\}$ is the pure complex $\bigcup_{i \in [0,n]} \{x_j \mid j \in [0,n], i \neq j\}$ of dimension $n-1$. It is denoted $\delta(\sigma)$, it is the union of the facets of σ.

Let A and B be simplicial complexes. A map $f \colon V(A) \to V(B)$ defines a *simplicial map* if it preserves the simplices, *i.e.* for each simplex σ of A, the image $f(\sigma)$ is a simplex of B. By linear combination of the barycentric coordinates, f extends to the linear simplicial map $f \colon \wr A \wr \to \wr B \wr$, which is continuous. See [19, Lemma 2.7].

We also have colored simplicial complexes. These are simplicial complexes C together with a function $\chi : V(C) \to \Pi$ such that the restriction of χ on any maximal simplex of C is a bijection. A simplicial map that preserves colors is called chromatic.

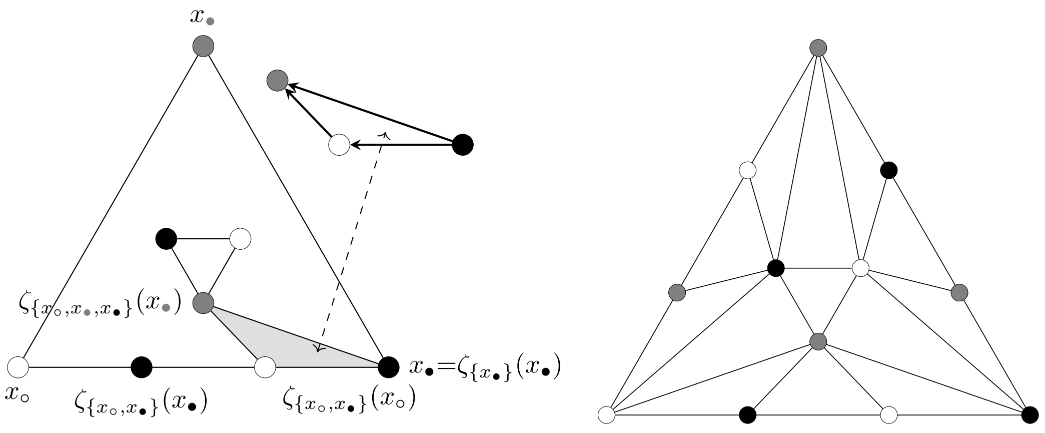
Finally, $\mathbb{S}.$ will denote "the" simplex of dimension n. Through this paper we assume a fixed embedding in $\mathbb{R}^N$ for $\mathbb{S}. = (x_0^*, \ldots, x_n^*)$. We will also assume that its diameter $diam(\mathbb{S}.)$ is 1.

3.1.2 The Standard Chromatic Subdivision

Here we present the standard chromatic subdivision, [12] and [15], as a geometric complex. We start with subdivisions and chromatic subdivisions.

▶ **Definition 10** (Subdivision)**.** *A subdivision of a simplex S is a simplicial complex C with $\wr C \wr = |S|$.*

▶ **Definition 11** (Chromatic Subdivision)**.** *Given $(S, \mathcal{P})$ a chromatic simplex, a chromatic subdivision of S is a chromatic simplicial complex $(C, \mathcal{P}_C)$ such that*
- *C is a subdivision of S*(i.e.*$\wr C \wr = |S|$),*
- *$\forall x \in V(S), \mathcal{P}_C(x) = \mathcal{P}(x)$.*


$x_{\bullet}$
$\zeta_{\{x_\circ,x_\bullet,x_\bullet\}}(x_\bullet)$
$x_\bullet = \zeta_{\{x_\bullet\}}(x_\bullet)$
$x_\circ$
$\zeta_{\{x_\circ,x_\bullet\}}(x_\bullet)$
$\zeta_{\{x_\circ,x_\bullet\}}(x_\circ)$

(a) Images by ζ_V for various $V \subset \{\circ, \bullet, \bullet\}$. **(b)** $Chr(\mathbb{S}^2)$.

Figure 1 Standard chromatic subdivision construction for dimension 2. On the left, the association between an instant graph of IS_2 (top) and a simplex of $Chr(\mathbb{S}^2)$ (grey area) is illustrated.

Note that it is not necessary to assume $V(S) \subset V(C)$ here, since the vertices of the simplex S being extremal points, they are necessarily in $V(C)$.

We start by defining some geometric transformations of simplices (here seen as sets of points). The choice of the coefficients will be justified later.

▶ **Definition 12.** *Consider a set $V = (y_0, \dots, y_d)$ of size $d+1$ in $\mathbb{R}^N$. We define the function $\zeta_V : V \longrightarrow \mathbb{R}^N$ by, for all $j \in [0, d]$*

$$\zeta_V(y_j) = \frac{1 - \frac{d}{2d+1}}{d+1} y_j + \sum_{i \neq j} \frac{1 + \frac{1}{2d+1}}{d+1} y_i$$

We now define in a geometric way the *standard chromatic subdivision* of colored simplex $(S, \mathcal{P})$, where $S = \{x_0, x_1, \dots, x_n\}$ and $\mathcal{P}(x_i) = i$.

The chromatic subdivision $\mathrm{C}hr(S)$ for the colored simplex $S = \{x_0, \dots, x_n\}$ is a simplicial complex defined by the set of vertices $V(\mathrm{C}hr(S)) = \{\zeta_V(x_i) \mid i \in [0, n], V \subset V(S), x_i \in V\}$.

For each pair (i, V), $i \in [0, n]$ and $V \subset V(S)$, there is an associated vertex y of $\mathrm{C}hr(S)$, and conversely each vertex has an associated pair. The *color* of (i, V) is i. The set V is called the *view*. We define Φ the following *presentation* of a vertex y, $\Phi(y) = (\mathcal{P}(y), V_y)$ where $\mathcal{P}(y) = i$ and $V_y = V$.

The simplices of $\mathrm{C}hr(S)$ are the set of $d+1$ points $\{\zeta_{V_0}(x_{i_0}), \cdots, \zeta_{V_d}(x_{i_d})\}$ where

- there exists a permutation π on $[0, d]$ such that $V_{\pi(0)} \subseteq \cdots \subseteq V_{\pi(d)}$,
- If $i_j \in \mathcal{P}(V_\ell)$ then $V_j \subset V_\ell$.

In Fig. 1, we present the construction for $Chr(\mathbb{S}^2)$. For convenience, we associate $\circ, \bullet, \bullet$ to the processes $0, 1, 2$ respectively. In Fig. 1a, we consider the triangle $x_\circ, x_\bullet, x_\bullet$ in $\mathbb{R}^2$, with $x_\circ = (0, 0)$, $x_\bullet = (1, 0)$, $x_\bullet = (\frac{1}{2}, \frac{\sqrt{3}}{2})$. We have that $\zeta_{\{x_\circ, x_\bullet\}}(x_\bullet) = (\frac{1}{3}, 0)$, $\zeta_{\{x_\circ, x_\bullet\}}(x_\circ) = (\frac{2}{3}, 0)$ and $\zeta_{\{x_\circ, x_\bullet, x_\bullet\}}(x_\bullet) = (\frac{1}{2}, \frac{\sqrt{3}}{10})$. The relation between instant graph (top) and simplex $\left\{(\frac{2}{3}, 0), (1, 0), (\frac{1}{2}, \frac{\sqrt{3}}{5})\right\}$ (grey area) is detailed in the following section.

In the following, we will be interested in iterations of $Chr(\mathbb{S}^n, \mathcal{P})$. The last property of the definition of chromatic subdivision means with we can drop the C index in the coloring of complex C and use $\mathcal{P}$ to denote the coloring at all steps. From its special role, it is called the *process color* and we drop $\mathcal{P}$ in $Chr(S, \mathcal{P})$ using in the following $Chr(S)$ for all simplices S of iterations of $Chr(\mathbb{S}^n)$.

In [16], Kozlov showed how the standard chromatic subdivision complex relates to Schlegel diagrams (special projections of cross-polytopes), and used this relation to prove the standard chromatic subdivision was actually a subdivision.

In [12, section 3.6.3], a general embedding in R^n parameterized by $\epsilon \in \mathbb{R}$ is given for the standard chromatic subdivision. The geometrization here is done choosing $\epsilon = \frac{d}{2d+1}$ in order to have “well balanced” drawings.

3.2 Encoding Iterated Immediate Snapshots Configurations

3.2.1 Algorithms in the Iterated Immediate Snapshots Model

It is well known, see *e.g.* [12, Chap. 3&4, Def. 3.6.3], that each maximal simplex $S = \{\zeta_{V_0}(x_{i_0}), \cdots, \zeta_{V_n}(x_{i_n})\}$ from the chromatic subdivision of $\mathbb{S}^n$ can be associated with a graph of IS_n denoted $\Theta(S)$. We have $V(\Theta(S)) = \Pi_n = [0, n]$ and set $\Theta(\zeta_{V_j}(x_{i_j})) = \mathcal{P}(x_{i_j})$.The arcs are defined using the representation Φ of points, $A(\Theta(S)) = \{(i, j) \mid i \neq j, V_i \subseteq V_j\}$. The mapping θ will denote Θ^{-1}. We can transpose this presentation to an averaging algorithm called the *Chromatic Average* Algorithm presented in Algorithm 1.

Algorithm 1 The Chromatic Average Algorithm for process i.

```
$x \leftarrow x_i^*$;
Loop forever
    SendAll($(i, x)$);
    $V$ ←Receive() // set of all received messages;
    $d \leftarrow sizeof(V) - 1$ // $i$ received $d+1$ messages including its own ;
    $x = \frac{1-\frac{d}{2d+1}}{d+1}x + \sum_{(j,x_j)\in V, j\neq i} \frac{1+\frac{1}{2d+1}}{d+1}x_j$;
```

Executing one round of the loop in Chromatic Average for instant graph G, the state of process i is $x'_i = \zeta_{V_i}(x_i^*)$, where V_i is the view of i on this round, that is the set of (j, x_j) it has received; with $\Theta(\{\zeta_{V_0}(x_0^*), \cdots, \zeta_{V_n}(x_n^*)\}) = G$. See eg. in Fig. 1a, the simplex of the grey area corresponds to the ordered sequence of views $\{x_\bullet\} \subset \{x_\bullet, x_\circ\} \subset \{x_\bullet, x_\circ, x_{\bullet}\}$, associated to the directed graph depicted at the top right. Adjacency for a given i corresponds to the smallest subset containing x_i. By iterating, the chromatic subdivisions $Chr^r(\mathbb{S}^n)$ are given by the global state under all possible r rounds of the Chromatic Average Algorithm. Finite rounds give the Iterated Chromatic Subdivision (hence the name). This is an algorithm that is not meant to terminate (like the full information protocol). The infinite runs are used below to define a topology on IIS_n.

The Chromatic Average algorithm is therefore the geometric counterpart to the Full Information Protocol that is associated with Chr [12]. In particular, any algorithm can be presented as the Chromatic Average together with a terminating condition and an output function of x.

This one round transformation for the canonical $\mathbb{S}^n$ can actually be done for any simplex S of dimension n of $\mathbb{R}^N$. For $G \in IS_n$, we denote $\mu_G(S)$ the geometric simplex that is the image of S by one round of the Chromatic average algorithm under instant graph G.

The definitions of the previous section can be considered as mostly textbook (as in [12]), or folklore. To the best of our knowledge, the Chromatic Average Algorithm, as such, is new, and there is no previous complete proof of the link between the Chromatic Average Algorithm and iterated standard chromatic subdivisions. However, one shall remark that people are, usually, actually *drawing* standard chromatic subdivisions using the Chromatic Average Algorithm.

3.2.2 A Topology for IIS_n

Let $w \in IIS_n$, $w = G_1G_2\cdots$. For the prefix of w of size r, S a simplex of dimension n, we define $geo(w_{|r})(S) = \mu_{G_r} \circ \mu_{G_{r-1}} \circ \cdots \circ \mu_{G_1}(S)$. Finally, we set $geo(w) = \lim_{r\longrightarrow\infty} geo(w_{|r})(\mathbb{S}^n)$. We prove in Section A.1 that this actually converges.

We define the *geometrization topology* on the space IIS_n by considering as open sets the sets $geo^{-1}(\Omega)$ where Ω is an open set of $\mathbb{R}^N$. A collection of sets can define a topology when any union of sets of the collection is in the collection, and when any finite intersection of sets of the collection is in the collection. This is straightforward for a collection of inverse images of a collection that satisfies these properties.

A neighbourhood for point x is an open set containing x. In topological spaces, a pair of distinct points x, y is called *non-separable* if there does not exist two disjoint neighbourhoods of these points. The pre-images $geo^{-1}(x)$ that are not singletons are *non-separable sets*. We will see that we always have non-separable sets and that they play an important role for task solvability.

Subset of IIS_n will get the subset topology, that is , for $\mathcal{M} \subseteq IIS_n$, open sets are the sets $geo^{-1}(\Omega) \cap \mathcal{M}$ where Ω is an open set of $\mathbb{R}^N$. We set $\wr\mathcal{M}\wr = geo(\mathcal{M})$ the geometrization of $\mathcal{M}$.

Note that the geometrization should not be confused with the standard *geometric realization*. They are the same at the set level but not at the topological level, see in Section A.2. At times, in order to emphasize this difference, for a simplex $S \subset \mathbb{R}^N$, we will also use $\wr S\wr$ instead of $|S|$. The *geometrization* of C, denoted $\wr C\wr$, that is the union of the convex hulls $|\sigma|$ of the simplices σ of C, is endowed with the standard topology from $\mathbb{R}^N$. We also note this topological space as $\wr C\wr$.

4 Geometrization Equivalence

As will be be shown later, the geometrization has a crucial role in order to understand the relationship between sets of possible executions and solvability of distributed tasks. In this section, we describe more precisely the pre-images sets, that is subsets of IIS_n of the form $geo^{-1}(x)$ for $x \in |\mathbb{S}^n|$. In particular, we will get a description of the non-separable sets of execution.

4.1 Definitions

We say that two executions $w, w' \in IIS_n$ are *geo-equivalent* if $geo(w) = geo(w')$. The set of all w' such that $geo(w) = geo(w')$ is called the equivalence class of w. Since the topology we are interested in for $\wr IIS_n\wr$ is the one induced by the standard separable space $\mathbb{R}^N$ via the geo^{-1} mapping, it is straightforward to see that non-separable sets are exactly the geo-equivalence classes that are not singletons. In this section, we describe all equivalence classes and show that there is a finite number of possible size for these sets.

We define the sets $Solo(P)$, that correspond to subsets of instant graphs where the processes in $P \subset \Pi$ have no incoming message from processes outside of P. We have $Solo(\Pi) = IS_n$.

▶ **Definition 13.** *In the complex* $\mathrm{Chr}(\mathbb{S}^n)$*, with* $P \subset \Pi$*,* $Solo(P)$ *is the set of simplices* $T \in \mathrm{Chr}(\mathbb{S}^n)$ *such that* $\forall (p, q) \in A(\Theta(T)), q \in P \Rightarrow p \in P$.

We denote by $\mathcal{K}_\Pi$ the instant graph that is complete on Π. An execution w is said *fair* for P, $w \in Fair(P)$, if $w \in Solo(P)^\omega$ and for all $p, q \in P$, $\forall r \in \mathbb{N}$, we have $p \overset{r}{\underset{w}{\rightsquigarrow}} q$. Fairness for P means that processes in P are only influenced by processes in P, and that any process always influence other processes infinitely many times. We have the equivalent, and constructive definition:

▶ **Proposition 14.** *Let $w \in Solo(P)^\omega$. An execution w is $Fair$ for P if and only if w has no suffix in $\bigcup_{Q\neq\emptyset, Q\subsetneq P} Solo(Q)^\omega$.*

Proof. Assume we have a suffix s for w in $Solo(Q)^\omega$ with $Q \subsetneq P$.Let $p \in P \setminus Q$ and r the starting index of the suffix. Then $\forall q \in Q$, we must have $p \overset{r}{\underset{w}{\rightsquigarrow}} q$ by definition of fairness for P. Denote q_0 the first element of Q to be causally influenced by p at some time $t \geq r$. So q_0 receive a message from some $p' \in \Pi$, $p' \neq q_0$ at time t. Since $s \in Solo(Q)^\omega$, this means that q_0 can only receive message from processes in Q. Hence $p' \in Q$ and p' was influenced by p at time $t-1$. A contradiction with the minimality of q_0. So w is not in $Fair(P)$.

Conversely, assume that $w \notin Fair(P)$. Then $\exists p, q \in P, \exists s, \forall r \geq s, \neg p \overset{r}{\underset{w}{\rightsquigarrow}} q$. We set Q as the set of processes that causally influence q for all $r \geq s$. We have $p \notin Q$ so $Q \subsetneq P$. We denote s_0, a time at which no process of $\Pi \setminus Q$ influence a process in Q. By construction, the suffix at step s_0 is in $Solo(Q)^\omega$. ◀

4.2 First Results on Geometrization

We start by presenting a series of results about geometrization. Lemma 35 gives the following immediate corollaries.

▶ **Corollary 15.** *Let w a run in IIS_n, then $\forall r \in \mathbb{N}$, $geo(w) \in |geo(w_{|r})(\mathbb{S}^n)|$.*

▶ **Proposition 16.** *Let w, w' two geo-equivalent runs in IIS_n, then $\forall r \in \mathbb{N}$, $geo(w_{|r})(\mathbb{S}^n) \cap geo(w'_{|r})(\mathbb{S}^n) \neq \emptyset$.*

Proof. The intersection of the geometrizations $|geo(w_{|r})(\mathbb{S}^n)|$ and $|geo(w'_{|r})(\mathbb{S}^n)|$ contains at least $geo(w)$ by previous corollary. Since the simplices $geo(w_{|r})(\mathbb{S}^n)$ and $geo(w'_{|r})(\mathbb{S}^n)$ belong to the complex $\mathrm{C}hr^r(\mathbb{S}^n)$, they also intersect as simplices. ◀

▶ **Proposition 17.** *Let S a maximal simplex of the chromatic subdivision $\mathrm{C}hr\mathbb{S}^n$ that is not $\theta(\mathcal{K}(\Pi))$. Then there is $P \subsetneq \Pi$ such that $\Theta(S) \in Solo(P)$.*

Conversely we can describe $Solo(P)$ more precisely. We denote by $\delta(\mathbb{S}^n, P)$ the sub-simplex of $\mathbb{S}^n$ corresponding to $P \subset \Pi$. This is the boundary relative to P in $\mathbb{S}^n$, and we have that $\bigcup_{P\subsetneq\Pi} \delta(\mathbb{S}^n, P) = \bigcup_{p\in\Pi} \delta(\mathbb{S}^n, \pi \setminus p) = \delta(\mathbb{S}^n)$.

▶ **Proposition 18** (Boundaries of $\mathrm{C}hr$ are $Solo$)**.** *Let P a subset of Π. Then $Solo(P) = \{S \mid S$ a maximal simplex of $\mathrm{Chr}(\mathbb{S}^n), |S| \cap |\delta(\mathbb{S}^n, P)| \neq \emptyset\}$.*

Proof. Denote q such that $\Pi = P \cup \{q\}$. Then by construction, $Solo(P)$ corresponds exactly to the simplex where the processes in P do not receive any message from q, ie the simplex intersecting the boundary $\delta(\mathbb{S}^n, P)$. ◀

For a given size s of P, the $Solo$ sets are disjoint, however this does not form a partition of $\mathrm{C}hr$. Finally, by iterating the previous proposition, the boundaries of $\mathbb{S}^n$ are described by $Solo(P)^\omega$.

▶ **Proposition 19.** *Let P a subset of Π. We have $\wr Solo(P)^\omega \wr = |\delta(\mathbb{S}^n, P)|$.*

We can now state the main result that links geometrically fair executions and corresponding simplices: in a fair execution, the corresponding simplices, that are included by convexity, have to eventually be strictly included in the interior.

▶ **Proposition 20** (Geometric interpretation of $Fair$)**.** *Let w an execution that is Fair for Π, then $\forall s \in \mathbb{N}, \exists r > s \in \mathbb{N}$, such that $\delta(geo(w_{|s})(S)) \cap geo(w_{|r})(S) = \emptyset$.*

Proof. Let $s \in \mathbb{N}$, and an execution w. We denote $T = geo(w_{|s}$. Consider a process $p \in \Pi$, for all process $q \neq p$ we have $p \overset{s}{\underset{w}{\rightsquigarrow}} q$. Since w is fair in Π, we can consider $r > s$ the time at which p is influencing all q from round s. At his step, for all $q \neq p$, the barycentric coordinate of the vertex of $geo(w_{|r}(S)$ of colour q relative to the vertex of $geo(w_{|r}(S)$ of colour p is strictly positive. This means that $geo(w_{|r}(S)$ does not intersect $\delta(T, \Pi \setminus p)$.

Since w is fair in Π, we can repeat this argument for any $p \in \Pi$. We denote the r^* the maximal such r and since $\bigcup_{p\in\Pi} \delta(T, \Pi \setminus p) = \delta(T)$, we have that $\delta(geo(w_{|s})(S)) \cap geo(w_{|r^*})(S) = \emptyset$. ◀

4.3 A Characterization of Geo-Equivalence

We start by simple, but useful, sufficient conditions about the size of geo-classes.

▶ **Proposition 21** ($Fair(\Pi)$ is separable)**.** *Let $w \in IIS_n$, denote Σ the geo-class of w. If w is $Fair$ on Π, then $\#\Sigma = 1$.*

Proof. Let $w' \in \Sigma$. We will show that w' shares all prefixes of w. Let $r \in \mathbb{N}$. From Prop. 20 and Lemma 35, we get that $geo(w)$ does not belong to the boundary of $geo(w_{|r})(S)$, nor to the boundary of $geo(w'_{|r})(S)$. Assume that w and w' have not the same prefix of size r, that is $geo(w_{|r})(S) \neq geo(w'_{|r})(S)$. From Prop. 16 $geo(w_{|r})(S)$, $geo(w'_{|r})(S)$ have to intersect (as simplices), and since they are different, they can intersect only on their boundary. This means that $geo(w)$ would belong to the boundary, a contradiction.

So they have the same prefixes and $w' = w$. ◀

▶ **Proposition 22** (Infinite Cardinal)**.** *Let $n \geq 2$. Let w, w' two distinct executions such that $geo(w) = geo(w')$ and there exist $s \in \mathbb{N}$ such that $\forall r > s \exists T geo(w_{|r})(S) \cap geo(w'_{|r})(S) = T$ with T a simplex of dimension $k \leq n - 2$. Then, the geo-equivalence class of w is of infinite size.*

Proof. Let w, w' two executions with $geo(w) = geo(w')$ and $\forall r > s$, $geo(w_{|r})(S) \cap geo(w'_{|r})(S) = T$ with T of dimension $k \leq n - 2$. Denote P the colors of T. Since $k \leq n - 2$, we have at least $p_1 \neq p_2 \in \Pi \setminus P$. The suffix at length s of w is in $Solo(P)$.

Hence, for the processes in P, when running in w or w', it is not possible to distinguish these 3 cases about the induced subgraph by $\{p_1, p_2\}$ in the instant graphs: $p_1 \leftarrow p_2$, $p_1 \leftrightarrow p_2$ and $p_1 \rightarrow p_2$.

So $\forall r > s$, we have 3 possible ways of completing what is happening on the induced subgraph by processes in P in $G \in Solo(P)$. So we have infinitely many different executions, the cardinality of the geo-class of w is infinite. ◀

Let's consider the remaining cases. Let $w \in IIS_n$, denote Σ the geo-class of w.

▶ **Proposition 23** (Boundaries of $\mathbb{S}^n$ are separable)**.** *If w is $Fair$ on $\Pi \setminus \{p\}$ for some $p \in \Pi$ then $\#\Sigma = 1$.*

Proof. We denote $Q = \Pi \setminus \{p\}$. We apply Prop. 21 for $n-1$ to w' the restriction of w to the set of processes Q (this is possible by definition of $Fair$: no process of Q receives message from outside of Q). Since w' satisfies the condition for Prop. 21 (by definition of Fair), which means that the geo-class of w' is of size 1.

Since there is only one unique way of completing an execution restricted to Q to one in $Solo(Q)$ (adding $(q,p), \forall q \in Q$), we get that there is only w in the equivalence class. ◀

A suffix of a word w is *strict* if it is not equal to w.

▶ **Proposition 24.** *If w has only a strict suffix that is $Fair$ on $\Pi \setminus \{p\}$ for some $p \in \Pi$ then $\#\Sigma = 2$.*

Proof. We denote $Q = \Pi \setminus \{p\}$. We can write $w = uav$ where $u \in IS_n^*$, $a \in IS_n$ and v is Fair on Q but av is not. We can choose u such that u has the shortest length.

We consider w' such that $geo(w') = geo(w)$. Let r be the length of ua. We denote by T the facet of $geo(ua)(\mathbb{S}^n)$ with colors Q. Since v is Fair for Q, we can apply to $v_{|Q}$ the restriction of v to Q Prop. 20. So $geo(w)$ is not on the boundaries of T which means, from Prop. 16, that either $geo(w'_{|r})(\mathbb{S}^n) = geo(w_{|r})(\mathbb{S}^n)$ either $geo(w'_{|r})(\mathbb{S}^n) \cap geo(w_{|r})(\mathbb{S}^n) = T$.

In both cases, we can apply Prop. 21 to v' the restriction of w to Q. Which means that there is only one restricted execution in Q. Since there is only one way to complete to p, there are as many elements in the class that simplices at round r that include T. Since we have a subdivision, we have exactly two simplices sharing the facet T.

In the first case, this means that $w'_{|r} = ua$ and $w = w'$.

In the second case, we have that $w'_{|r} = ub$ for some $b \neq a$. We remark that if $w'_{|r-1} \neq u$ this would contradict the minimality of u. Indeed, the prefixes of length $r-1$ are different, this means that av is Fair for Q. ◀

Using these previous propositions, and remarking that for any w, there exists P such that w has a suffix in $Fair(P)$, we can now present our main result regarding the complete classification of geo-equivalence classes. Let $n \in \mathbb{N}$ and Σ a geo-equivalence class on $\mathbb{S}^n$. Then there are exactly 3 cardinals that are possible for Σ (only 2 when $n = 1$, the case of [9]):

▶ **Theorem 25.** *Let $w \in IIS_n$, denote Σ the geo-class of w.*
$\mathcal{C}_1$: If w is $Fair$ on Π or on $\Pi \setminus \{p\}$ for some $p \in \Pi$, then $\#\Sigma = 1$;
$\mathcal{C}_2$: w has only a strict suffix that is $Fair$ on $\Pi \setminus \{p\}$ for some $p \in \Pi$ then $\#\Sigma = 2$;
$\mathcal{C}_\infty$: otherwise Σ is infinite.

5 The Set-Agreement Problem

For all n, the *set-agreement problem* is defined by the following properties [18]. Given initial *init* values in $[0, n]$, each process outputs a value such that

Agreement the size of the set of output values is at most n,
Validity the output values are initial values of some processes,
Termination All processes terminates.

We will consider in this part sub-IIS message adversaries $\mathcal{M}$, that is $\mathcal{M} \subseteq IIS_n$. It is well known that set-agreement is impossible to solve on IIS_n, we prove the following characterization.

▶ **Theorem 26.** *Let $\mathcal{M} \subset IIS_n$. It is possible to solve Set-Agreement on $\mathcal{M}$ if and only if $\wr\mathcal{M}\wr \neq |\mathbb{S}^n|$.*

5.1 Impossibility Result

On the impossibility side, we will prove a stronger version with non-silent algorithms. An algorithm is said to be *non-silent* if it sends message forever. Here, this means that a process could have decided a value while still participating in the algorithm.

▶ **Theorem 27.** *Let $\mathcal{M} \subset IIS_n$. If $\wr\mathcal{M}\wr = \wr IIS_n\wr = |\mathbb{S}^n|$ then it is not possible to solve Set-Agreement on $\mathcal{M}$, even with a non-silent algorithm.*

We will need the following definition from combinatorial topology.

▶ **Definition 28** (Sperner Labelling)**.** *Consider a simplicial complex C that is a subdivision of a chromatic simplex (S, χ). A labelling $\lambda : V(C) \longrightarrow \Pi$ is a* Sperner labelling *if for all $x \in V(C)$, for all $\sigma \subset S$, we have that $x \in |\sigma| \Rightarrow \lambda(x) \in \chi(\sigma)$.*

▶ **Lemma 29** (Sperner Lemma [26])**.** *Let a simplicial complex C that is a subdivision of a chromatic simplex (S, χ) with Sperner labelling λ. Then there exists $\sigma \in C$, such that $\lambda(\sigma) = \Pi$.*

A simplex σ with labelling using all Π colors is called *panchromatic.*

Proof of Theorem 27. By absurd, we assume there is a non-silent algorithm $\mathcal{A}$ (in full information protocol form) solving set-agreement on $\mathcal{M}$. We run the algorithm on initial inputs $init(i) = i$. We translate the full information protocol to the chromatic average, non-silent form: the initial value of i is x_i^*; when the decision value is given, we still compute and send the chromatic average forever. We can also assume a "normalized" version of the algorithm: when a process receives a decision value from a neighbour, it will decide instantly on this value. Such a normalization does not impact the correctness of the algorithm since set-agreement is a colorless task.

The proof will use the Sperner Lemma with labels obtained from the eventual decision value of the algorithm. However it is not possible to use directly the Sperner Lemma for the "full subdivision under $\mathcal{M}$" (which we won't define), since this subdivision could be infinite. The following proof will use König Lemma to get an equivalent statement.

Given $t \geq 0$, we consider $Chr^t(\mathbb{S}^n)$ under our algorithm with initial values $init(i) = i$. For any vertex, we define the following labelling λ_t: if the process i has not terminated at time t with state $x \in V(Chr^t(\mathbb{S}^n))$, then the Sperner label $\lambda_t(x) = i$, otherwise it is the decided value. Since the decided value depends only on the local state, the label of a vertex at time t is independent of the execution leading to it. The goal of the following is to show that there is an entire geo-equivalence class that does not belongs to $\mathcal{M}$.

By Integrity property, we have that the value decided on a face of $\mathbb{S}^n$ of processes $i_1, \ldots, i_n$, ie for $Solo(i_1, \ldots, i_n)^\omega$ are taken in $i_1, \ldots, i_n$. From Prop. 19, at any t, this labelling defines therefore a Sperner labelling of a (chromatic) subdivision of S.

We consider the set $\mathcal{S}$ of all simplices S of dimension n of $Chr^t(\mathbb{S}^n)$, for all t. For a given t, from Sperner Lemma, there is at least a simplex of $Chr^t(\mathbb{S}^n)$, that is panchromatic. There is therefore an infinite number of simplices S that are panchromatic in $\mathcal{S}$. We consider now $\mathcal{T} \subset \mathcal{S}$, the set of simplices $T \in \mathcal{S}$ such that there is an infinite number of panchromatic simplex S such that $|S| \subset |T|$. Note that T needs not be panchromatic. Since the number of simplices of $Chr^t(\mathbb{S}^n)$ is finite for a given t, there is at least one simplex of $Chr^t(\mathbb{S}^n)$ that is in $\mathcal{T}$. Therefore the set $\mathcal{T}$ is infinite.

We build a rooted-tree structure over $\mathcal{T}$: the root is $\mathbb{S}^n$ (indeed it is in $\mathcal{T}$), the parent-child relationship between T and T' is defined when $T' \in Chr(T)$. We have an infinite tree with finite branching. By König Lemma, we have an infinite simple path from the root. We denote

T_t the vertex at level t of this path. We have $|T_{t+1}| \subset |T_t|$ and $(T_t)_{t\in\mathbb{N}}$ converges (same argument as the end of Section A.1) to some $y \in |\mathbb{S}^n|$. The increasing prefixes corresponding to T_t define an execution w of IIS_n.

We will now consider two different cases, not on the fact whether or not, $w \in \mathcal{M}$, but on the result of $\mathcal{A}$ on execution w.

For first case, assume that algorithm $\mathcal{A}$ has eventually decided on all processes on run w at some time t_0. Since it could be that $w \notin \mathcal{M}$, we cannot conclude yet. But since all processes have decided, they do not change their label in subsequent steps. By definition, $T_{t_0} = geo(w_{|t_0}(\mathbb{S}^n)$ contains an infinite number of panchromatic simplices, *i.e.* at least one. So the simplex $geo(w_{|t_0}(\mathbb{S}^n)$ is panchromatic. Hence any run with prefix $w_{|t_0}$ cannot be in $\mathcal{M}$, since $\mathcal{A}$ solves set-agreement on $\mathcal{M}$. Therefore $w' = w_{|t_0}\mathcal{K}_\Pi^\omega$ (where $\mathcal{K}_\Pi$ is the complete graph), is a fair execution that does not belong to $\mathcal{M}$. Its entire geo-equivalence class, which is a singleton, is not in $\mathcal{M}$.

The second case is when algorithm $\mathcal{A}$ does not eventually decide on all processes on run w. Therefore $w \notin \mathcal{M}$. Now we show that all elements w' of the geo-class of w are also not in $\mathcal{M}$. Assume otherwise, then $\mathcal{A}$ halts on w'. By Prop. 16, at any t, the simplex corresponding to $w'_{|t}$ intersects T_t on a simplex of smaller dimension whose geometrization contains y. Consider t_0 such that the execution has decided at this round for w'. Consider now T_{t_0+1}, it intersects the decided simplex of $Chr^{t_0}(\mathbb{S}^n)$ corresponding to w', which means that the processes corresponding to the intersection were solo in $w'(t_0+1)$ and in $w(t_0+1)$. When a process does not belong to a set of solo processes of the round, it receives all their values. So by normalization property of algorithm $\mathcal{A}$, this means that in T_{t_0+1}, all processes have decided. A contradiction with the fact that $\mathcal{A}$ does not decide on all processes on run w. ◀

This impossibility result means that there are many strict subsets $\mathcal{M}$ of IIS_n where it is impossible to solve set-agreement, including cases where $IIS_n \setminus \mathcal{M}$ is of infinite size.

5.2 Algorithms for Set-Agreement

In this section, we consider message adversaries $\mathcal{M}$ that are of the form $IIS_n \setminus geo^{-1}(y)$ for a given $y \in |\mathbb{S}^n|$. We note $w \in IIS_n$, such that $geo(w) = y$. We have $w \notin \mathcal{M}$. In other words, $\mathcal{M} = IIS_n \setminus \mathcal{C}$, where $\mathcal{C} = geo^{-1}(geo(w))$ is the equivalence class of w. We also denote $\sigma_y(r)$ the simplex $geo(w_{|r})(\mathbb{S}^n)$.

5.2.1 From Sperner Lemma to Set-Agreement Algorithm

Remark that the protocol complex at time r is exactly $Chr^r(\mathbb{S}^n)$, there is no hole "appearing" in finite time for such $\mathcal{M}$. From Sperner Lemma, any Sperner labelling of a subdivision of $\mathbb{S}^n$ admits at least one simplex that is panchromatic. In order to solve set-agreement, the idea of Algorithm 2 is to try to confine the panchromatic, problematic but unavoidable, simplex of $Chr^t(\mathbb{S}^n)$ to $\sigma_y(r)$. Since the geo-class of w is not in $\mathcal{M}$, any execution will eventually diverge from $\sigma_y(r)$ and end in a non panchromatic simplex. We now define a special case of Sperner labelling of the Standard Chromatic Subdivision that admits exactly one given simplex that is panchromatic.

We consider the generic colored simplex (S, χ) where $S = (x_0, \ldots, x_n)$ and coloring function χ, that could be different from $\mathcal{P}$. We consider labellings of subdivisions C of S.

▶ **Definition 30.** *Let $\tau \in C$ a subdivision of S. $f : V(T) \longrightarrow \Pi$ is a* Sperner $\tau-$panlabelling *if: f is a Sperner labelling of C; for all simplex $\sigma \in C$, $f(\sigma) = \Pi$ if and only if $\sigma = \tau$.*

▶ **Proposition 31.** *Let τ be a face of $Chr(S, \chi)$, there exists a $\tau-$panlabelling λ of $Chr(S, \chi)$.*

Algorithm 2 Algorithm $\mathcal{A}_w$ for process i.

```
1 $x \leftarrow x_i^*$; $r \leftarrow 0$;
2 Loop while $x \in V(geo(w_{|r})(\mathbb{S}^n)$
3     $r \leftarrow r + 1$;
4     $Send((i, x))$;
5     $V \leftarrow$Receive() // set of all received messages;
6     $d \leftarrow sizeof(V) - 1$ // $i$ receives $d + 1$ messages ;
7     $x = \frac{1-\frac{d}{2d+1}}{d+1} x + \sum_{(j,x_j)\in V, j\neq i} \frac{1+\frac{1}{2d+1}}{d+1} x_j$;
8 Output: $\Psi_w(r)(x)$;
```

This technical proposition is proved in the appendix. Denote $\lambda_\tau(S, \chi)$ such a Sperner $\tau-$panlabelling of $Chr(S, \chi)$.

Before stating the algorithm, we show how to construct a sequence of panlabellings for $Chr^r(\mathbb{S}^n)$. Let $r \in \mathbb{N}$, we denote $\Psi_w(r)$ the following labelling defined by recurrence.

Intuitively, it is the following labelling. In $Chr^r(\mathbb{S}^n)$, we have $\sigma_y(r)$ that is panchromatic, all other simplices using at most n colors. In $Chr^{r+1}(\mathbb{S}^n)$, we label vertices that do not belongs to the subdivision of $\sigma_y(r)$ by the labels used at step r. In vertices from $Chr\sigma_y(r)$, we use $\lambda_{\theta(w(r+1))}$ the Sperner $\tau-$panlabelling associated with $\theta(w(r + 1))$ to complete the labelling that uses at most n colors on a given simplex, except at $\sigma_y(r)$. In order to simplify notation, we also note λ_G the labelling $\lambda_{\theta(G)}$. Of course, we apply $\lambda_{w(r+1)}$ using as input (corner) colors, the colors from $\Psi_w(r)$. This way, on the neighbours of $\sigma_y(r)$ the labelling is compatible.

We denote $\gamma_r(x)$ the precursor of level r of $x \in V(Chr^{r+1}(\mathbb{S}^n))$, that is the vertex of $V(Chr^r(\mathbb{S}^n))$ from which x is originating.

▶ **Definition 32.** *We set $\Psi_w(1)(x) = \lambda_{w(1)}(\mathbb{S}^n, \mathcal{P})(x)$ for all $x \in V(Chr^r(\mathbb{S}^n))$, and for $r \in \mathbb{N}^*$*

$$\begin{aligned}\Psi_w(r+1)(p) \quad = \quad & \Psi_w(r)(\gamma_r(x)) \text{ if } x \notin |geo(w_{|r})(\mathbb{S}^n)| \\ & \lambda_{w(r+1)}(\Psi_w(r)(\sigma_y(r))(x) \text{ if } x \in |geo(w_{|r})(\mathbb{S}^n)|\end{aligned}$$

▶ **Proposition 33.** *For all r, $\Psi_w(r)$ is a Sperner $\sigma_y(r)-$panlabelling of $Chr^r(\mathbb{S}^n)$.*

Proof. The proof is done by recurrence. The case $r = 1$ is Prop 31. Assume that $\Psi_w(r)$ is a Sperner $\sigma_y(r)-$panlabelling of $Chr^r(\mathbb{S}^n)$.

Consider now $\Psi_w(r + 1)$ for $Chr^{r+1}(\mathbb{S}^n)$. By construction and recurrence assumption, panchromatic simplices can only lay in $|\sigma_y(r)|$. Since $\lambda_{w(r+1)}$ is a Sperner panlabelling and that the corner colors for σ_y are taken from $\Psi_w(r)$, we have that $\sigma_y(r)$ is the only panchromatic simplex of $Chr^{r+1}(\mathbb{S}^n)$. ◀

We now prove the correctness of $\mathcal{A}_w$ presented in Algorithm 2. Consider an execution $v \in \mathcal{M}$. For Termination: since elements of the geo-class of w are not in $\mathcal{M}$, there exists a round r at which $v_{|r} \neq w'_{|r}$ for all $w' \in geo^{-1}(geo(w)$, *i.e.* the conditional at line 2 is false for all processes and the algorithm is terminating. For Agreement: when terminating at round r, i is not in $\sigma_y(r)$, by loop conditional, so since $\Psi_w(r)$ is only panchromatic on $\sigma_y(r)$, the number of decided values is less than n. Integrity comes from the fact that $\Psi_w(r)$ is a Sperner labelling.

5.2.2 Lower Bounds

It is possible to use the impossibility result to prove the following lower bound. Algorithm 2 is therefore optimal for fair w.

▶ **Theorem 34.** *Let $\mathcal{A}$ be an algorithm that solves set-agreement on $\mathcal{M} = IIS_n \backslash geo^{-1}(geo(w))$ with $w \in IIS_n$. Then, for any execution $v \in \mathcal{M}$, $t \in \mathbb{N}$, such that $v_{|t} = w'_{|t}$ for some $w' \in geo^{-1}(geo(w))$, $\mathcal{A}$ has not terminated at t.*

Proof. Suppose $\mathcal{A}$ has decided on all process at t, with $v_{|t} = w'_{|t}$ for some $w' \in geo^{-1}(geo(w))$. So $\mathcal{A}$ solve set-agreement on w'. A contradiction with Th. 27 since $\wr\mathcal{M} \cup \{w'\}\wr = |\mathbb{S}^n|$. ◀

6 Conclusion and Implications for Topological Methods

In this note, we have presented how to construct a topology directly on the set of executions of IIS_n the Iterated Immediate Snapshot message adversary. Though this is not a simple textbook topology as usual, since there are non-separable points, the properties we presented enables to fully understand it. As a important application on using the underlying geometrization mapping *geo*, we were able to characterize precisely for the first time general subsets of IIS_n where set-agreement is solvable and give a topological interpretation of this result.

We also believe this new approach could be successfully applied to other distributed tasks and distributed models. When considering the input complex embedded in $\mathbb{R}^N$, the geometrization topology could be applied on all simplices, in effect providing a new topological framework for so called protocol complex. This could be done by applying the Chromatic Average algorithm. This was not detailed here as we did not need it to investigate set-agreement. Moreover, note that this construction works also for any model of computation that corresponds to a mesh-shrinking subdivision.

This geometrization topology provides also a nice topological interpretation for the characterization theorem. In particular the topology as defined here is the coarsest topology such that the mapping *geo* is continuous. Therefore the impossibility theorem could also be stated, using the No-Retraction Theorem of standard topology [11, Cor. 2.15]: set-agreement is solvable on message adversary $\mathcal{M}$ only if there exists a continuous function $f : \mathcal{M} \longrightarrow \partial\mathbb{S}^n$, where $\mathcal{M}$ has the geometrization topology. This is interesting since $\partial\mathbb{S}^n$, the boundary of $\mathbb{S}^n$, is exactly the output complex of the set-agreement task.

It should also be noted that, since we do have non-separable sets in our setting, it shows that the standard abstract simplicial complexes approach is actually not always directly usable, since abstract simplicial complexes are known to have separable topology. It means that, for the first time, we have to primarily use the geometric version of simplicial complexes to fully investigate general distributed computability. We call the topology defined here the *geometrization topology* to emphasize this change of paradigm.

References

1 Yehuda Afek and Eli Gafni. *Asynchrony from Synchrony*, pages 225–239. Number 7730 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013.

2 Hagit Attiya, Armando Castañeda, and Thomas Nowak. Topological characterization of task solvability in general models of computation. In Rotem Oshman, editor, *Proceedings of the 37th International Symposium on Distributed Computing (DISC'23)*, volume 281 of *LIPICS*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2023.

3 Elizabeth Borowsky and Eli Gafni. Generalized flp impossibility result for t-resilient asynchronous computations. In *STOC '93: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 91–100, New York, NY, USA, 1993. ACM Press. doi:10.1145/167088.167119.

4 Elizabeth Borowsky and Eli Gafni. A simple algorithmically reasoned characterization of wait-free computation (extended abstract). In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '97, pages 189–198. ACM, 1997. doi:10.1145/259380.259439.

5 Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. Time-varying graphs and dynamic networks. *Int. J. Parallel Emergent Distributed Syst.*, 27(5):387–408, 2012.

6 Bernadette Charron-Bost, Matthias Függer, and Thomas Nowak. Approximate consensus in highly dynamic networks: The role of averaging algorithms. In *ICALP (2)*, volume 9135 of *Lecture Notes in Computer Science*, pages 528–539. Springer, 2015.

7 Matthias Függer, Thomas Nowak, and Manfred Schwarz. Tight bounds for asymptotic and approximate consensus. *J. ACM*, 68(6):46:1–46:35, 2021.

8 Eli Gafni, Petr Kuznetsov, and Ciprian Manolescu. A generalized asynchronous computability theorem. In Magnús M. Halldórsson and Shlomi Dolev, editors, *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*, pages 222–231. ACM, 2014.

9 Emmanuel Godard and Eloi Perdereau. Back to the coordinated attack problem. *Math. Struct. Comput. Sci.*, 30(10):1089–1113, 2020.

10 Darald J. Hartfiel. Behavior in Markov set-chains. In Darald J. Hartfiel, editor, *Markov Set-Chains*, Lecture Notes in Mathematics, pages 91–113. Springer, Berlin, Heidelberg, 1998. doi:10.1007/BFb0094591.

11 Allen Hatcher. *Algebraic Topology.* Cambridge University Press, 2002.

12 Maurice Herlihy, Dmitry N. Kozlov, and Sergio Rajsbaum. *Distributed Computing Through Combinatorial Topology.* Morgan Kaufmann, 2013.

13 Maurice Herlihy, Sergio Rajsbaum, and Michel Raynal. Computability in distributed computing: A tutorial. *SIGACT News*, 43(3):88–110, 2012.

14 Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *J. ACM*, 46(6):858–923, 1999.

15 Dmitry N. Kozlov. *Combinatorial Algebraic Topology*, volume 21 of *Algorithms and computation in mathematics.* Springer, 2008.

16 Dmitry N. Kozlov. Chromatic subdivision of a simplicial complex. *Homology Homotopy Appl.*, 14(2):197–209, 2012. URL: http://projecteuclid.org/euclid.hha/1355321488.

17 Petr Kuznetsov, Thibault Rieutord, and Yuan He. An asynchronous computability theorem for fair adversaries. In Calvin Newport and Idit Keidar, editors, *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, Egham, United Kingdom, July 23-27, 2018*, pages 387–396. ACM, 2018. URL: https://dl.acm.org/citation.cfm?id=3212765.

18 Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

19 James R. Munkres. *Elements Of Algebraic Topology.* Addison Wesley Publishing Company, 1984.

20 Thomas Nowak, Ulrich Schmid, and Kyrill Winkler. Topological characterization of consensus under general message adversaries. In *PODC*, pages 218–227. ACM, 2019.

21 J.E. Pin and D. Perrin. *Infinite Words*, volume 141 of *Pure and Applied Mathematics.* Elsevier, 2004.

22 Sergio Rajsbaum. Iterated shared memory models. In Alejandro López-Ortiz, editor, *LATIN 2010: Theoretical Informatics*, pages 407–416, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

23 Michel Raynal. *Set Agreement*, pages 1956–1959. Springer New York, New York, NY, 2016. doi:10.1007/978-1-4939-2864-4_367.

24 Thibault Rieutord. *Combinatorial characterization of asynchronous distributed computability. (Caractérisation combinatoire de la calculabilité distribuée asynchrone)*. PhD thesis, University of Paris-Saclay, France, 2018. URL: https://tel.archives-ouvertes.fr/tel-02938080.

25 M. Saks and F. Zaharoglou. "wait-free k-set agreement is impossible: The topology of public knowledge. *SIAM J. on Computing*, 29:1449–1483, 2000.

26 Emanuel Sperner. Neuer beweis für die invarianz der dimensionszahl und des gebietes. *Math. Sem. Univ. Hamburg*, 6:265–272, 1928.

A Geometrization Topology

A.1 Convexity and Metric Results

We present some metric results relating vertices of the iterated chromatic subdivision. In particular we prove that the sequences $geo(w_{|r})(S)$ converge to a point. This is related to the known fact that the standard chromatic subdivision is *mesh-shrinking* [12].

The following lemma comes from the convexity of the μ_G transforms.

▶ **Lemma 35.** *Let w a run, let $r, r' \in \mathbb{N}, r < r'$ then $|geo(w_{|r'})(\mathbb{S}^n)| \subset |geo(w_{|r})(\mathbb{S}^n)|$.*

Proof. Consider only one step. We have that $\frac{1-\frac{d}{2d+1}}{d+1} + d \times \frac{1+\frac{1}{2d+1}}{d+1} = \frac{1-\frac{d}{2d+1}+d+\frac{d}{2d+1}}{d+1} = 1$. So one step of the Chromatic Average gives, on each process, a linear combination with non-negative coefficients that sums to 1, it is therefore a barycentric combination on the points of the simplex at the beginning of the round. It is therefore a convex mapping of this simplex. Since composing convex mapping is also convex, and that $\mathbb{S}^n$ is a convex set, we get the result by recurrence. ◀

▶ **Lemma 36.** *There exists reals $0 < K' < K < 1$, such that for all G of IS_n, all $p, q \in V(S)$, $p', q' \in V(\mu_G(S))$, such that $\mathcal{P}(p) = \mathcal{P}(p')$ and $\mathcal{P}(q) = \mathcal{P}(q')$, we have*

$$K'||p - q|| \leq ||p' - q'|| \leq K||p - q||.$$

Proof. This is a consequence of μ_G transforms being convex when $G \in IS_n$. It corresponds to a stochastic matrix (non-negative coefficients and all lines coefficient sums to one) that is scrambling (there is a line without null coefficients) hence contractive. See *e.g.* [10, Chap. 1] for definitions and a proof for any given G of IS_n.

Then K (resp. K') is the largest (resp. smallest) such bounds over all $G \in IS_n$. ◀

While iterating the chromatic subdivision, we remark that the diameter of the corresponding simplices is contracting. From Lemma 36, we have

▶ **Lemma 37.** *Let S a simplex of $\mathbb{R}^{\mathbb{N}}$, then $diam(\mu_{G_r} \circ \mu_{G_{r-1}} \circ \cdots \circ \mu_{G_1}(S)) \leq K^r diam(S)$, where K is the constant from the previous lemma.*

Since the simplices are contracted by the μ_G functions, the sequence of isobarycenters of $(geo(w_{|r}(S))_{r\in\mathbb{N}^*}$ has the Cauchy property and this sequence is therefore convergent to some point $x \in \mathbb{R}^N$. Since the diameter of the simplices converges to 0, it makes senses to say that the limit of the simplices is the point x. Note that it would also be possible to formally define a metric on the convex subsets of $\mathbb{R}^N$ and consider the convergence of the simplices in this space.

A.2 Geometrization Topology vs Geometric Realization Topology

In this section, we provide an example of a simplicial complex whose topology as a geometric realization is different from the topology it has in the ambient R^N space, here with $N = 1$ but that can be generalized to any N. This is actually quite well known, see e.g. [15].

We consider $C = \{0\} \cup \{[\frac{1}{r+1}, \frac{1}{r}] \mid r \in \mathbb{N}^*\}$.

We denote $|C|$ the topological space of C defined as a geometric realization. The closed sets of $|C|$ are the sets F such that $F \cap S$ is closed (in $\mathbb{R}$) for all $S \in C$, see [19]. Therefore $|C|$ has two connected components. We have $F =]0, 1]$ is closed in $|C|$ since $F \cap [\frac{1}{r+1}, \frac{1}{r}] = [\frac{1}{r+1}, \frac{1}{r}]$, hence is closed for all r. Moreover, $F \cap \{0\} = \emptyset$ which is also closed in $\mathbb{R}$. We also have that $\{0\}$ is closed in $|C|$, so C can be covered by two disjoint closed sets, it is not connected.

On the other end, at the set level, $\wr C \wr$ is exactly $[0, 1]$. So within the standard ambient topology of $\mathbb{R}$, $\wr C \wr$ is connected.

Since they do not have the same number of connected components, the two spaces C as a geometric realization and with the subset topology cannot be homeomorphic.

A full discussion of these differences could be very interesting. Given that the topologies are the same when the complex is finite, the question at stake seems to be the passage to the limit.

B Sperner Panlabellings of the Standard Chromatic Subdivision

In this section, n is fixed. We show how to construct a Sperner panlabelling of the standard chromatic subdivision. We consider the generic colored simplex (S, χ) where $S = (x_0, \dots, x_n)$ and coloring function χ, that could be different from $\mathcal{P}$. We consider labellings of the colored complex $Chr(S, \chi)$.

We show the following combinatorial result about Sperner labellings.

▶ **Theorem 38.** *Let τ be a maximal simplex of $Chr(S, \chi)$, then there exists a $\tau-$panlabelling λ of $Chr(S, \chi)$.*

We start by some definitions related to proving the above theorem. It is possible to associate to any simplex σ of $Chr(S)$ a pre-order $\succ$ on Π that corresponds to the associate graph $\Theta(\sigma)$: $i \succ j$ when $(i, j) \in A(\Theta(\sigma))$. We call equivalence classes for $\Theta(\sigma)$, the classes of the equivalence relation defined by $i \succ j \wedge j \succ i$. It corresponds actually to the strongly connected components of the directed graph $\Theta(\sigma)$.

We define the *process view* of a point. This is the color of points in the view V of vertex (i, V) of the standard chromatic subdivision.

▶ **Definition 39** (Process View). *The* process view *of point $x = (\chi(x), V) \in V(\mathrm{C}hr(S, \chi))$ is defined by : $V_x = \{\chi(y) | y \in V\}$.*

For $\tau \in \mathrm{C}hr(S, \chi)$, we also define the *process view relative to τ* of a process p, denoted V_p^τ. It is the process view of the point of τ whose color is p. It is linked to pre-order $\succ$: we have $V_p^\tau = \{q \mid q \succ p\}$.

Let τ be a fixed maximal simplex of $Chr(S)$. We show how to construct a $\tau-$panlabelling. We choose a permutation φ on Π such that it defines circular permutations on the equivalent classes of $\Theta(\tau)$. Let $p \in \Pi$, given $W \subset V_p^\tau$ such that $p \in W$, we denote by $min^*(p, W) = \min\{i \in \mathbb{N}^* \mid \varphi^i(p) \in W\}$. Note that since φ is a permutation, there exists $j > 0$ such that $\varphi^j(p) = p$, and since $p \in W$, the minimum is taken over a non-empty set. Finally we set $\varphi^*(p, W) = \varphi^{min^*(p,W)}(p)$. This is the first point of W that is in the orbit of p in φ.

▶ **Definition 40.** *We define $\lambda_\tau : V(Chr(S)) \to V(S)$, for $x \in V(Chr(S))$, we set*

$$\lambda_\tau(x) = \begin{cases} q & \text{if } \exists q \in V_x \text{ and } q \notin V^\tau_{\chi(x)} \\ \varphi^*(\chi(x), V_x \cap V^\tau_{\chi(x)}) & \text{otherwise.} \end{cases}$$

Intuitively, for a given vertex of $Chr(S)$ with view V, if the process sees an other process q than in τ, then it is labelled by this q, otherwise it will choose the first process in the circular orbit of φ that is in its view.

▶ **Proposition 41.** *The labelling λ_τ is a $\tau-$panlabelling.*

Proof. First we show that it is indeed a Sperner labelling. In both cases of the definition, $\lambda_\tau(x)$ belongs to V_x. For $x \in V(Chr(S))$, for $\sigma \subset S$, $x \in |\sigma|$, with σ minimum for this property, means that the presentation of x is $\Phi(x) = (i, \sigma)$ for some x_i such that $x_i \in V(\sigma)$ and $\chi(x_i) = i$.

Now we show that the only panchromatic simplex is τ. By construction, with $x \in V(\tau)$, $\varphi^*(\chi(x), V_x) = \varphi(\chi(x))$ since in this case $V_x = V^\tau_{\chi(x)}$. So τ is panchromatic through λ_τ.

Now we consider $\sigma \neq \tau$. We have two possible cases:

1. $\exists x \in V(\sigma), q \in V_x, q \notin V^\tau_{\chi(x)}$,
2. $\forall x \in V(\sigma), V_x \subseteq V^\tau_{\chi(x)}$.

We start with the first case, we denote by C the highest, for $\succ$ in σ, class such there is x in C satisfying the clause (1). We show that $\#\lambda_\tau(C) \cap C < \#C$, where $\#$ denotes the cardinal of a set. By definition of C, $\lambda_\tau^{-1}(C) \subseteq C$. Since $\lambda_\tau(x) \notin C$, this means that $\#\lambda_\tau(C) \cap C \neq \#C$. By assumption all classes C' that are higher than C choose colors in C', so σ is not panchromatic under λ_τ.

Now, we assume we do not have case (1), this means that $\forall x \in V(\sigma), \lambda_\tau(x) = \varphi^*(\chi(x), V_x)$. Since $\sigma \neq \tau$, there exists $x \in V(\sigma), V_x \subsetneq V^\tau_{\chi(x)}$. We choose the lowest such x for $\succ$ in τ. We consider C_x the class of x in σ. We show that $\#\lambda_\tau(C_x) < \#C_x$.

We denote C^τ_x the class of color $\chi(x)$ in τ. First we show that $C_x \subseteq C^\tau_x$. Indeed, assume there is $y \in C_x$ such that $y \notin C^\tau_x$. Since the view of elements of the same class are the same, this means that $\chi(x) \in V_y$ and y would satisfy property 1. A contradiction to the case we are considering. And this is true for all $y \in C_x$.

Now we show $C_x \subsetneq C^\tau_x$. We have $V_x \subsetneq V^\tau_{\chi(x)}$. Let $y \in V^\tau_{\chi(x)} \setminus V_x$. If $y \notin C^\tau_x$, by the same previous argument, we get a contradiction. Hence $y \in C^\tau_x$ and therefore $C_x \subsetneq C^\tau_x$.

We denote $p = \varphi^*(\chi(x), C^\tau_x \setminus C_x)$. We note $p' = \varphi^{-1}(p)$. We have by definition of $\varphi^*(., C^\tau_x \setminus C_x)$, that $p' \in C_x$, therefore $C_{\chi^{-1}_{|\sigma}(p')} = C_x$. Now we set $p'' = \varphi^*(p', C_x)$. The color p'' has at least two predecessors in the labelling: p' by construction (since x was chosen the lowest for $\succ$ then $V_{\chi^{-1}_{|\sigma}(p')} = V^\tau_{p'}$) and $p''' = \varphi^{-1}(p'')$ which is not p' since $\varphi(p') = p \notin C_x$.

So $\#\lambda_\tau(\chi^{-1}_{|\sigma}(V^\tau_x)) < \#V^\tau_x$, and $\lambda_\tau(\sigma) \neq \Pi$. ◀

Send/Receive Patterns Versus Read/Write Patterns in Crash-Prone Asynchronous Distributed Systems

Mathilde Déprés
École Normale Supérieure de Paris-Saclay, France

Achour Mostéfaoui
LS2N, Nantes Université, France

Matthieu Perrin
LS2N, Nantes Université, France

Michel Raynal
Univ Rennes IRISA, Inria, CNRS, France

Abstract

This paper is on the power and computability limits of messages patterns in crash-prone asynchronous message-passing systems. It proposes and investigates three basic messages patterns (encountered in all these systems) each involving two processes, and compares them to their Read/Write counterparts. It is first shown that one of these patterns has no Read/Write counterpart. The paper proposes then a new one-to-all broadcast abstraction, denoted *Mutual Broadcast* (in short MBroadcast), whose implementation relies on two of the previous messages patterns. This abstraction provides each pair of processes with the following property (called *mutual ordering*): for any pair of processes p and p', if p broadcasts a message m and p' broadcasts a message m', it is not possible for p to deliver first (its message) m and then m' while p' delivers first (its message) m' and then m. It is shown that MBroadcast and atomic Read/Write registers have the same computability power (independently of the number of crashes). Finally, in addition to its theoretical contribution, the practical interest of MBroadcast is illustrated by its (very simple) use to solve basic upper level coordination problems such as mutual exclusion and consensus. Last but not least, looking for simplicity was also a target of this article.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Computer systems organization → Fault-tolerant network topologies; Networks → Programming interfaces

Keywords and phrases Asynchrony, Atomicity, Broadcast abstraction, Characterization, Consensus, Crash failure, Distributed Computability, Distributed software engineering, Computability, Lattice agreement, Message-passing, Message pattern, Mutual exclusion, Quorum, Read/write pattern, Read/Write register, Test&Set, Simplicity, Two-process communication

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.16

Related Version *Full Version*: https://hal.science/hal-04087447
Previous Version: https://doi.org/10.1145/3583668.3594569

Funding This work was partially supported by the French “Étoile Montante en Pays De La Loire” regional project BROCCOLI devoted to the computability aspects of broadcast abstractions and the French ANR project ByBloS (ANR-20-CE25-0002-01) and PriCLeSS (ANR-10-LABX-07-81).

1 Introduction

1.1 On the nature of distributed computing

The aim of parallel computing is to allow programmers to exploit data independence in order to obtain efficient programs. In distributed computing, the situation is different: there is a set of predefined computing entities (imposed to programmers) that need to cooperate to a common goal. Moreover, the behavior of the underlying infrastructure (environment) on which the distributed application is executed is not on the control of the programmers, who have to consider it as a "hidden input". Asynchrony and failures are the most frequent phenomenons produced by the environment that create a "context uncertainty" distributed computing has to cope with. In short, distributed computing is characterized by the fact that, in any distributed run, the run itself is one of its entries [47].

1.2 From send/receive to cooperation abstractions

The operations send() and receive() constitute the machine language of underlying networks. So, in order to solve a distributed computing problem it is usual to first define an appropriate communication abstraction that makes easier the design of higher level algorithms. FIFO and Causal message-ordering [10, 26, 41, 50] are examples of such communication abstractions that make easier the construction of distributed objects such as, for example, the construction of a causal memory on top of an asynchronous message-passing system [4]. A well-know high level and very powerful communication abstraction is total order broadcast, which ensures that the message delivery order is the same at all processes.

Another example is the *Set-Constrained Delivery* (SCD) communication abstraction introduced in [32]. This broadcast abstraction allows processes to deliver a sequence of sets of messages of arbitrary size (instead of a sequence of messages) satisfying a non trivial intersection property. In terms of computability in the presence of asynchrony and process crashes, the power of SCD-broadcast is the same as the one of atomic read/write registers. SCD-broadcast is particularly well-suited to efficiently implement a snapshot object (as defined in [1, 5]) with an $O(n^2)$ message complexity in asynchronous crash-prone message-passing systems. Broadcast abstractions suited to specific problems have also been designed (e.g., [31] for k-set agreement).

d-Solo models consider asynchronous distributed systems where any number of processes may crash. In these models, up to d ($1 \leq d \leq n$) processes may have to run solo, computing their local output without receiving any information from other processes. Differently from the message-passing communication model where up to $d \geq 1$ processes are allowed to run solo, it is important noticing that the basic atomic read/write registers communication model allows at most one process to run solo. Considering the family of d-solo models, [28] presents a characterization of the colorless tasks that can be solved in each d-solo model.

1.3 On the read/write side

Read/write (RW) registers (i.e., the cells of a Turing machine) are at the center of distributed algorithms when the processes communicate through a shared memory. So a fundamental problem is the construction of atomic RW registers on top of crash-prone asynchronous message-passing system. This problem has been solved by Attiya, Bar-Noy and Dolev who presented in [7] a send/receive-based algorithm (ABD) for such a construction, and proved that, from an operational point of view, such constructions are possible if and only if at most

$t < n/2$ processes may crash. The ABD construction is based on the explicit use of sequence numbers, quorums and a send/receive pattern (used once to write and twice to read). The quorums are used to realize synchronization barriers[1].

Based on the same principles as ABD, algorithms building RW registers have been designed [52]. Some strive to reduce the size of control information carried by messages (e.g. [6, 42]) while others focus on fast read and/or fast write operations in good circumstances (e.g. [21, 25, 43]). All these algorithms allow existing RW-based algorithms to be used on top of crash-prone asynchronous message-passing systems.

The use of RW registers on top of a message-passing system to allow processes to use "for free" existing shared memory-based distributed algorithms has a cost, which can be higher than the one obtained with an algorithm directly designed on top of the basic send/receive operations (as shown, for example, in [16] for the snapshot object defined in [1, 5]). This means that, for some problems (as we will see for consensus), algorithms based on appropriate communication abstractions can be more efficient than the stacking of RW-based algorithms on top of simulated RW registers.

1.4 Content of the article

In the spirit of [24] (which states that computing is "science of abstractions") the present article introduces a new broadcast abstraction (denoted MBroadcast) that ensures that for any pair of processes p and p', if p broadcasts a message m and p' broadcasts a message m', it is not possible for p to deliver first its message m and then m' while p' delivers first its message m' and then m. It is important to notice that this property is on each pair of processes taken separately from the other processes. It is also shown how, at the upper layer, this broadcast abstraction allows a very simple design of message-passing algorithms solving distributed coordination and agreement problems. The main properties of MBroadcast are the following ones.

- It has the same computability power as RW registers.
- It constitutes the first characterization of RW registers in terms of (binary) message patterns.
- It allows to build higher level coordination abstractions without requiring as prerequisite the construction of an intermediary abstraction level made up of RW registers.
- When looking at a message exchange between two processes p and p', it shows that the fact that p' does not ignore the other process p (because it received a message from p before receiving its own message) is a powerful control information (more technically, this refers to the patterns MP2 and MP3 defined in Section 3).

An important point of the article is the fact that atomic RW registers can be implemented from two simple basic message patterns on each pair of processes only. Hence the article is on basic patterns that allow us to better understand the close relationship between RW registers and asynchronous message-passing in the presence of process crashes. More generally, it allows for a better understanding of the strengths and weaknesses of the world of asynchronous crash-prone message-passing systems.

As an important side note, this article also discusses a strengthening of MBroadcast, denoted PBroadcast, that ensures that any pair of processes deliver their own messages in the same order (in the terms of the patterns defined in Section 3, it means that the only

[1] Let us notice that, in the traditional use of the send/receive patterns, a process that broadcasts a message (i.e., sends a message m to all the processes including itself) is allowed to locally deliver m without waiting for a specific delivery condition to be satisfied.

possible pattern is MP2). On a computability viewpoint, it shows that if the only binary message pattern that can occur is MP2, then Test&Set, the consensus number of which is 2, can be built.

1.5 Roadmap

The article is composed of 8 sections, structured in two parts. The first part consists of sections 2-5, while the second part consists of sections 6-7. Section 2 presents the underlying computing model. Section 3 presents three basic binary message patterns (denoted MP1, MP2 and MP3) encountered in asynchronous message-passing computations, and establishes a "correspondence" linking these message patterns with read/write patterns on atomic registers. Section 4 introduces the high level MBroadcast and PBroadcast communication abstractions. Section 5 shows that MBroadcast and atomic read/write (RW) registers have the same computability power.

The second part illustrates uses of MBroadcast, that show the conceptual gain offered by this communication abstraction (i.e., simplicity). More precisely, Section 6 presents an MBroadcast-based rewriting of Lamport's bakery algorithm suited to message-passing (i.e., suited to state machine replication [37]). Section 7 presents a simple version of the well-known Paxos consensus algorithm [36].[2] It is important to state that none of the algorithms built on top of MBroadcast uses quorums (as we will see, this means if each binary message exchange pattern satisfies the pattern MP2 or the pattern MP3 these algorithms work correctly even if a majority of processes crashes).

Finally, Section 8 concludes the article (where design simplicity is considered as a first class citizen property). The proofs of the algorithms are presented in an appendix.[3]

2 Distributed Computing Model

The computing model is the classical asynchronous crash-prone message-passing model.

2.1 Process model

The computing model is composed of a set of n sequential processes denoted $p_1, ..., p_n$. Sometimes, when considering two processes, they are denoted p and p'.

Each process is asynchronous which means that it proceeds at its own speed, which can be arbitrary and remains always unknown to the other processes. A process may halt prematurely (crash failure), but executes correctly its local algorithm until it possibly crashes. The model parameter t denotes the maximal number of processes that may crash in a run. A process that crashes in a run is said to be *faulty*. Otherwise, it is *correct* or *non-faulty*.

2.2 Communication model

Each pair of processes communicate by sending and receiving messages through two uni-directional channels, one in each direction. Hence, the communication network is a complete network: any process p_i can directly send a message to any process p_j (including itself). A

[2] It is worth noticing that mutex and consensus are the two most famous distributed computing problems [49]. Additionally, Appendix B presents a very simple MBroadcast-based algorithm that builds a lattice agreement object.

[3] The definition of MBroadcast was first presented in a short version at PODC 2023 [18], and an extended version is available on the web [19].

process p_i invokes the operation "send TYPE(m) to p_j" to send the message m (whose type is TYPE) to p_j. The operation "receive TYPE(m) from p_j" allows p_i to receive from p_j a message m whose type is TYPE.

Each channel is reliable (no loss, no corruption, no creation of messages), not necessarily first-in/first-out, and asynchronous (while the transit time of each message is finite, there is no upper bound on message transit times). Let us notice that, due to process and message asynchrony, no process can know if another process crashed or is only very slow.

It is assumed that, in addition to basic send/receive operations, the network is enriched with FIFO and causal message deliveries, i.e. it provides the processes with the operations fifo_broadcast(), causal_broadcast(), causal_send(), and causal_delivery() [10, 26, 34, 41, 50, 47]. Two messages m and m' are causally related if (i) they have been sent by the same process and m was sent before m', or (ii) a process received m before sending m$'$, or (iii) there is chain of messages m, m_1, ..., m_x, m' such that each pair of consecutive messages is causally related by (i) or (ii). It is important to note that the addition of this assumption does not change the computability power of the communication model, since causal message delivery can always be implemented on top of send/receive channels. It is also important to note that, while the implementation of these operations requires messages to carry additional control information, it does not require the use of additional implementation messages.

2.3 Notation

The acronym $\mathcal{CAMP}_{n,t}[\emptyset]$ is used to denote the previous Crash-prone Asynchronous Message-Passing model without additional computability power. $\mathcal{CAMP}_{n,t}[H]$ denotes $\mathcal{CAMP}_{n,t}[\emptyset]$ enriched with the additional computational power denoted by H.

The acronym $\mathcal{CARW}_{n,t}[\emptyset]$ is used to denote the n-process asynchronous system where up to t processes may crash and communication is through read/write registers. $\mathcal{CARW}_{n,t}[H]$ denotes $\mathcal{CARW}_{n,t}[\emptyset]$ enriched with H.

3 Three Basic Binary Message Patterns and their RW Counterparts

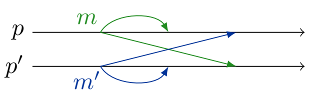
3.1 Three basic binary message patterns

Let us consider two processes p and p' that concurrently exchange messages, namely, p sends a message m to itself and p', while p' sends the message $m' \neq m$ to itself and p. Depending on the order in which messages are delivered at each process, there are exactly three cases to consider (swapping p and p' does not give rise to new message patterns).

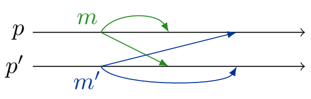
Message pattern MP1. This case is represented at Figure 1a. It is a symmetric pattern in which p delivers first the message m it broadcast and then the message m' broadcast by p', while p' delivers first its message m' and then the message m broadcast by p. This pattern captures the case where, when a process delivers its own message, it has no information on the fact the other processes broadcast or not a message.

Message pattern MP2. This case is represented at Figure 1b. It describes an asymmetric pattern from a message delivery point of view in which both p and p' deliver first m broadcast by p and then m' broadcast by p'. In this pattern both p and p' deliver the messages in the same order (an analogous pattern occurs when we swap p and p').

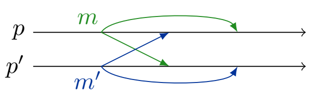
Message pattern MP3. This case is represented at Figure 1c. Similarly to MP1 this is a symmetric pattern in the sense that p delivers first the message m' from p' and then its message m, while p' delivers first the message m from p and then its message m'. The fundamental difference between MP1 and MP3 lies in the fact that when p (resp. p') delivers its own message, it has already delivered the message sent by the other process p' (resp. p).



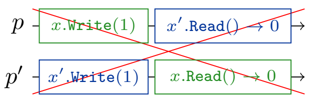
(a) Pattern MP1:
forbidden by MBroadcast
and by PBroadcast.



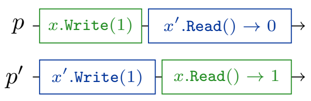
(b) Pattern MP2:
allowed by MBroadcast
and by PBroadcast.



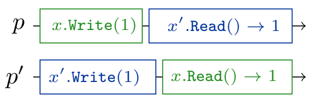
(c) Pattern MP3:
allowed by MBroadcast,
forbidden by PBroadcast.



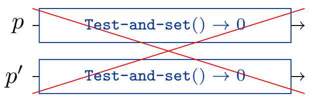
(d) Pattern RW1:
forbidden by atomic memory.



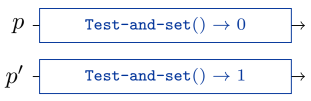
(e) Pattern RW2:
allowed by atomic memory.



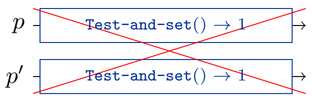
(f) Pattern RW3:
allowed by atomic memory.



(g) Pattern TS1:
forbidden by atomic Test&Set.



(h) Pattern TS2:
allowed by atomic Test&Set.



(i) Pattern TS3:
forbidden by atomic Test&Set.

Figure 1 The three binary message patterns, versus the three binary atomic memory patterns and the three binary atomic test-and-set patterns.

3.2 From message patterns to RW patterns

To deeply understand the meaning and the scope of the three previous message-based communication patterns, let us consider their "counterpart" in a context where p and p' cooperate through atomic one-bit RW registers initialized to 0. The RW register x, written by p and read by p', corresponds to m. The RW register x', written by p' and read by p corresponds to m'. Both registers are initialized to 0. In each case, the read/write pattern depicted at the bottom of Figure 1 simulates the message exchange pattern above it. More precisely we have the following.

RW pattern RW1. Figure 1d corresponds to the message pattern MP1: p writes 1 in x and reads the initial value of x', namely, the value 0. Concurrently, p' writes 1 in x' and reads the initial value of x, namely, the value 0.

RW pattern RW2. Figure 1e corresponds to the message pattern MP2: p writes 1 in x and then reads the initial value 0 from x', while p' writes 1 in x' and then reads the value of x, namely, the value 1.

RW pattern RW3. Figure 1f corresponds to the message pattern MP3: each process writes first "its" variable (x for p, x' for p'), and then reads the other variable and obtains 1.

3.3 Comparing message patterns and RW patterns

It is easy to see that the RW patterns RW2 and RW3 produce the same cooperation as the message patterns MP2 and MP3, respectively. Differently, while the message pattern MP1 can occur in an asynchronous message-passing system, the RW pattern RW1 cannot occur in a RW memory. This is due to the fact that, in RW1, the write of x by p and the write of x' by p' are linearized [29, 35] and, as a process writes a RW register before reading the other register, it is impossible that both read operations return 0 (it is easy to show that this remains true if the registers are only safe, regular, or part of a sequentially consistent memory). This is a fundamental difference between cooperation/communication through message passing and cooperation/communication through RW registers.

At the core of the approach. The fact that there is no RW pattern corresponding to pattern MP1 is implicitly used to solve many cooperation/synchronization problems in RW systems. The most famous is the "write first and then read" pattern used in all mutex algorithms [51]. When a process wants to enter the critical section, it first raises a flag to inform the other processes it starts competing, and only then it reads the flags (which are up or down) of the other processes. The total order imposed by the atomicity on the flag risings prevents RW1 from occurring.

Actually preventing the message pattern MP1 from occurring without bounding the number of process crashes is "equivalent" to the assumption $t < n/2$ without constraints on message exchange patterns, in the sense that both prevent partitioning and consequently allow atomic RW registers to be built despite crashes and asynchrony.

3.4 On the test-and-set side

For comparison, Figures 1g, 1h and 1i consider three communication patterns where the two processes cooperate through the Test&Set special instruction on an atomic register. In a similar way as with message patterns and RW patterns, we can define three TS patterns, depending on which processes obtain 0, the same outcome as in a solo execution; and which processes obtain the same outcome 1 as if their operation was linearized in second position.

TS pattern TS1. Figure 1g corresponds to the RW pattern RW1, in which both processes obtain 0, and hence to the message pattern MP1.

TS pattern TS2. Figure 1h corresponds to the asymmetric RW pattern RW2, in which one process obtains 0 and the other process obtains 1. Hence, it also corresponds to the message pattern MP2.

TS pattern TS3. Figure 1i corresponds to the symmetric RW pattern RW3, in which both processes obtain 1, and hence to the message pattern MP3.

This time, only the asymmetric communication pattern TS2 is admitted. The fact that the pattern TS3 is impossible is a major difference between read/write registers and test-and-set registers, that can be used to solve consensus between two processes.

4 Mutual Broadcast

4.1 Mutual broadcast: Definition

Mutual-broadcast (MBroadcast) is a broadcast abstraction that allows a process to broadcast a message that will be delivered at least by all the correct processes. This abstraction provides the processes with two operations denoted mbroadcast() and mdeliver(). When a process invokes mbroadcast(m) we say "it mbroadcasts the message m". The invocation of mdeliver() returns a message m and we say that a process "mdelivers m" or that "m is mdelivered". To simplify the presentation (and without loss of generality), it it assumed that all the messages that are mbroadcast are different. The following properties define MBroadcast.

Validity. If a process p_i mdelivers a message m from a process p_j, then p_j previously invoked mbroadcast(m).

No-duplication. A process mdelivers a message m at most once.

Mutual ordering. For any pair of processes p and p', if p mbroadcasts a message m and p' mbroadcasts a message m', it is not possible that p mdelivers m before m' and p' mdelivers m' before m.

Local termination. If a correct process invokes mbroadcast(m), it returns from its invocation.

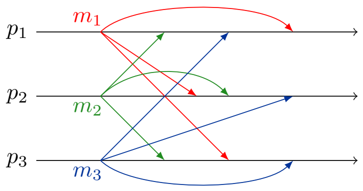
Global CS-termination. If a correct process invokes mbroadcast(m), all correct processes mdeliver m. ("CS" is used to stress the fact that the sender is required to be correct.)

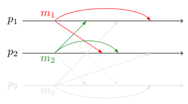
Let us notice that, at the user level, the Mutual ordering property prevents the pattern MP1 from occurring, boils down to the pattern MP2 when a process delivers its message first, and does not prevent pattern MP3 from occurring (which occurs when each process delivers first the message from the other process before its own message).

As it is the case with other broadcast abstractions, MBroadcast can be enriched with other properties, defined in [18], that do not change its computing power but add more usage comfort in some algorithms. We denote by fifo-MBroadcast, causal-MBroadcast and reliable-MBroadcast the MBroadcast abstraction that also respect the properties of fifo-broadcast, causal-broadcast and reliable-broadcast, respectively.

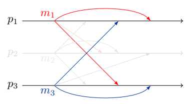
4.2 What does MBroadcast do

When looking at MBroadcast from a *binary* communication point of view between two processes p and p', it appears that MBroadcast ensures that, for any message exchanged by p and p', the message patterns produced is MP2 or MP3. Moreover, it is possible that some of their message exchange patterns are MP2 while others are MP3, and this remains unknown to the processes.

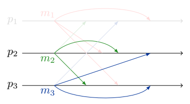




(a) Pattern MP3 between m_1 and m_2.



(b) Pattern MP3 between m_1 and m_3.



(c) Pattern MP2 between m_2 and m_3.

Figure 2 Example of message deliveries with three processes.

Figure 2 presents an example with three processes, that shows MBroadcast message deliveries. As we can see p_1 mdelivers the sequence of messages m_2, m_3, m_1, p_2 mdelivers the sequence of messages m_1, m_2, m_3, and p_3 mdelivers the sequence of messages m_2, m_1, m_3. So, the processes mdeliver the three messages in different orders. Nevertheless, when we consider the projection of these messages exchange on each pair of processes, we observe that the processes p_1 and p_2 mdeliver m_1 and m_2 according to the pattern MP3. As depicted in Figure 2.a each process mdelivers the message from the other process before its own message. The same message pattern MP3 occurs for the messages exchanged by p_1 and p_3 (Figure 2.b). Differently, as shown in Figure 2.c, the messages exchanged by p_2 and p_3 obey the pattern MP2: both processes mdeliver first m_2 and then m_3. The fundamental point is that the pattern MP1 never occurs: MBroadcast prevents it from occurring, which implicitly prevents system partitioning and hides the system parameter t to the algorithms build on top of MBroadcast.

4.3 A real-time property

When combined with the fact that a message cannot be received before it has been sent, MBroadcast ensures that if a process p mdelivers a message m it has mbroadcast before a process p' mbroadcast a message m', p' cannot mdeliver m' before m. This pattern is described in Figure 3. In other words, sending a “blank” synchronization message and waiting for its delivery is sufficient to “harvest” all the messages that were already mdelivered by their senders. To benefit from this property, we consider the synchronized broadcast operation based on a synchronization barrier as defined in Algorithm 1.

Algorithm 1 Synchronized MBroadcast.

```
operation synchro_mbroadcast($m$) is
                                        % code for $p_i$
(1)   mbroadcast($m$);
(2)   wait ($m$ has been mdelivered from $p_i$)
end operation.
```

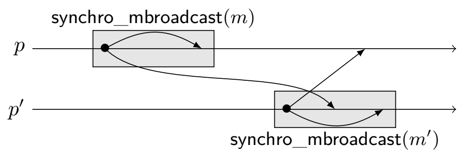


Figure 3 A real-time property.

The operation synchro_mbroadcast() inherits the properties defining MBroadcast, as well as the following property.

▶ **Property 1.** *If a process p returns from the invocation of* synchro_mbroadcast(m) *before p' invokes* synchro_mbroadcast(m'), *p' cannot return from* synchro_mbroadcast(m') *before it has mdelivered m.*

4.4 Pair Broadcast: MP2 Alone Characterizes Test&Set()

Pair broadcast: Definition. Pair broadcast (in short PBroadcast) is a broadcast abstraction that allows a process to broadcast a message that will be delivered at least by all the correct processes. This abstraction provides the processes with two operations denoted pbroadcast() and pdeliver(). When a process invokes pbroadcast(m) we say "it pbroadcasts the message m". The invocation of pdeliver() returns a message m and we say that a process "pdelivers m" or that "m is pdelivered". To simplify the presentation (and without loss of generality), it it assumed that all the messages that are pbroadcast are different. PBroadcast is defined as the same Validity, No-duplication, Local termination and Global CS-termination properties as MBroadcast, and the Pair ordering property (that is a strengthening of the Mutual ordering property):

Pair ordering. For any pair of processes p and p', if p pbroadcasts a message m and p' pbroadcasts a message m', and if m and m' are both pdelivered by p and p', then p and p' pdeliver m and m' in the same order.

Let us notice that, at the user level, the Pair ordering property only allows the pattern MP2 to occur. So, it prevents both the patterns MP1 and MP3 from occurring. It follows that PBroadcast is strictly stronger than Mutual ordering.

On the computability side of PBroadcast. When only two processes participate in an execution, the pair ordering property implies that all messages are pdelivered by the two processes in the same order. In other words, PBroadcast boils down to total-order broadcast in a system composed of only two processes. In particular, it is possible to solve consensus between any pair of processes when PBroadcast is available, and consequently the Test&Set() operation whose consensus number is 2 [27] (and more generally the objects of the class Common2 defined in [3]) can be built on top of PBroadcast. Appendix C details such a construction. Conversely, Appendix C also presents an algorithm that builds PBroadcast in the model $\mathcal{CAMP}_{n,t}[\text{consensus}_2]$ ($\mathcal{CAMP}_{n,t}[\emptyset]$ enriched with consensus objects available between any pair of processes). Therefore, it is easy to see that PBroadcast has consensus number 2, and is part of the equivalence class Common2.

Unsurprisingly, PBroadcast, that only allows the message pattern MP2 to occur, is computationnaly equivalent to the Test&Set() operation, that only allows the memory pattern TS2 to occur. This is similar to the fact that preventing the message pattern MP1 from occuring makes MBroadcast equivalent to shared memory that prevents the memory pattern RW1.

Algorithm 2 MBroadcast on top of $\mathcal{CARW}_{n,t}[\emptyset]$.

```
operation mbroadcast($m$) is                                          % code for $p_i$
(1)   $SENT[i] \leftarrow SENT[i] \oplus m$;
(2)   catch_up();
(3)   mdelivery of $m$ from $p_i$.
end operation.

periodically do catch_up().

internal uninterruptible routine catch_up() is
(4)   for $j$ from $1$ to $i-1$ and then from $i+1$ to $n$ do
(5)      $msgj \leftarrow SENT[j]$;
(6)      for $k$ from $delivered_i[j]$ to $|msgj|-1$ do mdelivery of $msgj[k+1]$ from $p_j$ end for;
(7)      $delivered_i[j] \leftarrow |msgj|$
(8)   end for.
```

Remark on trivial extension attempts. It seems intuitive that, if imposing an order on the messages sent by pairs of processes gives a broadcast with consensus number two, imposing an order on the messages sent by triplets of processes should give a broadcast with consensus number three. Unfortunately, albeit at first glance it seems counter-intuitive, this is not the case. Indeed, let us consider $n \geq 4$ processes p_1, ..., p_n broadcasting respectively messages m_1, ..., m_n using an abstraction providing the following guarantee: each triplet of processes (p_i, p_j, p_k) receives the same first message among m_i, m_j and m_k. In particular, it is impossible that all processes receive their own message first, so there is a process p_i that receives m_j sent by $p_j \neq p_i$ as its first message. Remark that p_j must receive its own message first since receiving $m_k \neq m_j$ first would violate the property for the triplet (p_i, p_j, p_k). Therefore, any process p_k must receive p_j's message first since receiving $m_\ell \neq m_j$ first would violate the property for the triplet (p_j, p_k, p_ℓ). This fact can be exploited to solve consensus between n processes. It follows that imposing an order on the messages sent by triplets of processes provide us with a broadcast whose consensus number is ∞.

5 MBroadcast versus RW Registers

This section first shows that MBroadcast can be implemented on top of RW registers, and then shows that RW registers can be implemented on top of MBroadcast. Hence, MBroadcast and RW registers have the same computability power. It is important to notice that the algorithms described below are independent of t, so they can cope with any number of process crashes.

5.1 From regular RW registers to MBroadcast

This section shows that reliable-MBroadcast can be build on top of RW registers in the presence of asynchrony and process crashes. The algorithm only assumes regular registers, hence it also works on top of atomic registers [35].

Shared memory and local variables.

- The n processes share an array of n SWMR regular registers denoted $SENT[1..n]$ such that, for any i, $SENT[i]$ can be read by any process and written only by p_i. It contains the (initially empty) list of messages mbroadcast by p_i. The first message deposited in $SENT[i]$ will be in position 1, the second in position 2, etc.
- Each process p_i manages an array of local counters (initialized to 0) denoted $delivered_i$ such that, for any $j \neq i$, $delivered_i[j]$ contains the number of messages mdelivered from p_j ($delivered_i[i]$ is not used).

Algorithm 3 Atomic RW register on top of $\mathcal{CAMP}_{n,t}$[MBroadcast].

```
operation write($v$) is                                % code for $p_i$
(1)  synchro_mbroadcast SYNCH();
(2)  let $t_i$ such that $\langle t_i, i\rangle > clock_i$;
(3)  synchro_mbroadcast WRITE($v$, $\langle t_i, i\rangle$)
end operation.

operation read() is
(4)  synchro_mbroadcast SYNCH();
(5)  $v \leftarrow val_i$;
(6)  synchro_mbroadcast WRITE($v, clock_i$);
(7)  return($v$)
end operation.

when WRITE($v, c$) is mdelivered from $p_j$ do
(8)  if $c > clock_i$ then $val_i \leftarrow v$; $clock_i \leftarrow c$ end if.
```

Description of the algorithm. Algorithm 2 builds reliable-MBroadcast on top of n regular RW registers. When a process p_i invokes mbroadcast(m) it adds m at the end of the shared list $SENT[i]$ ($\oplus$ stands for concatenation), invokes the internal uninterruptible routine catch_up() and then locally mdelivers m.

The internal routine catch_up() is repeatedly invoked to allow p_i to mdeliver the messages mbroadcast by the other processes.

▶ **Theorem 2.** *Algorithm* 2 *builds* MBroadcast *on top of* regular RW registers*.* (Proof in Appendix A.)

5.2 From MBroadcast to atomic RW registers

Combined with the previous section, this section shows that MBroadcast and atomic RW registers have the same computational power. The algorithm, closely inspired from the ABD algorithm, builds an MWMR atomic register.

Local variables. Each process p_i manages three local variables.

- val_i contains the current value of the register as known by p_i.
- t_i is the date of the last write issued by p_i.
- $clock_i$ is a pair (timestamp) $\langle date, writer\rangle$ defining the identity of the value in val_i as a timestamp: $writer$ is the identity of the writer of val_i and $date$ the associated Lamport's logical date. Let us remind that all the logical dates are totally ordered according to the usual lexicographical order.

Description of the algorithm. Algorithm 3 implements an MWMR atomic register on top of MBroadcast, i.e., in the model $\mathcal{CAMP}_{n,t}$[MBroadcast].

When p_i invokes write(), it first mbroadcasts a pure control message SYNCH() to resynchronize its local state with respect to possible write operations that modify the last value of the register (line 1). Then, it computes the timestamp associated with value it wants to write (line 2), and propagates its write operation using a synchronized MBroadcast (line 3).

The read operation is similar. Lines 4-5 provide p_i with the value it has to return, while line 6 implements the "reads have to write" strategy needed to ensure atomicity of the read operation [7, 9, 39, 46]. Finally, when p_i mdelivers a WRITE() message, it updates its local context if the new timestamp is higher than the one it currently has in its local variables.

▶ **Theorem 3.** *Algorithm* 3 *implements an* atomic RW register *on top of* $\mathcal{CAMP}_{n,t}$[MBroadcast]. (Proof in Appendix A.)

Suppressing Line 1 (resp. Line 6) builds an SWMR atomic register (resp. an SWMR regular register). Algorithm 3 can also be easily adapted to work with the four types of MWMR regular registers defined in [53].

5.3 A remark on complexity

Although Algorithms 2 and 3 prove that mbroadcast and atomic RW registers have the same computability power, the same cannot be said from a complexity point of view, since n single-writer multi-reader atomic registers must be used and read in Algorithm 2 to implement MBroadcast.

In fact, this complexity is necessary, even with the use of multi-writer multi-reader atomic registers. It was proven in [30], that at least n multi-writer multi-reader atomic registers are necessary to implement an array of one single-writer multi-reader atomic register per process. This lower bound also applies to MBroadcast since, if $k < n$ registers were sufficient to implement MBroadcast, Algorithm 2 could, in turn, be used to simulate the array of n single-writer multi-reader registers. This justifies our introduction of a new abstraction, that allows algorithms with better complexities than read/write registers.

Conversely, it is possible to implement the MBroadcast abstraction in the system model $\mathcal{CAMP}_{n,t}[t < n/2]$ ($\mathcal{CAMP}_{n,t}[\emptyset]$ enriched with the constraint $t < n/2$), using the algorithm presented in [18], whose cost is only $O(n)$ implementation messages when mutual broadcast delivery concerns only correct processes.

5.4 What is actually needed to build a RW register

It follows from the previous results that the operational condition

$$(\text{MP2} \ \vee \ \text{MP3}) \ \ \textbf{or} \ \ (t < n/2)$$

is necessary and sufficient to build an atomic RW register on top of a crash-prone asynchronous message-passing system. It is worth noticing that the first sub-condition MP2 $\vee$ MP3 is on the messages exchanged by each pair of processes (and, for any pair of processes, can change from one message exchange to another one without being explicitly known by the processes) while the other one $t < n/2$ is on global system parameters.

What is captured by MP2 $\vee$ MP3 is the fact that, for each pair of processes, as soon as one of them does not ignore the message from the other one (which is not a pattern captured by MP1), it is possible to build an atomic RW register.

Remark: From consensus on pairs of processes to multi-writer multi-reader registers. Since PBroadcast is stronger than MBroadcast, the implementation of PBroadcast in the model $\mathcal{CAMP}_{n,t}[\text{consensus}_2]$ presented in Appendix C also provides an implementation of MBroadcast that can be exploited to implement an atomic register (using Algorithm 3), in any message-passing system where consensus is available between any pair of processes. In particular, the assumption that $t < \frac{n}{2}$ is not required in this case. Remark that this fact could have been previously established by using the consensus between two processes to implement single-reader single-writer registers, that have the same computability power as multi-writer multi-reader registers [35], but, to our knowledge, it had never been stated explicitly.

Algorithm 4 An MBroadcast-based version of Lamport's Bakery algorithm.

```
operation acquire() is                    % code for $p_i$
(1)  fifo_synchro_mbroadcast HELLO();
(2)  let $t_i$ such that $\langle t_i, i\rangle > \mathsf{max}(tickets_i)$;
(3)  fifo_synchro_mbroadcast TICKET($t_i$);
(4)  wait$\big(\langle t_i, i\rangle = \mathsf{min}(ticket_i)\big)$
end operation.

operation release() is
(5)  fifo_broadcast GOODBYE($t_i$)
end operation.

when HELLO() is fifo-mdelivered from $p_j$ do
(6)  $ticket_i \leftarrow ticket_i \cup \langle 0, j\rangle$.

when TICKET($t$) is fifo-mdelivered from $p_j$ do
(7)  $ticket_i \leftarrow \big(ticket_i \setminus \{\langle 0, j\rangle\}\big) \cup \{\langle t, j\rangle\}$.

when GOODBYE($t$) is fifo-delivered from $p_j$ do
(8)  $ticket_i \leftarrow ticket_i \setminus \{\langle t, j\rangle\}$.
```

6 MBroadcast in Action: Mutex

Preliminary remark. As announced in the Introduction, this section and the next section illustrate uses of the MBroadcast abstraction, where design simplicity is considered as a first class criterion. As already said, it is important to notice that none of the algorithms presented below is explicitely based on quorums. Moreover, as the reader will see, the mutex algorithm and the consensus algorithm have the same structure as Algorithm 3.

6.1 Mutex

Considering the asynchronous message-passing system (i.e. the computing model $\mathcal{CAMP}_{n,t}[\emptyset]$) this section addresses the mutual exclusion problem (mutex). This problem was introduced by E.W. Dijkstra in 1965 [20]. From a historical point of view it is the first distributed computing problem (see [51] for a historical survey on RW-based mutex algorithms). It is defined by two operations denoted acquire() and release(), that are used to bracket a section of code, called *critical section*, such that the following properties are satisfied.

- Safety. At most one process at a time can be executing the critical section.
- Liveness. If a process invokes acquire(), it eventually enters the critical section.

One of the most famous mutex algorithms is the Bakery algorithm due to L. Lamport [33, 37]. This algorithm is based on non-atomic RW registers. This section presents an MBroadcast-based re-writing of this algorithm suited to the asynchronous message-passing communication model.

6.2 An MBroadcast-based rewriting of Lamport's Bakery algorithm

Local variables. Each process p_i manages two local variables.

- An initially empty set $ticket_i$ that will contain timestamps $\langle t, id\rangle$ where t is a ticket number and id a process identity.
- The variable t_i contains the last ticket number used by p_i.

Description of the algorithm. Algorithm 4 is an MBroadcast-based version of Lamport's Bakery algorithm [33]. When a process p_i invokes acquire(), it first invokes fifo_synchro_mbroadcast HELLO() to make public the fact that it starts competing to access the critical section (line 1). When this synchronized fifo_synchro_mbroadcast() invocation terminates, p_i knows that processes that will start competing afterwards are informed it started competing. Process p_i then computes a ticket number higher than all the ticket numbers it knows (line 2) and invokes again fifo_synchro_mbroadcast to inform the processes that its request for the critical section is timestamped $\langle ticket_i, i\rangle$ (line 3). Finally, p_i waits until its request has the smallest timestamp (according to lexicographical order) (line 4).

When a process p_i fifo-mdelivers the message HELLO() from process p_j, it adds the pair $\langle 0, j\rangle$ to its set $ticket_i$, which registers the requests of all the competing processes as known by p_i (line 6);

When a process p_i fifo-mdelivers the message TICKET(t) from process p_j, it replaces the pair $\langle 0, j\rangle$ by the pair $\langle t, j\rangle$ in its local $ticket_i$, which now stores the request of p_j with its competing timestamp (line 7).

When a process p_i invokes release(), it invokes synchro_mbroadcast GOODBYE() to inform the other processes it is no longer interested in the critical section. (line 5). Consequently when a process p_i fifo-delivers a message GOODBYE() from a process p_j, it updates accordingly its local set of requests $ticket_i$ (line 8).

▶ **Theorem 4.** *Considering the system model $\mathcal{CAMP}_{n,t}$[MBroadcast], Algorithm 4 ensures that there is at most one process at a time in the critical section (safety), and that if no process crashes while executing* acquire()*,* release()*, or the code inside the critical section, then all invocations of* acquire() *and* release() *terminate (liveness).* (Proof in Appendix A.)

Remark. Using the failure detector introduced in [17], it is possible to implement mutex in the presence of process crashes occurring while a process is executing acquire(), release(), or the code inside the critical section.

7 MBroadcast in Action: Consensus

The consensus problem was introduced by Lamport, Shostak and Pease [38, 44] in the context of synchronous systems prone to Byzantine process failures. As stated previously, here we consider it in the context of asynchrony and process crashes. Let us recall that consensus is a fundamental problem that lies at the center of the set of distributed agreement problems.

7.1 Definition

Consensus is defined by a single one-shot operation denoted propose() that takes a value as input parameter and returns a value as result. When a process invokes propose(v), we say it proposes v. If the returned value is w, we say the process decides w. Consensus is defined by the following three properties. Validity: If a process decides v, some process proposed v. Agreement: No two processes decide different values. Termination: If a process does not crash, it decides a value.

7.2 Enriching the model with additional computability power

It is well-known that consensus cannot be solved in asynchronous distributed systems where even a single process may crash, be the underlying communication medium message-passing [23] or RW registers [40]. We consider here that this additional computability power

Algorithm 5 An MBroadcast-based variant of Lamport's Paxos algorithm.

```
operation propose($v_i$) is                                          % code for $p_i$
(1)   $val_i \leftarrow v_i$;
(2)   while ($decided_i = \bot$) do
(3)         if (leader() $= i$) then
(4)            let $t_i$ such that $\langle t_i, i\rangle > clock_i$;
(5)            synchro_mbroadcast BEGIN($t_i$);
(6)            if $\langle t_i, i\rangle = clock_i$ then synchro_mbroadcast VOTED($t, val_i$) end if;
(7)            if $\langle t_i, i\rangle = clock_i$ then reliable_broadcast SUCCESS($val_i$) end if
(8)         end if
(9)   end while;
(10)  return($decided_i$)
end operation.

when BEGIN($t$) is mdelivered from $p_j$ do
(11)  if $\langle t, j\rangle > clock_i$ then $clock_i \leftarrow \langle t, j\rangle$ end if.

when VOTED($t, v$) is mdelivered from $p_j$ do
(12)  if $\langle t, j\rangle > clock_i$ then $clock_i \leftarrow \langle t, j\rangle$; $val_i \leftarrow v$ end if.

when SUCCESS($v$) is reliable-delivered from $p_j$ do
(13)  $decided_i \leftarrow v$.
```

is given by the failure detector denoted Ω, which is the weakest failure detector with which consensus can be solved [11]. Ω provides the processes with a single operation denoted leader(). This operation has no input, and each of its invocations returns a process identifier. It is defined by the following property. Eventual leadership: In any execution, there exists a process identifier j such that (1) p_j is a correct process and (2) the number of times leader() returns an identifier $k \neq j$ to any process is finite.

7.3 An MBroadcast-based variant of the Paxos consensus algorithm

Local variables. Each process p_i manages three local variables.

- $decided_i$, initialized to $\bot$ (a default value that cannot be proposed to consensus), will contain the decided value.
- t_i is a scalar logical time (its initial value is irrelevant). Each round of the algorithm initiated by p_i is uniquely identified by a timestamp $\langle t_i, i\rangle$, that plays the same role as the ballot number in Lamport's article. Recall that any two such pairs can be ordered by lexicographical order.
- $clock_i$, initialized to $\langle 0, 0\rangle$, contains the timestamp identifying the current round.

Description of the algorithm. Algorithm 5 solves the consensus problem in the system model $\mathcal{CAMP}_{n,t}[\text{MBroadcast}, \Omega]$. When a process p_i invokes propose(v_i) it first initializes val_i (its current local estimate of the decided value) to v_i (line 1). It then enters a loop it will exit when $decided_i$ is different from $\bot$ (lines 2-9). If $decided_i \neq \bot$, it decides it (line 10). Otherwise, p_i checks if it is the leader by calling leader() on Ω (line 3). If it is not, it re-enters the while loop. If leader() $= i$, p_i competes to impose its current estimate to be the decided value. To this end, it first computes a new timestamp $\langle t_i, i\rangle$ greater than any timestamp it knows (line 4) (this timestamp identifies its current competition to impose a decided value) and invokes synchro_mbroadcast BEGIN(t_i) to inform the other processes it starts competing (line 5). Process p_i then checks if the timestamp $\langle t_i, i\rangle$ is equal to $clock_i$ (line 6). If it is not the case, it aborts the competition. Otherwise, it invokes synchro_mbroadcast VOTED(t, val_i) (line 7) to inform the processes that it champions val_i timestamped $\langle t_i, i\rangle$. Then, it checks again that $\langle t_i, i\rangle = clock_i$, and aborts its competition if it is not. It then discovers that it has won the competition and informs the other processes that val_i is the decided value (line 7). Let us notice that between its reads at line 6 and line 7, the value of $clock_i$ may have been modified. When p_i mdelivers BEGIN(t) or VOTED(t, v) from a process p_j, it updates its local variables accordingly. It does the same when it delivers the message SUCCESS(v).

▶ **Theorem 5.** *Algorithm* 5 *solves* consensus *in* $\mathcal{CAMP}_{n,t}$[MBroadcast,Ω]. (Proof in [19].)

8 Conclusion

Having a better understanding of the power and weaknesses of the basic communication mechanisms provided by asynchronous crash-prone distributed systems is a central issue of distributed computing. The aim of this article was to be a step in such an approach. To this end, the article investigated basic relations linking send/receive message patterns and read/write patterns. It introduced three basic message patterns, each involving a pair of processes, and showed that only two of these patterns have a RW counterpart. This gives a new and deeper view of the different ways processes communicate (and consequently cooperate) in RW systems and in message-passing systems. Then the article introduced a new message-passing communication abstraction denoted Mutual Broadcast, the computability power of which is the same as the one of RW registers. Its main property (called *mutual ordering*) is the fact that, for each pair of processes p and p', if p broadcasts a message m and p' broadcasts a message m', it is not possible for p to deliver first (its message) m and then m' while p' delivers first (its message) m' and then m. In a very interesting way, it appears that this implicit synchronization embedded in the MBroadcast abstraction allows for the design of simple algorithms in which, among other properties, no notion of quorums is explicitly required. The simplicity of MBroadcast-based algorithms has been highlighted with examples including simple re-writing of existing algorithms such Lamport's Bakery and Paxos algorithms.

Nevertheless the quest for the Grail of a deeper understanding of message-passing systems is not complete. On the side of shared memory systems an apex has been attained with Herlihy's consensus hierarchy and its extensions [2, 13, 27, 45, 48]. The same has not yet been achieved for message-passing systems: is there a consensus hierarchy based on specific broadcast abstractions? If the answer is "yes", which is this hierarchy? The case for the consensus number ∞ is total order broadcast [12], the case for the consensus number 2 is PBroadcast, but no specific broadcast abstraction is known for each consensus number $x \in [3..+\infty)$. We conjecture that there are no such specific broadcast abstractions.[4]

References

1 Yehuda Afek, Danny Dolev, Hagit Attiya, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. In *Proc. of the 9th Annual ACM Symposium on Principles of Distributed Computing, Quebec, Canada, August 22-24, 1990*, pages 1–13, 1990.

2 Yehuda Afek, Faith Ellen, and Eli Gafni. Deterministic objects: Life beyond consensus. In *Proc. of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, pages 97–106, 2016.

[4] Let us remind that, it has been shown in [15] that the weakest failure detector to implement Test&Set and Compare&Swap in asynchronous shared-memory systems prone to any number of crashes is the same failure detector (namely, Ω).

3 Yehuda Afek, Eytan Weisberger, and Hanan Weisman. A completeness theorem for a class of synchronization objects (extended abstract). In *Proc. of the Twelth Annual ACM Symposium on Principles of Distributed Computing, Ithaca, New York, USA, August 15-18, 1993*, pages 159–170, 1993.

4 Mustaque Ahamad, Gil Neiger, James E Burns, Prince Kohli, and Phillip W Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

5 James H. Anderson. Multi-writer composite registers. *Distributed Comput.*, 7(4):175–195, 1994.

6 Hagit Attiya. Efficient and robust sharing of memory in message-passing systems. *J. Algorithms*, 34(1):109–127, 2000.

7 Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, 1995.

8 Hagit Attiya, Maurice Herlihy, and Ophir Rachman. Atomic snapshots using lattice agreement. *Distributed Computing*, 8(3):121–132, 1995.

9 Hagit Attiya and Jennifer L. Welch. *Distributed computing - fundamentals, simulations, and advanced topics (2. ed.)*. Wiley series on parallel and distributed computing. Wiley, 2004.

10 Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, 1987.

11 Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, 1996.

12 Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

13 Eli Daian, Giuliano Losa, Yehuda Afek, and Eli Gafni. A wealth of sub-consensus deterministic objects. In *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018*, volume 121 of *LIPIcs*, pages 17:1–17:17, 2018.

14 Luciano Freitas de Souza, Petr Kuznetsov, Thibault Rieutord, and Sara Tucci Piergiovanni. Accountability and reconfiguration: Self-healing lattice agreement. In *25th International Conference on Principles of Distributed Systems, OPODIS 2021, December 13-15, 2021, Strasbourg, France*, volume 217 of *LIPIcs*, pages 25:1–25:23, 2021.

15 Carole Delporte-Gallet, Hugues Fauconnier, and Rachid Guerraoui. Tight failure detection bounds on atomic object implementations. *J. ACM*, 57(4):22:1–22:32, 2010.

16 Carole Delporte-Gallet, Hugues Fauconnier, Sergio Rajsbaum, and Michel Raynal. Implementing snapshot objects on top of crash-prone asynchronous message-passing systems. *IEEE Trans. Parallel Distributed Syst.*, 29(9):2033–2045, 2018.

17 Carole Delporte-Gallet, Hugues Fauconnier, and Michel Raynal. On the weakest information on failures to solve mutual exclusion and consensus in asynchronous crash-prone read/write systems. *J. Parallel Distributed Comput.*, 153:110–118, 2021.

18 Mathilde Déprés, Achour Mostéfaoui, Matthieu Perrin, and Michel Raynal. Brief announcement: The mbroadcast abstraction. In *Proc. of the 2023 ACM Symposium on Principles of Distributed Computing, PODC 2023, Orlando, FL, USA, June 19-23, 2023*, pages 282–285, 2023.

19 Mathilde Déprés, Achour Mostéfaoui, Matthieu Perrin, and Michel Raynal. Send/Receive Patterns versus Read/Write Patterns: the MB-Broadcast Abstraction (Extended Version). Research report, University of Nantes, 2023. URL: `https://hal.archives-ouvertes.fr/hal-04087447`.

20 Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, 1965.

21 Partha Dutta, Rachid Guerraoui, Ron R. Levy, and Marko Vukolic. Fast access to distributed atomic memory. *SIAM J. Comput.*, 39(8):3752–3783, 2010.

22 Jose M. Faleiro, Sriram K. Rajamani, Kaushik Rajan, G. Ramalingam, and Kapil Vaswani. Generalized lattice agreement. In Darek Kowalski and Alessandro Panconesi, editors, *ACM*

Symposium on Principles of Distributed Computing, PODC '12, Funchal, Madeira, Portugal, July 16-18, 2012, pages 125–134. ACM, 2012.

23 Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

24 Michael J. Fischer and Michael Merritt. Appraising two decades of distributed computing theory research. *Distributed Comput.*, 16(2-3):239–247, 2003.

25 Chryssis Georgiou, Theophanis Hadjistasi, Nicolas Nicolaou, and Alexander A. Schwarzmann. Implementing three exchange read operations for distributed atomic storage. *J. Parallel Distributed Comput.*, 163:97–113, 2022.

26 Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report Tech Report 94-1425, Cornell University, 1994. Extended version of "Fault-Tolerant Broadcasts and Related Problems" in *Distributed systems, 2nd Edition*, Addison-Wesley/ACM, pp. 97-145 (1993.

27 Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

28 Maurice Herlihy, Sergio Rajsbaum, Michel Raynal, and Julien Stainer. From wait-free to arbitrary concurrent solo executions in colorless distributed computing. *Theor. Comput. Sci.*, 683:1–21, 2017.

29 Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

30 Damien Imbs, Petr Kuznetsov, and Thibault Rieutord. Progress-space tradeoffs in single-writer memory implementations. In *21st International Conference on Principles of Distributed Systems, OPODIS 2017, Lisbon, Portugal, December 18-20, 2017*, volume 95 of *LIPIcs*, pages 9:1–9:17, 2017.

31 Damien Imbs, Achour Mostéfaoui, Matthieu Perrin, and Michel Raynal. Which broadcast abstraction captures k-set agreement? In Andréa W. Richa, editor, *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, volume 91 of *LIPIcs*, pages 27:1–27:16, 2017.

32 Damien Imbs, Achour Mostéfaoui, Matthieu Perrin, and Michel Raynal. Set-constrained delivery broadcast: A communication abstraction for read/write implementable distributed objects. *Theor. Comput. Sci.*, 886:49–68, 2021.

33 Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

34 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

35 Leslie Lamport. On interprocess communication. part I: basic formalism. *Distributed Comput.*, 1(2):77–85, 1986.

36 Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

37 Leslie Lamport. Deconstructing the bakery to build a distributed state machine. *Commun. ACM*, 65(9):58–66, 2022.

38 Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

39 Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

40 Loui M.C. and Abu-Amara H.H. *Memory requirements for agreement among unreliable asynchronous processes*. Advances in Computing Research, 4:163-183. JAI Press, 1987.

41 Anshuman Misra and Ajay D. Kshemkalyani. Solvability of byzantine fault-tolerant causal ordering problems. In Mohammed-Amine Koulali and Mira Mezini, editors, *Proc. of the 10th International Conference on Networked Systems, NETYS 2022, Virtual Event*, volume 13464 of *Lecture Notes in Computer Science*, pages 87–103. Springer, 2022.

42 Achour Mostéfaoui and Michel Raynal. Two-bit messages are sufficient to implement atomic read/write registers in crash-prone systems. In George Giakkoupis, editor, *Proc. of the 2016*

ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016, pages 381–389. ACM, 2016.

43 Achour Mostéfaoui, Michel Raynal, and Matthieu Roy. Time-efficient read/write register in crash-prone asynchronous message-passing systems. *Computing*, 101(1):3–17, 2019.

44 Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

45 Matthieu Perrin, Achour Mostéfaoui, Grégoire Bonin, and Ludmila Courtillat-Piazza. Extending the wait-free hierarchy to multi-threaded systems. *Distributed Computing*, 35(4):375–398, 2022.

46 Michel Raynal. *Concurrent Programming - Algorithms, Principles, and Foundations*. Springer, 2013.

47 Michel Raynal. *Fault-Tolerant Message-Passing Distributed Systems - An Algorithmic Approach*. Springer, 2018.

48 Michel Raynal. *Concurrent crash-prone shared memory systems: a few theoretical notions*. Morgan & Claypool Publishers, 2022.

49 Michel Raynal. Mutual exclusion vs consensus: both sides of the same coin? *Bull. EATCS*, 140, 2023.

50 Michel Raynal, André Schiper, and Sam Toueg. The causal ordering abstraction and a simple way to implement it. *Information Process. Letters*, 39(6):343–350, 1991.

51 Michel Raynal and Gadi Taubenfeld. A visit to mutual exclusion in seven dates. *Theor. Comput. Sci.*, 919:47–65, 2022.

52 Eric Ruppert. Implementing shared registers in asynchronous message-passing systems. In *Encyclopedia of Algorithms - 2008 Edition*. Springer, 2008.

53 Cheng Shao, Jennifer L. Welch, Evelyn Pierce, and Hyunyoung Lee. Multiwriter consistency conditions for shared memory registers. *SIAM J. Comput.*, 40(1):28–62, 2011.

A Missing proofs

A.1 From atomic RW registers to MBroadcast

▶ **Theorem 2.** *Algorithm* 2 *builds* MBroadcast *on top of* regular RW registers*.*

Proof. Let us consider an execution of Algorithm 2. We prove each property of reliable-mutual-broadcast.

Proof of the Validity property. Suppose p_i mutual-delivers m from p_j. If $i = j$, this happens on line 3, so m was mbroadcast by p_i. Otherwise, it happens on line 6, so m was read on line 5 and written on line 1 by p_j, that mbroadcast m.

Proof of the No-duplication property. Suppose p_i mdelivers twice the k^{th} message mbroadcast by p_j. By Lines 6-7, at the second mdelivery we have $delivered_i[j] \geq k$, which is impossible by line 6.

Proof of the Local termination property. All operations of Algorithm 2 terminate because they do not contain while loops or recursive calls, and messages that are mbroadcast by p_i are delivered by p_i on line 3.

Proof of the Global CF-Termination property. Suppose p_i mdelivers a message m from p_j. Then p_j inserted it in $SENT[j]$ and never deleted it. Therefore all correct processes eventually mdeliver m when later they execute `catch_up`().

Proof of the Global CS-Termination property. It is a direct consequence of Global CF-Termination.

Proof of the Mutual ordering property. Suppose that p_i mbroadcasts m_i and p_j mbroadcasts m_j. At least one of the two scenarios must happen:

- p_i completes its write on $SENT[i]$ (line 1) before p_j starts its read on $SENT[i]$ (line 5). In that case, since $SENT[i]$ is regular, p_j reads m_i from $SENT[i]$ and mdelivers it during its execution of catch_up() (line 2), before mdelivering m_j on line 3.
- Or p_j completes its write on $SENT[j]$ (line 1) before p_i starts its read on $SENT[j]$ (line 5), so p_i mdeliver m_j before m_i. ◀

A.2 From MBroadcast to atomic RW registers

▶ **Theorem 3.** *Algorithm* 3 *implements an* atomic RW register *on top of* $\mathcal{CAMP}_{n,t}$[MBroadcast].

Proof. Let us first remark that Algorithm 3 contains no loop, so all its operations terminate.

Let us consider an execution admitted by Algorithm 3. For each operation o, we define the timestamp $ts(o)$ of o as follows. If o is a write by p_i, then $ts(o) = \langle t_i, i\rangle$ at the end of line 2. If o is a read by p_i, then $ts(o) = clock_i$ at the beginning of line 6. In other words, $ts(o)$ is the timestamp of the value that is read or written by o. We also define the binary relation $\rightarrow$ between operations as $o_1 \rightarrow o_2$ if either 1) o_1 was terminated before o_2 was started (denoted by $o_1 \rightarrow_1 o_2$), or 2) o_2 is a write and $ts(o_1) < ts(o_2)$ (denoted by $o_1 \rightarrow_2 o_2$), or 3) o_1 is a write, o_2 is a read, and $ts(o_1) \leq ts(o_2)$ (denoted by $o_1 \rightarrow_3 o_2$).

Let us first notice that, if $o_1 \rightarrow o_2$, then $ts(o_1) \leq ts(o_2)$, and if moreover o_2 is a write, then $ts(o_1) < ts(o_2)$. This is true by definition for $\rightarrow_2$ and $\rightarrow_3$. For $\rightarrow_1$, the first part is a direct consequence of the blocking-mutual-ordering property between the `Write` message sent at the end of o_1 (line 3 or 6) and the SYNCH message sent at the beginning of o_2 (line 1 or 4). Moreover, if o_2 is a write, then $ts(o_2) > ts(o_1)$ by line 2.

Let us prove that $\rightarrow$ is cycle-free. Indeed, suppose there is a cycle $o_1 \rightarrow o_2 \rightarrow \ldots \rightarrow o_k = o_1$ containing at least two operations. By what precedes, all operations in the cycle have the same timestamp, hence there cannot be any write operation. Moreover, there cannot be only reads, because they would be ordered only by $\rightarrow_1$ which is cycle-free.

Finally, the reflexive and transitive closure of $\rightarrow$ can be extended into a total order that respects real time thanks to $\rightarrow_1$, and such that any read returns the initial value if its timestamp is $\langle 0, 0\rangle$, or the value written by the preceding write since val_i and $clock_i$ are updated jointly on line 8, thanks to $\rightarrow_2$ and $\rightarrow_3$. Hence, the algorithm is linearizable. ◀

A.3 Proof of the Mutex algorithm

▶ **Theorem 4.** *Considering the system model* $\mathcal{CAMP}_{n,t}$[MBroadcast]*, Algorithm* 4 *ensures that there is at most one process at a time in the critical section (safety), and that if no process crashes while executing* acquire()*,* release()*, or the code inside the critical section, then all invocations of* acquire() *and* release() *terminate (liveness).*

Proof.

Proof of the safety property. Suppose, by contradiction, that two processes p_i and p_j are in the critical section at the same time, and let t_i and t_j be their respective order values after line 2. Without loss of generality, let us suppose that $\langle t_i, i\rangle < \langle t_j, j\rangle$. Two cases are consistent with the Mutual Ordering property applied to p_i's HELLO() message and p_j's TICKET(t_j) message.

- If p_i receives TICKET(t_j) before it fifo-mbroadcasts HELLO(), then $\langle t_j, j\rangle \in \mathit{ticket}_i$ after line 2 thanks to line 7, and, due to fifo ordering, this will remain true as long as p_j remains in critical section. Then, p_i picks t_i such that $\langle t_i, i\rangle > \langle t_j, j\rangle$ (line 2). This is in contradiction with the fact that $\langle t_i, i\rangle < \langle t_j, j\rangle$.
- If p_j fifo-mdelivers HELLO() from p_i before it fifo-mbroadcasts TICKET(t_j), then, when p_j entered the critical section, either $\langle 0, i\rangle \in \mathit{ticket}_j$ (by line 6) or $\langle t_i, i\rangle \in \mathit{ticket}_j$ (by line 7), due to the fifo ordering. In both cases, this contradicts the fact that $\langle t_j, j\rangle = \min(\mathit{ticket}_j)$ (line 4).

Proof of the liveness property. Let us observe that the release() operation has no loop, and the acquire() operation has a single wait() statement (line 4). Suppose, by contradiction, that some process p_i remains forever blocked at line 4. Without loss of generality, let us assume that its timestamp is the smallest one, (i.e. all processes with a lower timestamp eventually entered and exited the critical section).

After all correct processes have received the message TICKET(t_i) sent by p_i (line 3), all processes p_j that execute line 2 pick a value t_j such that $\langle t_j, j\rangle > \langle t_i, i\rangle$. In other words, a finite number of timestamps $\langle t_j, j\rangle < \langle t_i, i\rangle$ are ever picked in the execution, and, by minimality of $\langle t_i, i\rangle$, all these timestamps let their process enter the critical section. As all critical sections terminate, they also all leave critical section and send a GOODBYE() message, that is eventually received by p_i. After p_i has received all these messages, a finite number of processes p_j will call acquire() again and pick t_j such that $\langle t_j, j\rangle > \langle t_i, i\rangle$. When p_i has received all the respective $\mathit{ticket}(t_j)$ messages, it will have $\langle t, i\rangle = \min(\mathit{ticket}_i)$. A contradiction. ◀

B MBroadcast in Action: Lattice Agreement

One of the very first articles on the use of lattices to solve distributed computing problems is [8] where a snapshot object is built from a lattice data structure. Later developments appeared in [22]. More recently, lattice agreement has been used as a building block to solve accountability and reconfiguration issues encountered in distributed computing [14].

B.1 Definition

A bounded join-semilattice $(L, \bot, \sqsubseteq, \sqcup)$ is composed of a set L of elements partially ordered according to a relation $\sqsubseteq$, such that there is a smallest element $\bot$ and for all $x, y \in L$, there exists a least upper bound of x and y, denoted by $x \sqcup y$.

Lattice agreement is similar to consensus, namely each process may propose a value and decide a value. It is defined by the following properties.

Validity. The value decided by a process p_i is the least upper bound of a subset of the proposed values and contains the value proposed by p_i.
Consistency. If p_i decides x_i and p_j decides x_j, then $x_i \sqsubseteq x_j$ or $x_j \sqsubseteq x_i$.
Termination. If a process does not crash, it decides a value.

Algorithm 6 Lattice agreement in $\mathcal{CAMP}_{n,t}$[MBroadcast].

```
operation propose($v_i$) is                        % code for $p_i$
(1)  repeat $prev_i \leftarrow view_i$;
(2)           synchro_mbroadcast STATE($view_i \sqcup v_i$)
(3)  until ($prev_i = view_i$);
(4)  return($view_i$)
end operation.

when STATE($v$) is mdelivered from $p_j$ do
(5)  $view_i \leftarrow view_i \sqcup v$.
```

B.2 An MBroadcast-based lattice agreement algorithm

Local variables. Each process p_i manages two local variables.

- $view_i$ contains the current view of the value that p_i will return. It is initialized to $\perp$.
- $prev_i$ is an auxiliary variable.

Description of the algorithm. Algorithm 6 solves lattice agreement in the system model $\mathcal{CAMP}_{n,t}$[MBroadcast]. It is based on the classical double-scan principle. When a process p_i invokes propose(v_i) it repeatedly mbroadcasts its current view of the decided value enriched with the value v_i it proposes, until $view_i$ has not been modified since the previous mbroadcast. When it mdelivers a message STATE(v), p_i updates its local view $view_i$. Let us notice that $view_i$ can be updated before p_i invokes propose(v_i).

▶ **Theorem 6.** *Algorithm* 6 *solves* lattice agreement *in the system model* $\mathcal{CAMP}_{n,t}$[MBroadcast]. (Proof in [19].)

C On the Computability Side: MP2 Alone Characterizes Test&Set()

C.1 From two-process consensus to PBroadcast

Algorithm 7 implements PBroadcast on top of consensus between two processes. It is inspired by the implementation of the total-order broadcast given in [12]. The main difference lies in the fact that the messages are not globally ordered by all processes, but rather by each pair of processes independently.

As far as shared objects are concerned, for all i and $j \neq i$, $CONSENSUS\{i,j\}$ denotes an unbounded sequence of consensus instances between p_i and p_j. The k^{th} element of this sequence is denoted $CONSENSUS\{i,j\}[k]$.

Moreover, each process p_i manages two local variables.

- $delivered_i$ is the set of all messages that p_i has already pdelivered (initially $\emptyset$).
- $ordered_i$, an array of n integer values (initially 0, $ordered_i[i]$ is not used), such that $ordered_i[j]$ is the number of consensus instances between p_i and p_j in which p_i took part, i.e. the index of the next consensus object $CONSENSUS\{i,j\}$ that can be used by p_i.

Each pair of processes (p_i, p_j) agrees on a sequence of all messages pbroadcast by p_i or p_j, thanks to the sequence of consensus instances saved in $CONSENSUS\{i,j\}$ (Lines 5-6). Notice that it is possible that the same message happens twice in this sequence, in which case only the first occurrence will be considered in the order in which messages are delivered (Lines 9-12).

In order to pbroadcast a message m, a process p_i first broadcasts a message PB(m) on line 1 to ensure that m will eventually win a consensus against messages from p_j. Upon delivery of this message, p_j helps p_i to win a consensus by proposing m as well on $CONSENSUS\{i,j\}$ until some consensus is won by m. It is assumed that there is no concurrency between the execution of pbroadcast(m) and the code associated to the reliable delivery of PB(m).

Algorithm 7 PBroadcast on top of $\mathcal{CAMP}_{n,t}[\text{consensus}_2]$.

```
operation pbroadcast($m$) is                                          % code for $p_i$
(1)   reliable_broadcast PB($m$);
(2)   for $j$ from 1 to $i-1$ and then from $i+1$ to $n$ do order($m, j$) end for;
(3)   deliver($m, i$)
end operation.

when PB($m$) is reliable-delivered from $p_j$
(4)   if $j \neq i$ then order($m, j$); deliver($m, j$) end if.

operation order($m, j$) is
(5)   repeat  $m' \leftarrow CONSENSUS\{i, j\}[ordered_i[j]]$.propose($m$);
(6)           $ordered_i[j] \leftarrow ordered_i[j] + 1$;
(7)           if $m' \neq m$ then deliver($m', j$) end if
(8)   until $m' = m$
end operation.

operation deliver($m, j$) is
(9)   if $m \notin delivered_i$ then
(10)          $delivered_i \leftarrow delivered_i \cup \{m\}$;
(11)          pdelivery of $m$ from $p_j$
(12)  end if
end operation.
```

Then p_i tries to insert its message m in the sequences it shares will all the other processes, until m is the next message it has agreed to pdeliver with all other processes (line 2), and then pdelivers m (line 3).

▶ **Theorem 7.** *Algorithm* 7 *implements* reliable-PBroadcast. (Proof in [19].)

C.2 From PBroadcast to Test&Set()

For simplicity, this section considers one-shot Test&Set(). It can be easily generalized to multi-shot Test&Set().

Definition of Test&Set(). A test&set object is an object that can take only two values `true` or `false`. Its initial value is `true`. It provides the processes with a single one-shot atomic operation denoted Test&Set(), that sets the value of the object to `false` and returns the previous value of the object. As the operation is atomic, its executions can be linearized and the only invocation that returns `true` is the first that appears in the linearization order. From a computability point of view the consensus number of Test&Set() is 2 [3, 27].

A PBroadcast-based implementation of Test&Set() and its proof. Algorithm 8 is a PBroadcast-based implementation of a Test&Set() object. It uses a series of two-process tournaments to elect one and only one winner. To this end each process p_i manages two local variables.

- $round_i$ is an integer between 0 and $\max(2, \lceil \log_2(n) \rceil + 1)$ (initially 0) that represents p_i's current progress.
- $vying_i$ a Boolean, initially `true`, that becomes `false` after p_i has lost a tournament.

The levels of the tournament tree are associated with rounds such that at each round r, two winners from round $r-1$ compete by PBroadcasting a message COMPETE(r) (line 2). Since both processes pdeliver both messages in the same order, the first message decides which process reaches round $r+1$. More precisely, the tournament tree is such that the set of processes is partitioned into subsets of size 2^r, i.e. $\{p_1, ..., p_{2^r}\}$, $\{p_{2^r+1}, ..., p_{2\times 2^r}\}$, ...,

Algorithm 8 Test&Set() on top of $\mathcal{CAMP}_{n,t}$[PBroadcast].

```
operation Test&Set() is                                         % code for $p_i$
(1)  while $round_i < 2 \vee (round_i \leq \lceil \log_2(n) \rceil \wedge vying_i)$ do
(2)      synchro_pbroadcast COMPETE($round_i$)
(3)  end while;
(4)  return($vying_i$)
end operation.

when COMPETE($r$) is pdelivered from $p_j$ do
(5)  if $i = j$ then $round_i \leftarrow r + 1$
(6)  else if $round_i = 0 \wedge r = 1$ then $vying_i \leftarrow$ false
(7)  else if $round_i < r \wedge \left\lfloor \frac{i-1}{2^r} \right\rfloor = \left\lfloor \frac{j-1}{2^r} \right\rfloor$ then $vying_i \leftarrow$ false
(8)  end if.
```

$\left\{p_{\left\lfloor \frac{n-1}{2^r} \right\rfloor \times 2^r + 1}, \ldots, p_n\right\}$. Hence, p_i looses round r if it receives a message COMPETE(r) from another process p_j playing in the same set at round r, i.e. such that $\left\lfloor \frac{i-1}{2^r} \right\rfloor = \left\lfloor \frac{j-1}{2^r} \right\rfloor$, when $round_i < r$ (line 7), which indicates that p_i did not receive its own message COMPETE(r) yet (line 5). Processes play until they loose a round or they win the finale at round $\lceil \log_2(n) \rceil$ (condition $round_i \leq \lceil \log_2(n) \rceil \wedge vying_i$ on line 1). Then, they return the content of their variable $vying_i$, which can be true only for a process that won all its tournaments.

In order to ensure linearizability, Algorithm 8 adds a round 0 to prevent late processes to win if they start their execution after another process played its first tournament: p_i forfeits if it receives a message COMPETE(1) from another process before its own message COMPETE(0) (line 6). In this case, p_i still participates in round 1, so its message COMPETE(1) forces even slower processes to forfeit as well (condition $round_i < 2$ on line 1).

▶ **Theorem 8.** *Algorithm* 8 *implements a wait-free linearizable* Test&Set() *object in the system model* $\mathcal{CAMP}_{n,t}$[PBroadcast]. (Proof in [19].)

Modular Recoverable Mutual Exclusion Under System-Wide Failures

Sahil Dhoked
Department of Computer Science, The University of Texas at Dallas, Richardson, TX, USA

Wojciech Golab
Department of Electrical and Computer Engineering, University of Waterloo, Canada

Neeraj Mittal
Department of Computer Science, The University of Texas at Dallas, Richardson, TX, USA

Abstract

Recoverable mutual exclusion (RME) is a fault-tolerant variation of Dijkstra's classic mutual exclusion (ME) problem that allows processes to fail by crashing as long as they recover eventually. A growing body of literature on this topic, starting with the problem formulation by Golab and Ramaraju (PODC'16), examines the cost of solving the RME problem, which is quantified by counting the expensive shared memory operations called remote memory references (RMRs), under a variety of conditions. Published results show that the RMR complexity of RME algorithms, among other factors, depends crucially on the failure model used: individual process versus system-wide. Recent work by Golab and Hendler (PODC'18) also suggests that *explicit* failure detection can be helpful in attaining *constant* RMR solutions to the RME problem in the system-wide failure model. Follow-up work by Jayanti, Jayanti, and Joshi (SPAA'23) shows that such a solution exists even without employing a failure detector, albeit this solution uses a more complex algorithmic approach.

In this work, we dive deeper into the study of RMR-optimal RME algorithms for the system-wide failure model, and present contributions along multiple directions. First, we introduce the notion of *withdrawing* from a lock acquisition rather than resetting the lock. We use this notion to design a withdrawable RME algorithm with optimal $O(1)$ RMR complexity for both cache-coherent (CC) and distributed shared memory (DSM) models in a *modular* way without using an explicit failure detector. In some sense, our technique marries the simplicity of Golab and Hendler's algorithm with Jayanti, Jayanti and Joshi's weaker system model. Second, we present a variation of our algorithm that supports fully dynamic process participation (*i.e.*, both joining and leaving) in the CC model, while maintaining its constant RMR complexity. We show experimentally that our algorithm is substantially faster than Jayanti, Jayanti, and Joshi's algorithm despite having stronger correctness properties. Finally, we establish an impossibility result for fully dynamic RME algorithms with bounded RMR complexity in the DSM model that are adaptive with respect to space, and provide a wait-free withdraw section.

2012 ACM Subject Classification Theory of computation → Concurrent algorithms

Keywords and phrases mutual exclusion, shared memory, persistent memory, fault tolerance, system-wide failure, RMR complexity, dynamic joining, dynamic leaving

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.17

Funding *Wojciech Golab*: Researcher supported in part by an Ontario Early Researcher Award, a Google Faculty Research Award, as well as the Natural Sciences and Engineering Research Council (NSERC) of Canada.

Acknowledgements We thank the anonymous reviewers for their helpful feedback and insightful suggestions.

1 Introduction

One of the most common techniques to mitigate race conditions in a concurrent system is to use *mutual exclusion (ME)*, which establishes a *critical section (CS)* in which a program can access a shared resource without risking interference from other processes. Correct use of mutual exclusion or *mutex locks* ensures that the system always stays in a consistent state, and produces correct outcomes. The ME problem was first defined by Dijkstra more than half a century ago in [16], later formalized by Lamport [35, 36], and then widely studied in scientific literature (*e.g.*, see [4, 41]) under a variety of assumptions regarding both the degree of synchrony and the set of synchronization primitives available for accessing shared memory. The vast majority of research in this area in recent years emphasizes so-called *local spin* algorithms, which incur a bounded number of *remote memory references* (RMRs) – expensive memory operations that trigger communication on the interconnect joining processors with memory – in each attempt to acquire and release the lock. Whether or not a memory operation incurs an RME depends on the underlying shared memory model – cache-coherent (CC) or distributed shared memory (DSM) (*e.g.*, see [38, 12, 20]).

Over decades of research, the study of shared memory algorithms has started to shift away from traditionally strong modelling assumptions, such as a failure-free execution environment where processes agree ahead of time on a set of named shared objects, toward richer models based on faulty shared memories [1, 40], anonymous systems [42, 47], and faulty processes [7, 8, 23, 39]. Alternative models are particularly interesting and important for the ME problem in asynchronous environments since solutions rely intrinsically on blocking synchronization, and failures can have undesirable and even disastrous ripple effects. For example, if one process crashes in the critical section, or even while acquiring or releasing a mutex lock, several other processes who are contending for the same lock can potentially stall. Although such failures are not common, they can occur in the real world due to software bugs, or even hardware failures in large scale platforms where processing elements are interconnected in complex ways.

The *recoverable mutual exclusion* (RME) problem, formulated recently by Golab and Ramaraju [23, 24], is the most recent attempt to marry resilience against process failures with mathematical rigour in the ongoing study of lock-based synchronization. The RME problem involves designing an algorithm that ensures mutual exclusion under the assumption that processes may fail at *any* point during their execution, either independently or simultaneously. Since a slow process cannot be distinguished reliably from a crashed process in an asynchronous environment [11, 18], one expects informally that an RME algorithm must receive some help from the environment in responding to a failure. The primary mechanism for this in Golab and Ramaraju's model is the crucial assumption that a crashed process[1] eventually recovers and cleans up the internal state of the RME lock by attempting to acquire and release it again, which can be regarded as an implicit form of failure detection. Golab and Hendler's work on system-wide failures [22] adds an explicit epoch-based failure detector that helps synchronize a cohort of recovering processes, which simplifies the RME problem to the point where it can be solved for n processes using common synchronization primitives with (*optimal*) $O(1)$ RMR complexity for both CC and DSM models.

The growing body of work on the RME problem focuses primarily on fundamental correctness properties of RME and techniques for implementing these properties, with little attention paid to how RME locks could be used in practice to implement fault-tolerant data structures. Results are especially sparse for RMR-efficient algorithms in the system-wide

[1] More precisely, a process that crashes outside of the lock's non-critical section.

Algorithm 1 Process execution model.

```
while true do
    Non-Critical Section (NCS)
    Recover();    ⎫
    Enter();      ⎪                    ⎧
    Critical Section (CS)    ⎬    OR    ⎨ Withdraw();
    Exit();       ⎭                    ⎩
```

failure model, which is arguably the more practical failure model; in a well-designed software system, processes are more likely to fail together due to a power outage than individually due to software bugs. Somewhat surprisingly, only two studies have been published in this space so far, namely work by Golab and Hendler (GH) [22], and Jayanti, Jayanti and Joshi (JJJ) [28, 29]. Both works follow roughly the same algorithmic technique, which starts with a conventional mutex algorithm (base mutex), and applies a transformation to reset the base mutex carefully after a system-wide failure. GH is a black box technique that uses a single instance of the base mutex and requires an explicit failure detector (as explained earlier), whereas JJJ avoids the failure detector but uses a somewhat complex arrangement of three instances of a base mutex (with wait-free exit) and provides a weaker fairness guarantee.

In this work, we advance the state of the art with respect to RME locks for system-wide failures along several directions. First, we introduce a new algorithmic technique called *withdrawing* whereby a process can remove itself from a queue of waiting processes upon failure without resetting the entire queue to its initial state. This technique allows us to combine the simplicity of the GH algorithm with the weaker assumptions of the JJJ model. We demonstrate the power of our approach by constructing a novel RME lock with optimal $O(1)$ RMR complexity for both CC and DSM models using the well-known Mellor-Crummey and Scott's (MCS) mutex lock [38, 17] as the building block. Our algorithm requires only one instance of this base mutex and provides first-come-first-served (FCFS) fairness, like GH, and yet does not rely on a failure detector, similar to JJJ. Furthermore, we make the argument that the ability to withdraw is a useful feature for an RME lock to not only use internally, but also expose to the application, and formulate the *withdrawable recoverable mutual exclusion* problem. In particular, we show in Appendix A that withdrawability alone can be sufficient for an application's recovery goals using the example of a lock-based concurrent linked-list presented in [25]. We also advocate for *modular* algorithmic designs that separate optional features like Golab and Ramaraju's critical section re-entry (CSR) property from the core functions of an RME lock. Second, we present a variation of our RME algorithm for the CC model that supports fully dynamic process participation (*i.e.*, support for both joining and leaving) while maintaining its most desirable features. Our algorithmic approach not only allows for a separation of concerns but, as we show experimentally, also gives us a substantial performance advantage over JJJ, which relies crucially on critical section ownership state to achieve correct recovery. Finally, we establish an impossibility result for fully dynamic RME algorithms that are RMR-efficient in the DSM model, adaptive with respect to space, and provide a wait-free withdraw section.

Roadmap. The rest of the text is organized as follows. We present our system model and formulate the withdrawable RME problem in Section 2. We present an RMR-optimal RME lock with a wait-free withdraw section assuming system-wide failures for both CC and DSM models in Section 3, a transformation to add the CSR property in Section 4, and a fully dynamic variant for the CC model in Section 5. Section 6 describes an impossibility result about designing a fully dynamic RME algorithm with a wait-free withdraw section for the DSM model under certain conditions. We discuss related work in Section 7. Section 8 summarizes our conclusions and outlines directions for future research. Discussion of a use case for withdrawable locks, detailed experiments, and an equivalence result for withdrawability and abortability are presented in Appendix A, B, and C, respectively.

2 System Model

Our model is based on [22, 23, 24]. We consider an asynchronous shared memory system in which processes communicate by performing single-word read, write and read-modify-write (RMW) operations on shared variables, and also have access to private variables. Our algorithms use two RMW primitives: (i) Fetch-And-Store (`FAS`), which retrieves the old value and blindly writes a new value, and (ii) Compare-And-Swap (`CAS`), which conditionally writes a new value if the old value matches a given comparison value, and returns a boolean success indicator.

We assume the *crash-recover* failure model, meaning that a process may fail at any time during its execution by crashing, and may recover by resuming execution from the beginning of its program. Failures are *system-wide* [22], meaning that all processes crash simultaneously, as opposed to the individual failure model considered in [23, 24, 15]. Upon crashing, a process loses its call stack, and its private variables (including the program counter) are reset to their initial values; however shared variables are stored in persistent memory and retain their most recently written value prior to the crash.

The execution model of a process with respect to a recoverable lock is depicted in algorithm 1. A process typically executes the NCS (non-critical section), `Recover`, `Enter`, CS (critical section) and `Exit` sections, in that order, and then returns to the NCS. The internal structure of the lock is cleaned up, if needed, inside the `Recover` section, then the `Enter` section is used to acquire the lock, and the `Exit` section releases it. A process can also execute the NCS and `Withdraw` sections, which bypasses the CS and yet allows the internal structure of the lock to be cleaned up after a failure if needed inside the `Withdraw` section. A system-wide failure returns every process to the NCS by resetting its program counter to the initial value.

The `Withdraw` section is a *new feature* in our model, as compared to [23, 24], that provides the application more flexibility in using the recoverable lock. This becomes especially useful when an RME lock is used as a component to build another more advanced RME lock satisfying additional desirable properties, as illustrated in this work; the specific execution path taken by a process depends on the needs of the program using the lock. A trivial way to implement `Withdraw` section is to simply execute the `Recover`, `Enter` and `Exit` sections in order, with an empty CS (*e.g.*, as on Line 50 in Figure 6 of [21]). However, this naïve implementation is inherently blocking, and can cause deadlock in the application if not used carefully. On the other hand, as we show in this work, it is possible to implement the `Withdraw` section efficiently in a *wait-free* manner (*i.e.,* bounded number of steps).

A system execution is modeled as a sequence of process steps called a *history*. In each step, a process performs some local computation affecting only its private variables, and executes at most one shared memory operation. A process p is said to be *live* in a history H if it leaves the NCS at least once, meaning that H contains at least one step by p. We also consider crash steps that represent system-wide failures and do not belong to any process, but we do not model executions of the NCS and CS as steps. The projection of a history H onto a process p, denoted $H|p$, is the maximal subsequence of H comprising all steps of p as well as crash steps. The concatenation of histories G and H is denoted $G \circ H$. An *epoch* is a contiguous subhistory of a history H between (and excluding) two consecutive crash steps.

A *c-passage* of a process p is a sequence of steps by p from the first step of the `Recover` section to the last step of the `Exit` section, or a crash failure, whichever occurs first. A *w-passage* of a process p is a sequence of steps by p from the first step of the `Withdraw` section to the last step of the `Withdraw` section, or a crash failure, whichever occurs first. A *passage* is either a c-passage or a w-passage. A passage is *failure-free* unless a process crashes before completing the last step of the `Exit` or `Withdraw` section. A *super-passage* of a process p is a maximal non-empty sequence of consecutive passages executed by p, where only the last passage in the sequence can be failure-free. A history H is *fair* if it is finite, or it is infinite and every process that is live in H either takes infinitely many steps or stops taking steps after completing a failure-free passage and returning to the NCS.

Two passages *interfere* if their respective super-passages overlap (*i.e.*, neither ends before the other starts). A passage by a process p is *0-failure-concurrent* (0-FC) if p crashes in the respective super-passage. A passage is *k-failure-concurrent* (k-FC), $k > 0$, if it interferes with some $(k-1)$-FC passage (possibly itself). Clearly, k-FC implies $(k+1)$-FC for all $k \geq 0$. Intuitively, the parameter k in the notion of failure-concurrency measures the maximum "distance" of a given passage from any failure, where two interfering passages are said to be at a "distance" of one from each other.

Unless otherwise stated, we allow *dynamic* participation of processes. A *new* process can join the system at run time and use the lock to execute a CS, referred to as *dynamic joining* [28, 29]. As part of joining, a process may need to allocate some memory – shared as well as private – needed to synchronize with other processes. We also consider the dual problem in which any existing process can leave the system at run time after completing a failure-free passage, referred to as *dynamic leaving*. In this case, the departing process needs to know that it can safely reclaim its memory (unless a separate garbage collection mechanism is being used). We say that an algorithm is *fully dynamic* if it supports dynamic joining as well as dynamic leaving. The original MCS algorithm (without wait-free exit) is fully dynamic. The RME algorithms in [28, 29] only support dynamic joining but not dynamic leaving.

2.1 RME Correctness and Other Properties Reformulated

This section summarizes the correctness properties of RME, reformulated as needed to accommodate `Withdraw` section and w-passage. New additions are typeset in *italics*.

Mutual Exclusion (ME) At most one process is in the CS at any point in any history.

Starvation Freedom (SF) Let H be an infinite fair history in which every process fails only a finite number of times during each of its super-passages. If a process p leaves the NCS in some step of H, then p eventually either enters the CS *or begins executing* `Withdraw` *section.*

If a process fails inside the CS, then a shared resource (*e.g.*, a data structure) may be left in an inconsistent state. In such cases, it may be desirable to allow the same process to re-enter CS and "fix" the shared resource, if needed, before any other process can enter the CS (*e.g.*, [24, 21, 22, 27]). This property is referred to as *critical section re-entry (CSR).* We use a stronger variant of CSR in this work, which is defined as:

Bounded Critical Section Reentry (BCSR) For any history H, if a process p crashes inside the CS, then, until p has reentered the CS *or begun executing* `Withdraw`, any subsequent execution of `Recover` and `Enter` sections by p either completes within a bounded number of p's own steps or ends with p crashing.

In addition to the above qualitative properties, it is also desirable for an RME algorithm to satisfy the following:

Bounded Exit (BE) For any infinite history H, any execution of `Exit` by any process p either completes in a bounded number of p's own steps or ends with p crashing.

Bounded Recovery (BR) For any infinite history H, any execution of `Recover` by process p either completes in a bounded number of p's own steps or ends with p crashing.

Bounded Withdraw (BW) *For any infinite history H, any execution of* `Withdraw` *by process p either completes in a bounded number of p's own steps or ends with p crashing.*

Note that the `Withdraw` section can in some cases serve an application's recovery goals better than Golab and Ramaraju's CSR property [23, 24], which we reformulated earlier. As a specific example, consider the linked list in Chapter 9 of Herlihy and Shavit [25], which we reproduce in Appendix A along with a discussion of recoverability. The `Add` and `Remove` methods both use hand-over-hand locking to traverse the list, and both operations take effect atomically at a single line of code that writes a pointer (Line 163 and Line 178, respectively). Since garbage collection is used to reclaim list nodes, strict linearizability [2, 6] can be achieved easily by simply repairing the locks (via either `Recover`/`Enter`/`Exit` or `Withdraw`). Re-entering the CS in this case is unnecessary since the linked list structure itself cannot be corrupted by a crash failure, not to mention that the application would need to determine specifically which linked list operation was interrupted by a failure to reach the correct CS as each node-level lock protects multiple critical sections.

Many applications require a lock to provide some guarantees about fairness. Intuitively, a fairness property imposes a constraint on when and/or how often a process trying to enter the CS can be overtaken by another process. Definitions of such properties refer to a *doorway*, which is a *bounded* prefix of the `Enter` section, and intuitively determines the order in which processes acquire the lock. We consider the following two notions of fairness, the first of which has novel formulation for the system-wide failure model:

First-Come-First-Served (FCFS) For any two *concurrent* c-passages π and π' belonging to processes p and p', respectively, if p completes its doorway in π before p' starts its doorway in π', then p' does not enter the CS in π' before p enters the CS in π.

k-First-Come-First-Served (k-FCFS), where $k \geq 0$ For any two c-passagees π and π' belonging to processes p and p', respectively, where neither passage is k-FC, if p completes its doorway in π before p' starts its doorway in π', then p' does not enter the CS in π' before p enters the CS in π.

Intuitively, FCFS imposes constraints on all c-passages, whereas k-FCFS imposes constraints only on those c-passages that are sufficiently far away from any failure. Note that FCFS implies 0-FCFS and k-FCFS implies $(k + 1)$-FCFS for all $k \geq 0$. Further, FCFS and (B)CSR are mutually incompatible properties. An RME algorithm can satisfy at most one of these properties; but it can simultaneously satisfy k-FCFS, for some k, as well as BCSR.

2.2 Complexity Measures

In terms of complexity measures, we are concerned in this work with time and space. Time complexity is quantified by counting *remote memory references* (RMRs), which are defined in an architecture-dependent manner. In the *cache-coherent* (CC) model, we conservatively count every shared memory operation as an RMR, except where a process p reads a variable that p has already read earlier, and which has not been updated (*i.e.*, accessed by any means other than a read) since p's most recent read. In the *distributed shared memory* (DSM)

Algorithm 2 MCS algorithm with wait-free exit (adapted from [17, 15]).

```
1  struct QNode {
2      locked: boolean variable, initially FALSE;
3      next: reference to QNode, initially null;
4  };
5  global shared variables
6      TAIL: reference to QNode, initially null;
7  per-process shared/persistent variables
8      POOL_p: array [0, 1] of QNode, all elements
               initially {FALSE, null};
9      MINE_p: reference to QNode, initially &POOL[0];
10     CURR_p: integer variable ∈ {0, 1}, initially 0;
11 private variables
12     pred: reference to QNode;

13 Procedure Enter()
14     MINE_p := &POOL_p[CURR_p];
15     MINE_p.locked := TRUE;
16     MINE_p.next := null;
17     pred := FAS(TAIL, MINE_p);
18     if pred = null then return;
19     if CAS(pred.next, null, MINE_p) then
20         await ¬MINE_p.locked;
21 Procedure Exit()
22     if CAS(TAIL, MINE_p, null) then return;
23     CAS(MINE_p.next, null, MINE_p);
24     if MINE_p.next ≠ MINE_p then
25         MINE_p.next.locked := FALSE;
26     CURR_p := 1 − CURR_p;
```

model, each shared variable is statically allocated to a memory module that is local to exactly one process and remote to all others. Space complexity is simply the number of memory words used per process.

3 An RMR-Optimal RME Algorithm with Dynamic Joining for CC and DSM Models

In this section, we present an RME algorithm for system-wide failures that satisfies the ME and SF correctness properties, and has $O(1)$ RMR complexity in both CC and DSM models. Later, we describe a separate transformation to add the BCSR property to any RME lock.

3.1 Background: MCS Algorithm with Wait-Free Exit

The MCS algorithm is a queue-based ME algorithm that has optimal $O(1)$ RMR complexity in both the CC and DSM models. It maintains a (single) queue of all outstanding requests for critical section; requests are satisfied in the order in which they are inserted into the queue. Pseudocode for the algorithm is given in algorithm 2, and has been modified from the original version [38] to also satisfy the BE property as described in [17].

A request is represented using a `QNode`, which consists of two fields: (i) a boolean variable, *locked*, to indicate whether the node is currently locked and thus its owner does not have permission to enter its critical section, and (ii) a reference to `QNode`, *next*, to store the address of the successor node.

To enter its critical section, a process first appends its node to the queue (Line 17). If the queue was not empty (implying that it has a predecessor), it tries to create a forward link from its predecessor's node to its own node (Line 19). If successful, it waits for its node to be unlocked by its predecessor (Line 20). If it either did not have a predecessor (Line 18) or failed to create the link (Line 19), then it implies that the process holds the lock; in this case the process simply returns.

When leaving its critical section, a process first attempts to remove its node from the queue (Line 22). The attempt will fail if another process has already appended its node to the queue. In that case, the process notifies its successor that the critical section is now empty by either writing a special value to the next field of its node (if the link has not been created yet) (Line 23) or unlocking its successor's node (if the link has already been created) (Line 25).

In the original MCS algorithm, a process can reuse the same node immediately after completing its exit section for its next CS request. However, in the wait-free exit version, immediate reuse may create a deadlock. Using a pool of two nodes and alternating between them [17] solves this problem (Line 26). Intuitively, once a process is "enabled" to enter its critical section, then it can be inferred indirectly that the node it used for its previous request has served its purpose and no process would access its fields anymore; thus, it can be reclaimed for its next request. Similar techniques have been used to achieve the wait-free exit property in other queue-based locks [12, 43, 37, 26], but our particular RME algorithm is derived from the queue-based ME lock described in [17].

3.2 The Main Idea

We modify the augmented MCS algorithm described in Section 3.1 to obtain an RMR-optimal RME algorithm under the system-wide failure model that satisfies the ME and SF but not the BCSR property. Assume, for now, that processes do not reuse queue nodes. The key idea is that, if a process crashes while executing its passage, it terminates or aborts its interrupted attempt to use the lock (for executing its critical section), and initiates a new attempt to use the lock using a fresh node. A process aborts its attempt by basically executing steps of the exit section and unlocking its successor node, if any. Note that a process aborts its attempt even if it crashed in its critical section. Intuitively, this "clears" the queue of any "old" nodes, which is then "repopulated" using "new" nodes.

Note that, in the (augmented) MCS algorithm used for solving the ME problem, queue nodes are unlocked in a serial manner in the same order in which they were appended to the queue. However, in the RME variant, queue nodes associated with aborted attempts may be unlocked out of order and even concurrently. This out-of-order unlocking of queue nodes associated with aborted attempts does not violate the ME property because *none* of these queue nodes, which were appended to the queue before the failure, can be used by its owner now to execute its critical section.

However, the above approach interferes with the node reuse mechanism used in the augmented MCS algorithm. A slow process p may erroneously unlock a reused node owned by a fast process q because q's node was the successor of p's node in an earlier epoch. To address this problem, we replace the *locked* field in a queue node with a field that stores the address of the predecessor node. A process unlocks its successor node by replacing the contents of this field in its successor's node with a **null** value using a `CAS` instruction, which will succeed only if the (successor) node has not been reused.

3.3 A Formal Description

Pseudocode of the RMR-optimal RME algorithm for system-wide failures is presented in algorithm 3. To avoid repetition, we only describe the differences between algorithm 2 (augmented MCS) and algorithm 3.

As explained earlier, the *locked* field in `QNode` (Line 2) has been replaced with the *pred* field (Line 28); a process stores the address of its predecessor node in *pred* (Line 55). The steps of the `Exit` section in algorithm 2 have been abstracted into a `Cleanup` method in algorithm 3, which is then invoked from the `Exit`, `Recover` and `Withdraw` sections. A process unlocks its successor node by clearing the *pred* field of the (successor) node using a `CAS` instruction (Line 50). At the beginning of the `Cleanup` method, a process also deletes the link from its predecessor node to its own node (Lines 45–46). This step is required to prevent too many "older" predecessors from performing a `CAS` instruction on the *pred* field of its node. Although these `CAS` instructions will fail, they may still cause the process to incur an RMR while spinning on *pred* field in the CC model.

Algorithm 3 An RMR-optimal RME algorithm for tolerating system-wide failures that satisfies the ME and SF properties.

```
27 struct QNode {
28     pred: reference to QNode, initially null;
29     next: reference to QNode, initially null;
30 };
31 global shared variables
32     TAIL: reference to QNode, initially null;
33 per-process shared/persistent variables
34     POOL_p: array [0, 1] of QNode, all elements
               initially {null, null};
35     MINE_p: reference to QNode, initially &POOL[0];
36     CURR_p: integer variable ∈ {0, 1}, initially 0;
37 private variables
38     pred: reference to QNode;
39 Procedure Withdraw()
40     Cleanup();
41 Procedure Recover()
42     Cleanup();
43     CURR_p := 1 − CURR_p;
44 Procedure Cleanup()
45     if MINE_p.pred ≠ null then
46         CAS(MINE_p.pred.next, MINE_p, null);
47     CAS(TAIL, MINE_p, null);
48     CAS(MINE_p.next, null, MINE_p);
49     if MINE_p.next ≠ MINE_p then
50         CAS(MINE_p.next.pred, MINE_p, null);
51 Procedure Enter()
52     MINE_p := &POOL_p[CURR_p];
53     MINE_p.pred := null;
54     MINE_p.next := null;
55     pred := FAS(TAIL, MINE_p);
56     MINE_p.pred := pred;
57     if MINE_p.pred = null then return;
58     if CAS(MINE_p.pred.next, null, MINE_p) then
59         await MINE_p.pred = null;
60 Procedure Exit()
61     Cleanup();
```

Recall that, in the ME algorithm described in algorithm 2, a process alternates between two nodes. The same idea works in the recoverable version, except that we perform the node switch at the end of the `Recover` section.

We refer to the algorithm described in algorithm 3 as MCS-SW (SW stands for system-wide). The doorway of MCS-SW consists of Lines 52–55. We have:

▶ **Theorem 1.** *MCS-SW satisfies the ME, SF, BE, BR, BW and FCFS properties of the RME problem. It has $O(1)$ RMR complexity in both CC and DSM models, as well as $O(1)$ space complexity per process. Finally, it uses bounded variables and supports dynamic joining.*

4 Adding the BCSR Property

To our knowledge, two RMR-preserving transformations have been proposed to add the BCSR property to an RME lock that only satisfies the ME and SF properties. The first transformation, given by Golab and Ramaraju [23, 24], assumes the individual failure model. The second transformation, given by Golab and Hendler [22], assumes the system-wide failure model. Neither transformation can be applied to the RME lock described in Section 3.

The first transformation works under the assumption that a process cannot detect the failure of another process. This implies that, if a process p has entered its CS during a passage, then no other process can gain entry into its CS even if p fails during its CS until p has started its exit section, possibly in a future passage. While this assumption holds for any RME lock designed for independent failure model without explicit failure detection, it may not hold for an RME lock designed for the system-wide failure model (with or without explicit failure detection). Specifically, the RME lock in Section 3 exploits the property that one's own failure also implies the failure of every other process in the system. So, a process p can gain entry into its CS after another process q fails while executing its CS without requiring q to start its exit section. The second transformation, on the other hand, requires an explicit epoch-based failure detector.

Note that the transformation for the CC model is relatively straightforward and involves spinning on a global variable. However, the one for the DSM model is non-trivial because spinning on a global variable may incur unbounded RMRs in the worst case. In this section, we present a new transformation that can be applied to any RME lock for system-wide failures without using an explicit failure detector, and incurs only $O(1)$ RMRs in both CC and DSM models. Additionally, it preserves all properties of MCS-SW except for fairness which now weakens to 0-FCFS, a direct consequence of CSR.

4.1 The Main Idea

The RMR-optimal RME algorithm described in Section 3 does not satisfy the BCSR property. This is because, when processes append new nodes to the queue after aborting their interrupted attempts, these new nodes may be appended to the queue in a different order. As a result, a process that crashes while executing its critical section may need to wait for one or more processes to complete their critical sections before it can reenter its own because nodes of other processes *now precede* its own node in the queue.

To add the BCSR property, we maintain a global variable that stores the identifier of the process currently in CS. A process writes its identifier to the variable when it gains entry into the CS and resets it upon leaving the CS. If a process crashes during its CS, then, upon starting a new c-passage, it can return from `Recover` and `Enter` methods of the target lock immediately if the variable contains its own identifier. We refer to this path to enter the CS as *legacy* path. Otherwise, it first acquires the base lock and then possibly waits for the process with legacy admission into the CS to leave, if applicable. We refer to this path to enter the CS as *regular* path.

Jayanti, Jayanti and Joshi define a *capturable* object that can be used to synchronize access to the CS between processes taking the two paths, while incurring only $O(1)$ RMRs in both CC and DSM models [29]. However, their capturable object uses an *unbounded* sequence number as well as a *read-modify-write* instruction. We use a similar, but simpler, approach that avoids the two limitations. Our approach exploits the fact that any legacy admission can only happen at the beginning of an epoch (period between two consecutive system-wide failures), after which all admissions are regular.

The main idea is that a process that takes the regular path, say p, uses its own memory location (basically a boolean variable) to spin and stores the address of its spin location in another global variable before spinning (provided that the CS is already occupied). The process leaving the CS is responsible for signalling p by writing to the location provided by p.

4.2 A Formal Description

Pseudocode of the transformation to attain the BCSR property is presented in algorithm 4. We refer to the transformation as CSR-SW.

The algorithm uses the following shared variables: (i) base RME lock, BLOCK, that satisfies the ME and SF properties (*e.g.*, the lock in algorithm 3), (ii) integer, CSOWNER, to store the identifier of the process that currently owns the CS, (iii) location, CSWAIT, to store the address of the boolean variable on which a process will spin, (iv) per process boolean variable, LOCKED, for spinning until signalled, and (v) per process variable, *skipRE*, to track whether a process skipped acquiring the base RME lock in this current passage.

In the `Recover` method, a process first checks if it already owns the CS (Line 71). If yes, it makes a note of it (Line 72) and completes the super-passage of the base lock by invoking its `Withdraw` method (Line 73) (legacy path). Otherwise, it starts a new passage of the base lock by invoking its `Recover` method (Line 76) (regular path).

Algorithm 4 An RMR-optimal transformation to achieve the BCSR property.

```
62 global shared variables
63     BLock: instance of recoverable (base) lock;
64     CSOwner: process identifier, initially $\perp$;
65     CSWait: reference to a boolean variable,
               initially null;
66 per-process shared/persistent variables
67     $\text{Locked}_p$: boolean variable, initially null;
68 private variables
69     skipRE: boolean variable;
70 Procedure Recover()
71     if CSOwner $= p$ then
72         skipRE := TRUE;
73         BLock.Withdraw();
74     else
75         skipRE := FALSE;
76         BLock.Recover();
77 Procedure Enter()
78     if CSOwner $= p$ then return;
79     BLock.Enter();
80     $\text{Locked}_p$ := TRUE;
81     CSWait := &$\text{Locked}_p$;
82     if CSOwner $= \perp$ then $\text{Locked}_p$ := FALSE;
83     await not($\text{Locked}_p$);
84     CSOwner := $p$;
85 Procedure Exit()
86     CSOwner := $\perp$;
87     if CSWait $\neq$ null then *CSWait := FALSE;
88     if $\neg$skipRE then BLock.Exit();
89 Procedure Withdraw()
90     if CSOwner $= p$ then
91         CSOwner := $\perp$;
92         if CSWait $\neq$ null then *CSWait := FALSE;
93     BLock.Withdraw();
```

In the `Enter` method, if a process already owns the CS, it returns immediately (Line 78) (legacy path). Otherwise, it acquires the base lock (Line 79). It next initializes its spin location (Line 80), announces the address of its spin location (Line 81) and busy-waits if the CS is already occupied (Lines 82 and 83). Finally, it claims the ownership of the CS (Line 84) (regular path).

In the `Exit` method, a process releases the ownership of the CS (Line 86) and signals the waiting process if any (Line 87). If it took the regular path, it completes the super-passage of the base lock by invoking its `Exit` method (Line 88).

In the `Withdraw` method, if a process currently owns the CS, it releases its ownership (Line 91) and signals the waiting process if any (Line 92). It then invokes the `Withdraw` method of the base lock (Line 93).

The doorway of the target lock consists of Line 78 along with the doorway of the base lock if acquired. We have:

▶ **Theorem 2.** *CSR-SW preserves the ME, SF, BR, BE, BW and k-FCFS for $k \geq 0$ properties of the base lock, and adds the BCSR property to the target lock. It uses bounded variables and supports dynamic joining. Finally, it preserves the RMR and space complexities of the base lock.*

Intuitively, the ME property holds for the following reasons. First, regular admissions to the CS are serialized using the base lock. Second, there is at most one legacy admission to the CS during an epoch. Third, a regular admission can only occur if the process with legacy admission has vacated the CS.

Intuitively, the SF property holds for the following reasons. First, the base lock satisfies the SF property. Second, only a process that takes the regular path, after acquiring the base lock, needs to busy-wait for the CS to become empty due to legacy admission. If such a process, say p, finds CSOwner to have a non-bottom value implying that another process, say q, is currently in the CS, then p would have already written the address of its spin location to CSWait. Clearly, upon leaving CS, q is guaranteed to find the address of p's spin location and signal p to quit its busy-wait loop.

▶ **Corollary 3.** *The target lock obtained by applying CSR-SW to MCS-SW satisfies the ME, SF, BCSR, BE, BR, BW and 0-FCFS properties. It uses bounded variables and supports dynamic joining. Finally, it has $O(1)$ RMR complexity in both CC and DSM models, and uses $O(1)$ space per process.*

5 A Fully Dynamic RMR-Optimal RME Algorithm for the CC Model

The RME algorithms in [29] as well as those presented in Sections 3 and 4 support dynamic joining. While a process can leave the system if it has no super-passage in progress, it cannot reclaim its memory since other processes may still dereference one of its locations (*e.g.*, the *next* field of its queue node). A separate garbage collection mechanism is required to identify nodes whose memory can be safely reclaimed. We describe an RMR-optimal RME algorithm for the CC model that allows a process to not only join the system at any time but also leave at any time and *safely deallocate its memory when leaving.* The algorithm, however, uses an unbounded sequence number.

Our RME algorithm is derived from Lee's ME lock [37, 29], which is specifically designed for the CC model, and uses the idea described in Section 3. Like the MCS lock, Lee's lock is also queue-based and requests are satisfied in the order in which they are appended to the queue. However, the queue is implicit and a queue node does not store *next* pointers. A process instead spins on a memory location in its predecessor's node until it is signalled by the predecessor. Further, unlike in the MCS lock, a process does not attempt to remove its node from the queue.

Algorithm 5 describes the pseudocode of the RME algorithm with support for dynamic leaving, based on Lee's ME lock. We refer to the algorithm as LeeDL-SW. Note that the original Lee's lock consists of Lines 118–121 for acquiring the lock and Line 112 for releasing the lock. To acquire the lock, a process switches its queue node (Line 118), initializes it (Line 119), appends it to the queue (Line 120) and waits until signalled by its predecessor (Line 121). To release the lock, a process simply signals its successor (Line 112).

We now describe adding recoverability, as well as the *optional* critical section re-entry and dynamic leaving properties to Lee's lock, while preserving its RMR optimality. For convenience, the code lines used to achieve critical section re-entry and dynamic leaving properties are tagged with one and two "◀" symbols, respectively, at the end of each line.

Adding recoverability. We define a `Cleanup` method in which a process attempts to remove its node from the queue (Lines 110 and 111) and then signal its successor in case it has one (Line 112). The `Cleanup` method is invoked from the `Recover`, `Exit` and `Withdraw` methods.

Adding critical section re-entry. We use a shared variable to keep track of the process currently executing its CS (CSOwner). Once a process has been signalled by its predecessor (*i.e.*, it has acquired Lee's lock), the process has to wait until the CS has become empty (Line 122) and then claims the ownership of the CS (Line 123). A process releases the ownership of the CS as part of the `Cleanup` method (Line 109). If a process fails during its CS, it can immediately return from the `Recover` and `Enter` methods if it already owns the CS (Lines 114 and 117).

Adding dynamic leaving. A process can leave the system only if it has no super-passage in progress. Moreover, before leaving the system, the process must invoke the `SafeToReclaim` method. It then either waits for the method to complete or for a system-wide failure to occur. Once either of these two events has occurred, it can safely reclaim all its memory.

We use a sequence number to keep track of the *rough* number of times the CS has been occupied (CSBusy). A process increments this sequence number after obtaining the ownership of the CS (Line 124). Note that the count need not be exact due to failures. The purpose of the `SafeToReclaim` method is to ensure that the caller does not have a successor

Algorithm 5 An RMR-optimal RME lock for the CC model that supports dynamic joining *as well as dynamic leaving.* Code in red color is only required for the CSR property.

```
94  struct QNode{
95      locked: boolean variable, initially FALSE;
96  };
97  global shared variables
98      TAIL: reference to QNode, initially null;
99      CSOWNER: process identifier, initially ⊥;                ◀
100     CSBUSY: integer variable, initially 0;                   ◀◀
101 per-process shared/persistent variables
102     POOL_p: array [0,1] of QNode, all elements
            initially {FALSE};
103     FACE_p: integer variable, initially 0;
104     LEAVING_p: boolean variable, initially
            FALSE ;                                              ◀◀
105 private variables
106     pred, tail: reference to QNode;
107     busy: integer variable;
108 Procedure Cleanup()
109     if CSOWNER = p then CSOWNER := ⊥;                        ◀
110     if TAIL = &POOL_p[FACE_p] then
111         CAS(TAIL, &POOL_p[FACE_p], null)
112     POOL_p[FACE_p].locked := FALSE;
113 Procedure Recover()
114     if CSOWNER = p then return;                              ◀
115     Cleanup();
116 Procedure Enter()
117     if CSOWNER = p then return;                              ◀
118     FACE_p := 1 − FACE_p;
119     POOL_p[FACE_p].locked := TRUE;
120     pred := FAS(TAIL, &POOL_p[FACE_p]);
121     if pred ≠ null then await not(pred.locked) ;
122     await CSOWNER = ⊥;                                       ◀
123     CSOWNER := p;                                            ◀
124     CSBUSY := CSBUSY + 1;                                    ◀◀
125 Procedure Exit()
126     Cleanup();
127 Procedure Withdraw()
128     Cleanup();
129 Procedure SafeToReclaim()                                    ◀◀
130     if not(LEAVING) then                                     ◀◀
131         LEAVING := TRUE;                                     ◀◀
132         tail := TAIL;                                        ◀◀
133         if tail = null then return;                          ◀◀
134         busy := CSBUSY;                                      ◀◀
135         await (TAIL ≠ tail) or
                (CSBUSY > busy);                                 ◀◀
136         if TAIL = null then return;                          ◀◀
137         await (CSBUSY > busy);                               ◀◀
```

that is still executing its `Enter` method and thus may *dereference* one of its queue nodes. Clearly, the condition holds if the queue is empty (*i.e.*, the tail pointer is **null**). Note that, when a process invokes the `SafeToReclaim` method, the tail pointer cannot point to any of its two queue nodes since the process would have completed a failure-free instance of the `Cleanup` method with respect to each of its nodes. Thus, if a process finds that the queue is not empty when leaving, it can infer that one or more nodes have been appended to the queue after its most recent append operation. The issue is that some of these nodes may have been appended to the queue before the last failure, and thus will be eventually abandoned by their owners.

In the `SafeToReclaim` method, the process first reads the tail pointer and returns if the queue is empty (Lines 132 and 133). Otherwise, it reads the sequence number mentioned earlier (Line 134). It then waits until either the tail pointer or the sequence number has advanced (Line 135). It next re-reads the tail pointer and returns if the queue is now empty (Line 136). If not, it implies that a *new* node has been added to the queue since the last system-wide failure. Thus, either the sequence number will eventually advance (Line 137) or the system will crash (Lines 130 and 131). The process can safely reclaim its memory in either case.

It is possible that a process may incur multiple RMRs while spinning on the tail pointer. This may happen when the same node is appended to the queue again or when a failed `CAS` instruction is executed on the tail pointer. In the first case, we can show that the sequence number has also advanced. In the second case, we can show that failed `CAS` instructions can generate at most three RMRs for a spinning process. This is because a process signals its successor only after trying to remove its node from the queue.

▶ **Theorem 4.** *LeeDL-SW satisfies the ME, SF, BCSR, BE, BR, BW and 0-FCFS properties. It supports dynamic joining as well as dynamic leaving. Finally, it has $O(1)$ RMR complexity in the CC model, and uses $O(1)$ space per process.*

We implemented the RME algorithm from this section and compared it against the CC-specific version of Jayanti, Jayanti and Joshi's (JJJ) RME algorithm (Section 3 of [29]). With BCSR property, our algorithm provided 50-75% better throughput than JJJ at low levels of parallelism, and around 15% better throughput at higher levels of parallelism. Without BCSR property, our algorithm was roughly 2-3× faster than JJJ. Details of our experiments are provided in Appendix B.

6 On Achieving Dynamic Joining and Leaving, Wait-Free Withdraw, Adaptive Space, and Bounded RMR Complexity in the DSM Model

In this section, we present an impossibility result for the DSM model (without explicit failure detection), and for a particular class of RME algorithms characterized according to the following combination of properties:

1. *Bounded RMR complexity per passage.* A process can only busy-wait by spinning on a variable that is local to it in the DSM model, and can only access a bounded number of remote memory locations per passage.
2. *Adaptive space complexity with external memory allocation.* The algorithm's state comprises $O(1)$ memory words shared by all processes, called the *global state*, and $O(1)$ additional memory words per process called *process-local state*. The process-local state for process p is allocated by process p externally (*i.e.*, outside of the RME algorithm) and is local to process p in the DSM model.
3. *Memory safety.* Externally allocated memory can be accessed by the RME algorithm only after it has been allocated and before it has been freed (*i.e.*, *use-after-free* is not permitted).
4. *Dynamic joining and leaving.* Any process can become active in an execution history after allocating its process-local state. Any active process can leave after a failure-free passage, meaning that it can free its process-local state in a bounded number of its own steps and then halt in the NCS.
5. *Wait-free withdraw:* Any process can complete any invocation of the `Withdraw` method in a bounded number of its own steps.

We now present the main result of this section:

▶ **Theorem 5.** *No RME algorithm can satisfy properties 1–5 simultaneously.*

Proof. Suppose for contradiction that some RME algorithm $\mathcal{A}$ does satisfy each of the properties 1–5. Since the algorithm's global state comprises only $O(1)$ memory words by property 2, it is easy to find two processes p and q such that all the global state is remote to q. We first construct a finite failure-free execution history H_0 involving these two processes and having the structure $H_0 = G_0^p \circ G_0^q$ where p begins a c-passage and enters the CS in G_0^p, and q begins a c-passage in G_0^q and enters a busy-wait loop since it cannot enter the CS concurrently with p. It follows that q must be spinning on its local memory at the end of G_0^q since $\mathcal{A}$ has bounded RMR complexity (property 1) and since every variable in the global state is remote to q by our choice of q. Next, observe that since H_0 exists, the execution $H_1 = G_0^p \circ f \circ G_0^q$ is also possible where f is a crash step (system-wide failure). This is because q cannot distinguish between the prefixes G_0^p and $G_0^p \circ f$ without an explicit failure detector. We extend H_1 to the history $H_2 = G_0^p \circ f \circ G_0^q \circ G_1^p$ where p calls the `Withdraw` method in G_1^p and q takes no additional steps. It follows that p must eventually access q's local memory in some step s of G_1^p as otherwise q is stuck in a busy-wait loop forever if we extend H_2 by interleaving steps of p and q in a fair order. Next, we transform G_1^p to G_2^p by truncating this suffix at the step immediately before p accesses q's memory in step s, and observe that the history $H_3 = G_0^p \circ G_0^q \circ f \circ G_2^p$ is also possible, once again because there is no explicit failure detector. Process q cannot distinguish between the prefixes G_0^p and $G_0^p \circ f$, as explained earlier, and p cannot distinguish between $G_0^p \circ f \circ G_0^q$ and $G_0^p \circ G_0^q \circ f$ since only q takes steps in G_0^q. Finally, we transform H_3 to $H_4 = G_0^p \circ G_0^q \circ f \circ G_2^p \circ G_1^q$ where q executes the `Withdraw` method in G_1^q, frees its local memory, and leaves the execution (property 4) while p takes no additional steps. Such a history is possible since the `Withdraw` method is bounded (property 5). Extending H_4 by p's step s, which it is poised to execute at the end of fragment G_2^p, leads to a contradiction of memory safety (property 3) as p accesses q's local memory after q has already freed it. ◀

7 Related Work

Golab and Ramaraju's formulation of the RME problem [23, 24] is a theoretical take on the practical problem of making mutual exclusion locks robust against crash failures, which can be traced back to several earlier works [7, 8, 39, 46]. A pervasive pattern in this area of shared memory research is that fault-tolerant locks rely in various ways on support from the execution environment, for example where a centralized recovery process is invoked after a crash to clean up the internal state of the lock, where an explicit failure detector allows a waiting process to usurp a critical section held by a crashed process, or where shared variables are reset automatically to specific values during a failure. The RME problem formulation avoids such specific assumptions, and instead considers that a crashed process recovers and resumes execution eventually, unless it failed in the NCS.

Both classic ME algorithms and more recent RME algorithms are evaluated primarily with respect to remote memory reference (RMR) complexity in the cache-coherent (CC) and distributed shared memory (DSM) multiprocessor architectures, space complexity, as well as the set of correctness properties (*e.g.*, starvation freedom and fairness) achieved. In terms of worst-case RMR bounds for the individual process failure model, Golab and Ramaraju [23] established that RME can be solved for n processes using reads and writes with $O(\log n)$ RMRs per passage, which matches the lower bound of Attiya, *et al.* [5]. The latter bound applies to both ME and RME, and can be generalized for ME to comparison primitives using the RMR-efficient construction of Golab, *et al.* [20]. For algorithms that use other read-modify-write primitives, such as Fetch-And-Store or Fetch-And-Increment, a sub-logarithmic (*i.e.*, $O(\log n/\log\log n)$) upper bound was established jointly by Golab and Hendler [21] as well as Jayanti, Jayanti, and Joshi [27], and proven to be tight by Chan and Woelfel [10]. Katzan and Morrison [34] also proposed an $O(\log_w n)$ RMRs solution using w-bit Fetch-And-Add, which matches [21, 27] when $w \in \Theta(\log n)$ and reduces to $O(1)$ in the extreme case when $w \in \Theta(n)$; it was recently shown to be tight by Chan *et al.* in [9].

Most RME algorithms that tolerate individual process failures work correctly and achieve the same RMR complexity in the system-wide failure model. Golab and Hendler [22] solved the problem directly for system-wide failures using a failure detector and commonly supported primitives, and showed that the RMR complexity can be reduced to $O(1)$ using $O(\log n)$-bit Fetch-And-Store or Fetch-And-Increment primitives. Jayanti, Jayanti, and Joshi [33, 29] were the first to present RMR-optimal RME algorithms for system-wide failures that do not use an explicit failure detector. Their approach is presented as a way to transform a traditional

ME lock into an RME lock (under certain conditions) by maintaining three copies of the ME lock and using an intricate synchronization mechanism to guide requesting processes to an uncorrupted copy while concurrently resetting a possibly corrupted copy.

Some (ME) locks support the *abort* feature, which allows a process to abandon – within a bounded number of its own steps – its attempt to acquire the lock [19, 3, 30, 31]. This is useful in situations when a process may only wish to wait for a fixed amount of time to acquire the lock, and, if unable to do so, would prefer to cancel the attempt and perform some other task before reattempting to acquire the lock. The notion of abortability has been extended to RME locks as well in [32, 34]. We show in Appendix C that abortable RME and withdrawable RME are *equivalent* problems under *individual* failures and certain assumptions by giving RMR-preserving transformations that convert one type of lock into the other. Surprisingly, the two problems may not be same under *system-wide* failures. In particular, as we show in this work, it is possible to design an RMR-optimal RME lock that supports a wait-free withdraw section *assuming system-wide failures.* Note that any abortable RME lock under system-wide failures will also implement an abortable ME lock in a failure-free environment. However, the best known abortable ME lock has $\Omega(\log n/\log\log\ n)$ RMR complexity in the worst-case [3]. Intuitively, the reason for this gap is that a process may receive the abort signal *at any time* during its passage, whereas a process can execute the withdraw section only at the *beginning* of its passage *after a failure* to simplify recovery.

A comprehensive discussion of the RME problem and its solutions can be found in [14].

8 Conclusion and Future Work

In this work, we have presented a *modular* way to design an RMR-optimal RME lock for both CC and DSM models under system-wide failures without relying on an explicit failure detector. Our approach is flexible in the sense that an application can pick and choose the properties the RME lock should satisfy depending on its needs (*e.g.*, CSR, FRF[2], *etc.*). Further, we have proposed the notion of *withdrawing* from a lock acquisition as opposed to resetting the lock *after a failure.* The latter is more complex and requires greater synchronization among processes. Moreover, withdrawable RME locks make it easier in some scenarios to write fault-tolerant application programs for persistent memory.

In the future, we plan to conduct more comprehensive experiments to compare the performance of different RME lock alternatives to better understand how their features (*e.g.*, correctness properties, reliance on a failure detector) impact performance. We also plan to design a fully dynamic RMR-optimal RME algorithm for the DSM model, and investigate whether the blocking synchronization used in our dynamic algorithm for the CC model is inherently necessary.

References

1 Yehuda Afek, David S. Greenberg, Michael Merritt, and Gadi Taubenfeld. Computing with faulty shared objects. *Journal of the ACM (JACM)*, 42(6):1231–1274, 1995.

2 Marcos K. Aguilera and Svend Frølund. Strict linearizability and the power of aborting. Technical Report HPL-2003-241, Hewlett-Packard Labs, 2003.

[2] FRF refers to *failure-robust fairness*, a type of fairness property defined in [22] for the system-wide failure model, which constraints the number of times a given process can be overtaken by other processes as regard to their super-passages. It is possible to provide a general transformation that adds the FRF property to any RME lock similar to that for the BCSR property [13].

3 Adam Alon and Adam Morrison. Deterministic abortable mutual exclusion with sublogarithmic adaptive RMR complexity. In *Proc. of the 37th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 27–36, 2018.

4 James H. Anderson, Yong-Jik Kim, and Ted Herman. Shared-memory mutual exclusion: major research trends since 1986. *Distributed Computing (DC)*, 16(2-3):75–110, 2003.

5 Hagit Attiya, Danny Hendler, and Philipp Woelfel. Tight RMR lower bounds for mutual exclusion and other problems. In *Proc. of the 40th ACM Symposium on Theory of Computing (STOC)*, pages 217–226, 2008.

6 Ryan Berryhill, Wojciech Golab, and Mahesh Tripunitara. Robust shared objects for non-volatile main memory. In *Proc. of the 19th International Conference on Principles of Distributed Systems (OPODIS)*, pages 20:1–20:17, 2016.

7 Philip Bohannon, Daniel Lieuwen, and Avi Silberschatz. Recovering scalable spin locks. In *Proc. of the 8th IEEE Symposium on Parallel and Distributed Processing (SPDP)*, pages 314–322, 1996.

8 Philip Bohannon, Daniel Lieuwen, Avi Silberschatz, S. Sudarshan, and Jacques Gava. Recoverable user-level mutual exclusion. In *Proc. of the 7th IEEE Symposium on Parallel and Distributed Processing (SPDP)*, pages 293–301, 1995.

9 David Yu Cheng Chan, George Giakkoupis, and Philipp Woelfel. Word-size rmr trade-offs for recoverable mutual exclusion. In *Proc. of the 43rd ACM Symposium on Principles of Distributed Computing (PODC)*, 2023.

10 David Yu Cheng Chan and Philipp Woelfel. A tight lower bound for the RMR complexity of recoverable mutual exclusion. In *Proc. of the 40th ACM Symposium on Principles of Distributed Computing (PODC)*, 2021.

11 Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

12 Travis S. Craig. Building FIFO and priority-queuing spin locks from atomic swap. Technical Report 93-02-02, Department of Computer Science, University of Washington, 1993.

13 Sahil Dhoked. *Synchronization and Fault Tolerance Techniques in Concurrent Shared Memory Systems*. PhD thesis, The University of Texas at Dallas, 2022.

14 Sahil Dhoked, Wojciech Golab, and Neeraj Mittal. *Recoverable Mutual Exclusion*. Springer Nature, 2023.

15 Sahil Dhoked and Neeraj Mittal. An adaptive approach to recoverable mutual exclusion. In *Proc. of the 39th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 1–10, New York, NY, USA, 2020. doi:10.1145/3382734.3405739.

16 Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM (CACM)*, 8(9):569, 1965.

17 Rotem Dvir and Gadi Taubenfeld. Mutual exclusion algorithms with constant RMR complexity and wait-free exit code. In James Aspnes, Alysson Bessani, Pascal Felber, and João Leitão, editors, *Proc. of the International Conference on Principles of Distributed Systems (OPODIS)*, volume 95, pages 17:1–17:16, Dagstuhl, Germany, October 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

18 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32:374–382, 1985.

19 George Giakkoupis and Philipp Woelfel. Randomized abortable mutual exclusion with constant amortized RMR complexity on the CC model. In *Proc. of the 36th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 221–229, 2017.

20 Wojciech Golab, Vassos Hadzilacos, Danny Hendler, and Philipp Woelfel. RMR-efficient implementations of comparison primitives using read and write operations. *Distributed Computing (DC)*, 25(2):109–162, 2012.

21 Wojciech Golab and Danny Hendler. Recoverable mutual exclusion in sub-logarithmic time. In *Proc. of the 36th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 211–220, 2017.

22 Wojciech Golab and Danny Hendler. Recoverable mutual exclusion under system-wide failures. In *Proc. of the 37th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 17–26, 2018.

23 Wojciech Golab and Aditya Ramaraju. Recoverable mutual exclusion. In *Proc. of the 35th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 65–74, 2016.

24 Wojciech Golab and Aditya Ramaraju. Recoverable mutual exclusion. *Distributed Computing (DC)*, 32(6):535–564, 2019.

25 Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint.* Morgan Kaufman, 2012.

26 Prasad Jayanti, Siddhartha Jayanti, and Sucharita Jayanti. Towards an ideal queue lock. In *Proc. of the 21st International Conference on Distributed Computing and Networking (ICDCN)*, January 2020.

27 Prasad Jayanti, Siddhartha Jayanti, and Anup Joshi. A recoverable mutex algorithm with sub-logarithmic RMR on both CC and DSM. In *Proc. of the 38th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 177–186, 2019.

28 Prasad Jayanti, Siddhartha Jayanti, and Anup Joshi. Constant rmr recoverable mutex under system-wide crashes, 2023. `arXiv:2302.00748`.

29 Prasad Jayanti, Siddhartha Jayanti, and Anup Joshi. Constant RMR system-wide failure resilient durable locks with dynamic joining. In *Proc. of the 35th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 227–237, 2023.

30 Prasad Jayanti and Siddhartha V. Jayanti. Constant amortized RMR complexity deterministic abortable mutual exclusion algorithm for CC and DSM models. In *Proc. of the 38th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 167–176, 2019.

31 Prasad Jayanti and Anup Joshi. Recoverable mutual exclusion with abortability. In Mohamed Faouzi Atig and Alexander A. Schwarzmann, editors, *Proc. of the International Conference on Networked Systems (NetSys)*, pages 217–232, 2019.

32 Prasad Jayanti and Anup Joshi. Recoverable mutual exclusion with abortability. In *Proc. of 7th International Conference on Networked Systems (NETYS)*, pages 217–232, 2019.

33 Anup Joshi. *Recoverable Mutual Exclusion Algorithms for Crash-Restart Shared-Memory Systems.* PhD thesis, Dartmouth College, May 2020.

34 Daniel Katzan and Adam Morrison. Recoverable, abortable, and adaptive mutual exclusion with sublogarithmic RMR complexity. In *Proc. of the 24th International Conference on Principles of Distributed Systems (OPODIS)*, pages 15:1–15:16, 2021.

35 Leslie Lamport. The mutual exclusion problem: part I – a theory of interprocess communication. *Journal of the ACM (JACM)*, 33(2):313–326, 1986.

36 Leslie Lamport. The mutual exclusion problem: part II – statement and solutions. *Journal of the ACM (JACM)*, 33(2):327–348, 1986.

37 Hyonho Lee. Local-spin mutual exclusion algorithms on the DSM model using fetch-&-store objects. Master's thesis, University of Toronto, 2003.

38 John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

39 Maged M. Michael and Yong-Jik Kim. Fault tolerant mutual exclusion locks for shared memory systems, 2009. US Patent 7,493,618.

40 Thomas Moscibroda and Rotem Oshman. Resilience of mutual exclusion algorithms to transient memory faults. In *Proc. of the 30th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 69–78, 2011.

41 Michel Raynal. *Algorithms for Mutual Exclusion.* MIT, 1986.

42 Michel Raynal and Gadi Taubenfeld. Mutual exclusion in fully anonymous shared memory systems. *Information Processing Letters*, 158:105938, 2020.

43 I. Rhee. Optimizing a FIFO, scalable spin lock using consistent memory. In *Proc. of the 17th IEEE Real-Time Systems Symposium (RTSS)*, pages 106–114, December 1996.

44 Andy Rudoff. Re: cascade lake doesn't support clwb? [discussion post]. `https://groups.google.com/g/pmem/c/DRdYIc7ORHc/m/rtoP681rAAAJ`, 2021. Google Groups.

45 Andy Rudoff and the Intel PMDK Team. Persistent memory development kit, 2020. [last accessed 2/11/2021]. URL: `https://pmem.io/pmdk/`.

46 Gadi Taubenfeld. *Synchronization Algorithms and Concurrent Programming*. Prentice Hall, 2006.

47 Gadi Taubenfeld. Coordination without prior agreement. In Elad Michael Schiller and Alexander A. Schwarzmann, editors, *Proc. of the 36th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 325–334, 2017.

A A Recoverable Lock-Based Concurrent Linked List

We begin with a non-recoverable linked list algorithm adapted from Chapter 9 of Herlihy and Shavit's book [25], and shown below in algorithm 6. The list uses two sentinel nodes – head and tail – which ensures that a traversal can always acquire locks on consecutive nodes. The items stored in these sentinel nodes are minimum and maximum values from the domain of values stored in the list, and cannot be added or removed once the list is initialized.

Algorithm 6 A concurrent linked list based on fine-grained locking.

```
struct LLNode {
    item: value stored in this node;
    next: reference to next linked list node;
    M: mutex lock;
};
global shared variables
    head: reference to head sentinel node of
          linked list;
private variables
    pred, curr, newNode: reference to LLNode;
boolean Procedure Add(item)
    pred := head;
    pred.M.Enter();
    curr := pred.next;
    curr.M.Enter();
    while curr.item < item do
        pred.M.Exit();
        pred := curr;
        curr := curr.next;
        curr.M.Enter();
    if curr.item = item then
        curr.M.Exit();
        pred.M.Exit();
        return FALSE;
    newNode := new LLNode;
    newNode.next := curr;
    pred.next := newNode;
    curr.M.Exit();
    pred.M.Exit();
    return TRUE;
boolean Procedure Remove(item)
    pred := head;
    pred.M.Enter();
    curr := pred.next;
    curr.M.Enter();
    while curr.item < item do
        pred.M.Exit();
        pred := curr;
        curr := curr.next;
        curr.M.Enter();
    if curr.item = item then
        pred.next := curr.next;
        curr.M.Exit();
        pred.M.Exit();
        return TRUE;
    curr.M.Exit();
    pred.M.Exit();
    return FALSE;
```

A recoverable (*i.e.*, strictly linearizable [2]) version of the above algorithm is obtained by allocating the node structure and program variables (including *curr* and *pred*) in persistent memory, replacing the node-level ME locks with RME locks, and adding a recovery procedure that cleans up any locks that may have been corrupted by a crash. Program variables that were private before the transformation now need appropriate initial values since they may be accessed by the recovery procedure before they can be explicitly initialized. A value of **null** is appropriate for the linked list since the variables are all pointers to `LLNode`.

Algorithm 7 Recovery protocol based on withdrawable RME locks.

```
185 Procedure Recover()
186     if $pred \neq$ null then $pred.M$.Withdraw();
187     if $curr \neq$ null and $curr \neq pred$ then $curr.M$.Withdraw();
```

Recovery Using Withdrawable RME Locks. We begin by explaining how recovery can be achieved using the withdrawable RME locks introduced in this work. The recovery procedure executed by a process after a failure (or at the start of every execution) is presented in algorithm 7. The idea is to clean up any locks a process may have held at the time of a crash failure by executing the `Withdraw` section, and allow a garbage collector to deal with memory leaks. No further action is required since the fundamental structure of the linked list cannot be corrupted by a system-wide failure. The relevant locks are identified by the *curr* and *pred* variables, which usually point to the last two linked list nodes accessed, and sometimes to a single node (*e.g.*, if a crash occurs immediately after Line 174). The entire recovery protocol is wait-free as long as the locks provide *wait-free* `Withdraw` sections, which means that it cannot possibly introduce deadlock.

Recovery Using RME Locks in the Style of Jayanti, Jayanti, and Joshi. For comparison, we consider an alternative design of the recovery procedure based on Jayanti, Jayanti, and Joshi's RME lock [29], which lacks a `Withdraw` section and uses a slightly different interface in which the `Recover` section directs a process to either return to the NCS or proceed to the CS via its return value. As explained in Section 7, the JJJ algorithm can be used to simulate a `Withdraw` section, but that would go against the intent of their reformulation of the RME problem, which is to resume execution of the CS if that is where the failure occurred. Applying the latter principle to the linked list, we immediately run into difficulties. To begin with, the recovery procedure must consider different cases depending on the return values of the `Recover` section: IN_REM *vs.* IN_CS. With two locks to recover, there are four principal cases to analyze:

1. *pred.M*.`Recover()` returns IN_REM and *curr.M*.`Recover()` returns IN_REM (*e.g.*, crash at line 148): nothing to do
2. *pred.M*.`Recover()` returns IN_REM and *curr.M*.`Recover()` returns IN_CS (*e.g.*, crash at line 153): resume traversal
3. *pred.M*.`Recover()` returns IN_CS and *curr.M*.`Recover()` returns IN_REM (*e.g.*, crash at line 155): resume traversal
4. *pred.M*.`Recover()` returns IN_CS and *curr.M*.`Recover()` returns IN_CS (*e.g.*, crash at line 161): resume traversal

Upon closer inspection of the above four cases, we note additional complications. First, the traversal may need to be resumed either inside the `Add` procedure or inside the `Remove` procedure, and so the algorithm must be augmented with additional state to keep track of which procedure the process was executing; this hurts performance and effectively increases the number of cases during recovery from four to seven as the traversal may need to resume at slightly different points in cases 2–4 (case 1 applies equally well to both `Add` and `Remove`). Moreover, in case 4 alone we must consider separately the subcase when $pred = curr$ and the subcase when $pred \neq curr$, which increases the total number of cases from seven to nine. Without fleshing out the recovery protocol in full detail, we conclude that the solution would be substantially more complex than our solution in algorithm 7.

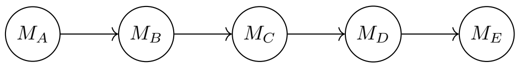


Figure 1 Illustration of deadlock scenario with coarse-grained idempotent actions.

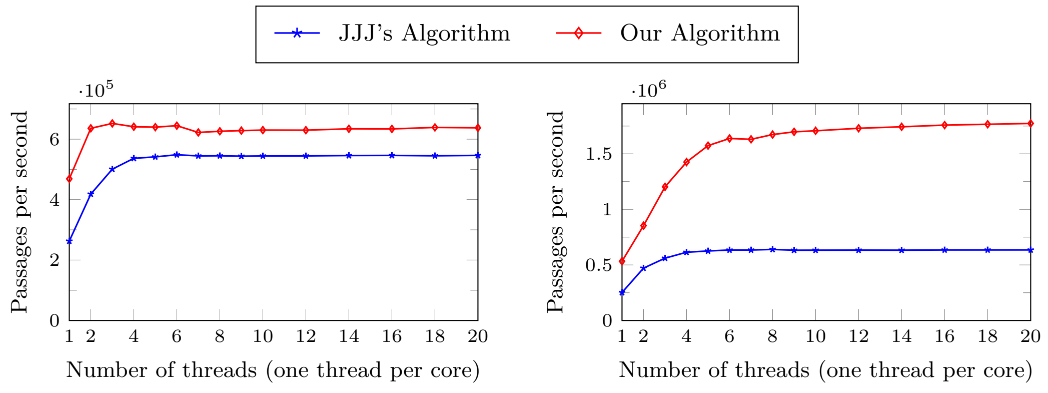
Recovery Using RME Locks in the Style of Golab and Ramaraju. Finally, we consider recovery using plain RME locks, as defined by Golab and Ramaraju [23, 24]. Such locks lack a `Withdraw` section, and are based on the classic concept of idempotent actions: a process recovers by simply repeating its entire passage. A course-grained interpretation of this paradigm has each process repeating the entire linked list operation it was performing at the time of failure, which easily leads to deadlock. In the example structure illustrated below in Figure 1, process p could crash in the CS of lock M_D, at the same time process q could crash in the CS of another lock M_B closer to the head of the list, then p could block while entering M_B during recovery because it is effectively still held by q, and q could subsequently block while entering the lock M_A in the predecessor node, which is still held by p due to hand-over-hand locking.

A more fine-grained interpretation of idempotent actions has each process recovering only the one or two locks it was accessing at the time of failure, similarly to algorithm 7 but with a sequence of calls to `Recover`, `Enter`, and `Exit` in place of a single call to `Withdraw`. However, once again we run into the possibility of deadlock. If a process p tries to recover $pred$ first and then $curr$, as in algorithm 7, then consider what happens if the system-wide failure occurred immediately after p released $pred.M$ at line 153: another process q could race ahead and traverse the linked list after the crash, acquire the same lock $pred.M$, then block on $curr.M$ where p effectively still holds the CS, and cause p to block forever while trying to enter $pred.M$. We achieve a better outcome in this scenario if p tries to recover $curr$ first and then $pred$, however even this strategy can cause deadlock if processes access multiple linked list structures and recover them in conflicting orders. This holds even if the RME locks provide wait-free `Recover` and `Exit` sections since a process can block in `Enter`. To summarize, although we can envision correct recovery protocols for linked data structures that use plain RME locks in the style of Golab and Ramaraju, such protocols are much harder to design than algorithm 7 as actions must be ordered carefully to avoid deadlocks.

B Experimental Evaluation

This section presents an empirical comparison of RME algorithms designed specifically for system-wide failures that do not use an explicit failure detector. The hardware platform is a 20-core 2nd generation Intel Xeon processor with Optane Persistent Memory, which supports the CC model. We compare Jayanti, Jayanti and Joshi's RME algorithm for the CC model (Section 3 of [29]), hereby referred to as JJJ, with a simplified version of our RME algorithm from Section 5 that lacks dynamic leaving. Both RME algorithms are based on Lee's ME algorithm [37]. Both algorithms were implemented in C++ using the Intel Persistent Memory Development Kit [45] on Linux. Shared variables were implemented using the `std::atomic` template class, and memory operations were applied with sequential consistency for simplicity. The `libpmemobj` library was used for memory allocation to ensure that shared variables are mapped to persistent memory, and for persistent pointers to deal with address space relocation across failures.

The cache system on our hardware platform is not part of the persistent domain and hence volatile. As a result, if the most recent value of a variable that is meant to be persistent has not been flushed to the persistent memory before a failure, then it will be lost. To



(a) System throughput with CSR property.

(b) System throughput without CSR property.

Figure 2 Scalability comparison of RME algorithms on one processor in failure-free execution.

run correctly on our hardware platform, recoverable algorithms designed for the CC model must be annotated by judicious placement of *persistence* instructions in the source code. The PMDK provides the `pmem_persist` function for this purpose, which internally applies a cache line write-back and store fence. Naïvely persisting variables after every shared memory operation is sufficient to ensure correctness with respect to fundamental correctness properties that refer to the program's state (*e.g.*, ME, SF, CSR), but does not preserve the RMR complexity of busy-wait loops. This is because our hardware platform implements the cache line write-back instruction in a simplistic way that always invalidates the cache line [44]. The naïve approach is easily optimized for algorithms that use simple busy-wait loops, namely ones that spin on a single variable until it reaches a specific value, by persisting the spin variable only after the last iteration of the loop.[3] Even this final persistence instruction can be skipped in the entry section of Lee's ME algorithm since the value of the spin variable is not relevant for recovery when this algorithm is used as a building block of an RME algorithm. It is also safe to omit the persistence instruction when a process reads a single-writer shared variable owned by that process, such as an element of the FACE array in Lee's algorithm. We apply these optimizations to both our and JJJ's algorithm.

Figure 2 presents the scalability of the two algorithms on our hardware platform. Each point plotted is the average of five repetitions, each lasting 5 s. The error bars represent the sample standard deviation, and are imperceptibly small in most cases. The throughput numbers (total number of passages completed per second) presented include the overhead of persistence instructions and the higher latency of persistent memory over DRAM. Figure 2a presents data for variations of the two RME algorithms that satisfy the CSR property, while Figure 2b considers variations of the same algorithms that do not satisfy the CSR property. The non-CSR version of our algorithm is obtained by deleting lines of pseudocode that track CS ownership (specifically, code ending with one "◀"). JJJ's algorithm, in contrast, relies on CS ownership state to achieve mutual exclusion among recovering processes, and so the same transformation is not applicable. Note that Line 14 in Figure 3 of [29] is safe to remove and Line 26 in Figure 3 of [29] must be adjusted.

[3] This also applies for the two spin loops executed in parallel in the entry section of JJJ's algorithm.

Algorithm 8 Equivalence between an RME lock that supports a wait-free `Withdraw` method and an abortable RME lock that supports a wait-free `Exit` method under individual failures.

a : Abortability to Withdrawability.

```
global shared variables
    M: instance of abortable RME lock;
per-process shared/persistent variables
    ABORT_p: abort flag;
Procedure Withdraw()
    ABORT_p := TRUE;
    if M.Recover() then
        if M.Enter() then
            M.Exit();
    ABORT_p := FALSE;
Procedure Recover()
    if ABORT_p then
        Withdraw();
    M.Recover();
Procedure Enter()
    M.Enter();
Procedure Exit()
    M.Exit();
```

b : Withdrawability to Abortability.

```
global shared variables
    M: instance of withdrawable RME lock;
per-process shared/persistent variables
    ABORT_p: abort flag;
boolean Procedure Recover()
    M.Recover() || await ABORT_p;
    if ABORT_p then
        M.Withdraw();
        return FALSE;
    else return TRUE;
boolean Procedure Enter()
    M.Enter() || await ABORT_p;
    if ABORT_p then
        M.Withdraw();
        return FALSE;
    else return TRUE;
Procedure Exit()
    M.Withdraw();
```

As the graphs show, with the CSR property, our algorithm has around 75% and 50% higher throughput than JJJ's algorithm for one and two threads, respectively. The gap stabilizes to around 15% at higher levels of parallelism. Without the CSR property, the gap between the two algorithm is much higher; specifically, our algorithm is 2-3 times faster than JJJ's algorithm. Further, while the performance of JJJ's algorithm is immune to whether or not the CSR property holds, our algorithm sees a significant speedup by a factor of 2-3 when the CSR property is not required.

C Abortability and Withdrawability

To support the notion of abortability in our execution model, we modify the `Recover` and `Enter` methods to return a boolean value. A return value of true indicates that the method was executed to completion, whereas a return value of false indicates the attempt to acquire the lock was abandoned and the method terminated prematurely. An application indicates its desire to abort by raising a boolean flag. Note that an attempt to acquire the lock can only be abandoned if the flag is raised. A process typically executes a passage by invoking the `Recover`, `Enter` and `Exit` methods in order. However, if either `Recover` or `Enter` returns false (which would only happen if the abort flag was raised), then the passage is considered to be complete and subsequent methods are not invoked.

In this section, we show the following equivalence between the two types of RME locks under the *individual* failure model. An RME lock that supports a wait-free `Withdraw` method can be transformed into an abortable RME lock that supports a wait-free `Exit` method. Conversely, an abortable RME lock that supports a wait-free `Exit` method can be transformed into an RME lock that supports a wait-free `Withdraw` method. Further, both transformations only incur $O(1)$ additional RMRs and thus are *RMR preserving*.

Abortability to withdrawability. The transformation is given in algorithm 8a. In the `Withdraw` method of the target lock, a process attempts to execute the `Recover`, `Enter` and `Exit` methods of the base abortable lock in sequence with the abort flag raised (by the algorithm). If either `Recover` or `Enter` returns false (indicating that the attempt to acquire the lock was abandoned), the method immediately returns without invoking the remaining methods. Otherwise, the process completes the super-passage of the base abortable lock by executing the `Exit` method. Clearly, all three methods are wait-free in the presence of the abort signal.

Withdrawability to abortability. The transformation is given in algorithm 8b. The main idea is that, in the `Recover` (respectively, `Enter`) method of the target lock, the process invokes the `Recover` (respectively, `Enter`) method of the base withdrawable lock and concurrently monitors the abort flag. If the abort flag is raised (by the environment), the process *simulates* a failure by prematurely terminating the method, and executes the `Withdraw` method instead to complete the super-passage of the base lock.

Optimal Computation in Leaderless and Multi-Leader Disconnected Anonymous Dynamic Networks

Giuseppe A. Di Luna[1] ✉
DIAG, Sapienza University of Rome, Italy

Giovanni Viglietta[1] ✉
Department of Computer Science and Engineering, University of Aizu, Japan

Abstract

We give a simple characterization of which functions can be computed deterministically by anonymous processes in dynamic networks, depending on the number of leaders in the network. In addition, we provide efficient distributed algorithms for computing all such functions assuming minimal or no knowledge about the network. Each of our algorithms comes in two versions: one that terminates with the correct output and a faster one that stabilizes on the correct output without explicit termination. Notably, these are the first deterministic algorithms whose running times scale *linearly* with both the number of processes and a parameter of the network which we call *dynamic disconnectivity* (meaning that our dynamic networks do not necessarily have to be connected at all times). We also provide matching lower bounds, showing that all our algorithms are asymptotically *optimal* for any fixed number of leaders.

While most of the existing literature on anonymous dynamic networks relies on classical mass-distribution techniques, our work makes use of a recently introduced combinatorial structure called *history tree*, also developing its theory in new directions. Among other contributions, our results make definitive progress on two popular fundamental problems for anonymous dynamic networks: leaderless *Average Consensus* (i.e., computing the mean value of input numbers distributed among the processes) and multi-leader *Counting* (i.e., determining the exact number of processes in the network). In fact, our approach unifies and improves upon several independent lines of research on anonymous networks, including Nedić et al., IEEE Trans. Automat. Contr. 2009; Olshevsky, SIAM J. Control Optim. 2017; Kowalski–Mosteiro, ICALP 2019, SPAA 2021; Di Luna–Viglietta, FOCS 2022.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Computing methodologies → Distributed algorithms

Keywords and phrases anonymous dynamic network, leaderless network, disconnected network, history tree

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.18

Related Version *Full Version*: `https://arxiv.org/abs/2207.08061`

Funding Giovanni Viglietta was partially funded by JSPS KAKENHI Grant number 23K10985.

1 Introduction

Dynamic networks. An increasingly prominent area of distributed computing focuses on algorithmic aspects of *dynamic networks*, motivated by novel technologies such as wireless sensors networks, software-defined networks, networks of smart devices, and other networks with a continuously changing topology [9, 32, 34]. Typically, a network is modeled by a system of n *processes* that communicate in synchronous rounds; at each round, the network's topology changes unpredictably.

[1] Both authors contributed equally to this research.

Disconnected networks. In the dynamic setting, a common assumption is that the network is *1-interval-connected*, i.e., connected at all rounds [30, 37]. However, this is not a suitable model for many real systems, due to the very nature of dynamic entities (think of P2P networks of smart devices moving unpredictably) or due to transient communication failures, which may compromise the network's connectivity. A weaker assumption is that the union of all the network's links across any T consecutive rounds induces a connected graph on the processes [28, 39]. We say that such a network is *T-union-connected*, and we call $T \geq 1$ its *dynamic disconnectivity*.[2]

Anonymous processes. Several works have focused on processes with unique IDs, which allow for efficient algorithms for many different tasks [8, 29, 30, 31, 34, 37]. However, unique IDs may not be available due to operational limitations [37] or to protect user privacy: A famous example are COVID-19 tracking apps, where assigning temporary random IDs to users was not enough to eliminate privacy concerns [43]. Systems where processes are indistinguishable are called *anonymous*. The study of *static* anonymous networks has a long history, as well [6, 7, 10, 11, 12, 21, 42, 45].

Networks with leaders. It is known that several fundamental problems for anonymous networks (a notable example being the *Counting problem*, i.e., determining the total number of processes n) cannot be solved without additional "symmetry-breaking" assumptions. The most typical choice is the presence of a single distinguished process called *leader* [1, 2, 3, 4, 16, 21, 23, 25, 33, 41, 46] or, less commonly, a subset of several leaders (and knowledge of their number) [24, 26, 27, 28].

Apart from the theoretical importance of generalizing the usual single-leader scenario, studying networks with multiple leaders also has a practical impact in terms of privacy. Indeed, while the communications of a single leader can be traced, the addition of more leaders provides differential privacy for each of them.

Leaderless networks. In some networks, the presence of reliable leaders may not always be guaranteed: For example, in a mobile sensor network deployed by an aircraft, the leaders may be destroyed as a result of a bad landing; also, the leaders may malfunction during the system's lifetime. This justifies the extensive existing literature on networks with no leaders [14, 15, 35, 36, 38, 44, 47]. Notably, a large portion of works on leaderless networks have focused on the *Average Consensus problem*, where the goal is to compute the mean of a list of numbers distributed among the processes [5, 13, 14, 20, 39, 40].

1.1 Our Contributions

Summary. Focusing on anonymous dynamic networks, in this paper we completely elucidate the relationship between leaderless networks and networks with (multiple) leaders, as well as the impact of the dynamic disconnectivity T on the efficiency of distributed algorithms. We remark that only a minority of existing works consider networks that are not necessarily connected at all times.

The full version of this paper is found at [19].

[2] We use the term "disconnected" to refer to T-union-connected networks in the sense that they may not be connected at any round. It is worth noting that non-trivial (terminating) computation requires some conditions on temporal connectivity to be met, such as a finite dynamic disconnectivity and its knowledge by all processes (refer to Proposition 2).

Computability. We give an exact characterization of which functions can be computed in anonymous dynamic networks with and without leaders, respectively. Namely, with at least one leader, all the so-called *multi-aggregate functions* are computable; with no leaders, only the *frequency-based multi-aggregate functions* are computable (see Section 2 for definitions). Interestingly, computability is independent of the dynamic disconnectivity T. Our contribution considerably generalizes a recent result on the functions computable with exactly one leader and with $T = 1$ [17].

Complete problems. While computing the so-called *Generalized Counting function* F_{GC} was already known to be a complete problem for the class of multi-aggregate functions [17], in this work we expand the picture by identifying a complete problem for the class of frequency-based multi-aggregate functions, as well: the *Frequency function* F_R (both F_{GC} and F_R are defined in Section 2). By "complete problem" we mean that computing such a function allows the immediate computation of any other function in the class with no overhead in terms of communication rounds.

Algorithms. We give efficient deterministic algorithms for computing the Frequency function (Section 3) and the Generalized Counting function (Section 4). Since the two problems are complete, we automatically obtain efficient algorithms for computing *all* functions in the respective classes.

For each problem, we give two algorithms: a *terminating* version, where each process is required to commit on its output and never change it, and a *stabilizing* version, where processes are allowed to modify their outputs, provided that they eventually stabilize on the correct output.

The stabilizing algorithms for both problems run in $2Tn$ rounds regardless of the number of leaders, and do not require any knowledge of the dynamic disconnectivity T or the number of processes n. Our terminating algorithm for leaderless networks runs in $T(n + N)$ rounds with knowledge of T and an upper bound $N \geq n$; the terminating algorithm for $\ell \geq 1$ leaders runs in $(\ell^2 + \ell + 1)Tn$ rounds with no knowledge about n. The latter running time is reasonable (i.e., linear) in most applications, as ℓ is typically a constant or very small compared to n.

A comparison of our results with the state of the art on Average Consensus and Counting problems is illustrated in Table 1 and discussed in Appendix A.

Negative results. Some of our algorithms assume processes to have a-priori knowledge of some parameters of the network; in Section 5 we show that all of these assumptions are necessary. We also provide lower bounds that asymptotically match our algorithms' running times, assuming that the number of leaders ℓ is constant (which is a realistic assumption in most applications).

Multigraphs. All of our results hold more generally if networks are modeled as multigraphs, as opposed to the simple graphs traditionally encountered in nearly all of the literature. This is relevant in many applications: in radio communication, for instance, multiple links between processes naturally appear due to the multi-path propagation of radio waves.

Table 1 Comparing results for Average Consensus and Counting in anonymous dynamic networks. For algorithms that support disconnected networks, T indicates the dynamic disconnectivity.

Problem	*Reference*	*Leaders*	*Disconn.*	*Term.*	*Notes*	*Running time*
Average Consensus	[35]	$\ell = 0$	✓		ϵ-convergence, T unknown, upper bound on processes' degrees known	$O(Tn^3 \log(1/\epsilon))$
	[14]	$\ell = 0$			ϵ-convergence	$O(n^4 \log(n/\epsilon))$
	[13]	$\ell = 0$			randomized Monte Carlo	$O(n)$
	[26]	$\ell \geq 1$		✓	ℓ known	$O(n^5 \log^3(n)/\ell)$
	this work	$\ell = 0$	✓		T unknown	$2Tn$
		$\ell = 0$	✓	✓	T and $N \geq n$ known	$T(n+N)$
(Generalized) Counting	[17]	$\ell = 1$				$2n-2$
		$\ell = 1$		✓		$3n-2$
	[28]	$\ell \geq 1$		✓	ℓ known	$O(n^4 \log^3(n)/\ell)$
	[27]	$\ell \geq 1$	✓	✓	ℓ and T known, $O(\log n)$-size messages	$\widetilde{O}(n^{2T+3}/\ell)$
	this work	$\ell \geq 1$	✓		ℓ known, T unknown	$2Tn$
		$\ell \geq 1$	✓	✓	ℓ and T known	$(\ell^2+\ell+1)Tn$

Table 1: Comparing results for Average Consensus and Counting in anonymous dynamic networks. For algorithms that support disconnected networks, T indicates the dynamic disconnectivity.

2 Definitions and Preliminaries

Problem	*Reference*	*Leaders*	*Disconn.*	*Term.*	*Notes*	*Running time*
Average Consensus	[11]	$\ell = 0$	✓		ϵ-convergence, T unknown, upper bound on processes' degrees known	$O(Tn^3 \log(1/\epsilon))$
	[16]	$\ell = 0$			ϵ-convergence	$O(n^4 \log(n/\epsilon))$
	[15]	$\ell = 0$			randomized Monte Carlo	$O(n)$
	[31]	$\ell \geq 1$		✓	ℓ known	$O(n^5 \log^3(n)/\ell)$
	this work	$\ell = 0$	✓		T unknown	$2Tn$
		$\ell = 0$	✓	✓	T and $N \geq n$ known	$T(n+N)$
(Generalized) Counting	[24]	$\ell = 1$				$2n-2$
		$\ell = 1$		✓		$3n-2$
	[33]	$\ell \geq 1$		✓	ℓ known	$O(n^4 \log^3(n)/\ell)$
	[32]	$\ell \geq 1$	✓	✓	ℓ and T known, $O(\log n)$-size messages	$\widetilde{O}(n^{2T+3}/\ell)$
	this work	$\ell \geq 1$	✓		ℓ known, T unknown	$2Tn$
		$\ell \geq 1$	✓	✓	ℓ and T known	$(\ell^2+\ell+1)Tn$

Figure 1: Comparing results for Average Consensus and Counting in anonymous dynamic networks. For algorithms that support disconnected networks, T indicates the dynamic disconnectivity.

We will give preliminary definitions and results, and recall some properties of history trees from [17].

Processes and networks. A *dynamic network* is modeled by an infinite sequence $\mathcal{G} = (G_t)_{t \geq 1}$, where $G_t = (V, E_t)$ is an undirected multigraph whose vertex set $V = \{p_1, p_2, \ldots, p_n\}$ is a system of n *anonymous processes* and E_t is a multiset of edges representing *links* between processes.

Each process p_i starts with an *input* $\lambda(p_i)$, which is assigned to it at *round* 0. It also has an internal state, which is initially determined by $\lambda(p_i)$. At each *round* $t \geq 1$, every process composes a message (depending on its internal state) and broadcasts it to its neighbors in G_t through all its incident links. By the end of round t, each process reads all messages coming from its neighbors and updates its internal state according to a local algorithm $\mathcal{A}$. Note that $\mathcal{A}$ is deterministic and is the same for all processes. The input of each process also includes a *leader flag*. The processes whose leader flag is set are called *leaders* (or *supervisors*). We will denote the number of leaders as ℓ.

Each process also returns an *output* at the end of each round, which is determined by its current internal state. A system is said to *stabilize* if the outputs of all its processes remain constant from a certain round onward; note that a process' internal state may still change even when its output is constant. A process may also decide to explicitly *terminate* and no longer update its internal state. When all processes have terminated, the system is said to *terminate* as well.

We say that $\mathcal{A}$ *computes* a function F if, whenever the processes are assigned inputs $\lambda(p_1), \lambda(p_2), \ldots, \lambda(p_n)$ and all processes execute the local algorithm $\mathcal{A}$ at every round, the system eventually stabilizes with each process p_i giving the desired output $F(p_i, \lambda)$. A stronger notion of computation requires the system to not only stabilize but also to explicitly terminate with the correct output. The (worst-case) *running time* of $\mathcal{A}$, as a function of n, is the maximum number of rounds it takes for the system to stabilize (and optionally terminate), taken across all possible dynamic networks of size n and all possible input assignments.

leader flag is set are called *leaders* (or *supervisors*). We will denote the number of leaders as ℓ.

Each process also returns an *output* at the end of each round, which is determined by its current internal state. A system is said to *stabilize* if the outputs of all its processes remain constant from a certain round onward; note that a process' internal state may still change even when its output is constant. A process may also decide to explicitly *terminate* and no longer update its internal state. When all processes have terminated, the system is said to *terminate* as well.

We say that $\mathcal{A}$ *computes* a function F if, whenever the processes are assigned inputs $\lambda(p_1), \lambda(p_2), \ldots, \lambda(p_n)$ and all processes execute the local algorithm $\mathcal{A}$ at every round, the system eventually stabilizes with each process p_i giving the desired output $F(p_i, \lambda)$. A stronger notion of computation requires the system to not only stabilize but also to explicitly terminate with the correct output. The (worst-case) *running time* of $\mathcal{A}$, as a function of n, is the maximum number of rounds it takes for the system to stabilize (and optionally terminate), taken across all possible dynamic networks of size n and all possible input assignments.

Classes of functions. Let $\mu = \{(z_1, m_1), (z_2, m_2), \ldots, (z_k, m_k)\}$ be the multiset of all processes'

Classes of functions. Let $\mu_\lambda = \{(z_1, m_1), (z_2, m_2), \dots, (z_k, m_k)\}$ be the multiset of all processes' inputs. That is, for all $1 \le i \le k$, there are exactly m_i processes $p_{j_1}, p_{j_2}, \dots, p_{j_{m_i}}$ whose input is $z_i = \lambda(p_{j_1}) = \lambda(p_{j_2}) = \dots = \lambda(p_{j_{m_i}})$; note that $n = \sum_{i=1}^{k} m_i$. A *multi-aggregate* function is defined as a function F of the form $F(p_i, \lambda) = \psi(\lambda(p_i), \mu_\lambda)$, i.e., such that the output of each process depends only on its own input and the multiset of all processes' inputs.

The special multi-aggregate functions $F_C(p_i, \lambda) = n$ and $F_{GC}(p_i, \lambda) = \mu_\lambda$ are called the *Counting* function and the *Generalized Counting* function, respectively. It is known that, if a system can compute the Generalized Counting function F_{GC}, then it can compute any multi-aggregate function in the same number of rounds: thus, F_{GC} is *complete* for the class of multi-aggregate functions [17].

For any $\alpha \in \mathbb{R}^+$, we define $\alpha \cdot \mu_\lambda$ as $\{(z_1, \alpha \cdot m_1), (z_2, \alpha \cdot m_2), \dots, (z_k, \alpha \cdot m_k)\}$. We say that a multi-aggregate function $F(p_i, \lambda) = \psi(\lambda(p_i), \mu_\lambda)$ is *frequency-based* if $\psi(z, \mu_\lambda) = \psi(z, \alpha \cdot \mu_\lambda)$ for every positive integer α and every input z (see [22]). That is, F depends only on the "frequency" of each input in the system, rather than on their actual multiplicities. Notable examples include statistical functions such as mean, variance, maximum, median, mode, etc. The problem of computing the mean of all input values is called *Average Consensus* [5, 13, 14, 15, 20, 26, 35, 36, 38, 39, 40, 44, 47].

The frequency-based multi-aggregate function $F_R(p_i, \lambda) = \frac{1}{n} \cdot \mu_\lambda$ is called *Frequency* function, and is complete for the class of frequency-based multi-aggregate functions, as stated below (the proof is simple, and is found in [19]).

▶ **Proposition 1.** *If F_R can be computed (with termination), then all frequency-based multi-aggregate functions can be computed (with termination) in the same number of rounds, as well.* ◀

History trees. *History trees* were introduced in [17] as a tool of investigation for anonymous dynamic networks; an example is found in Figure 1. A history tree is a representation of a dynamic network given some inputs to the processes. It is an infinite graph whose nodes are partitioned into *levels* L_t, with $t \ge -1$; each node in L_t represents a class of processes that are *indistinguishable* at the end of round t (with the exception of L_{-1}, which contains a single node r representing all processes). The definition of distinguishability is inductive: at the end of round 0, two processes are distinguishable if and only if they have different inputs. At the end of round $t \ge 1$, two processes are distinguishable if and only if they were already distinguishable at round $t-1$ or if they have received different multisets of messages at round t.

Each node in level L_0 has a label indicating the input of the processes it represents. There are also two types of edges connecting nodes in adjacent levels. The *black edges* induce an infinite tree rooted at node $r \in L_{-1}$ which spans all nodes. The presence of a black edge $\{v, v'\}$, with $v \in L_t$ and $v' \in L_{t+1}$, indicates that the *child node* v' represents a subset of the processes represented by the *parent node* v. The *red multi-edges* represent communications between processes. The presence of a red edge $\{v, v'\}$ with multiplicity m, with $v \in L_t$ and $v' \in L_{t+1}$, indicates that, at round $t+1$, each process represented by v' receives m (identical) messages from processes represented by v.

As time progresses and processes exchange messages, they are able to locally construct finite portions of the history tree. In [17], it is shown that there is a local algorithm $\mathcal{A}^*$ that allows each process to locally construct and update its own *view* of the history tree at every round. The view of a process p at round $t \ge 0$ is the subgraph of the history tree which is spanned by all the shortest paths (using black and red edges indifferently) from the root r to the node in L_t representing p (see Figure 1). As proved in [17, Theorem 3.1], the view of a process at round t contains all the information that the process may be able

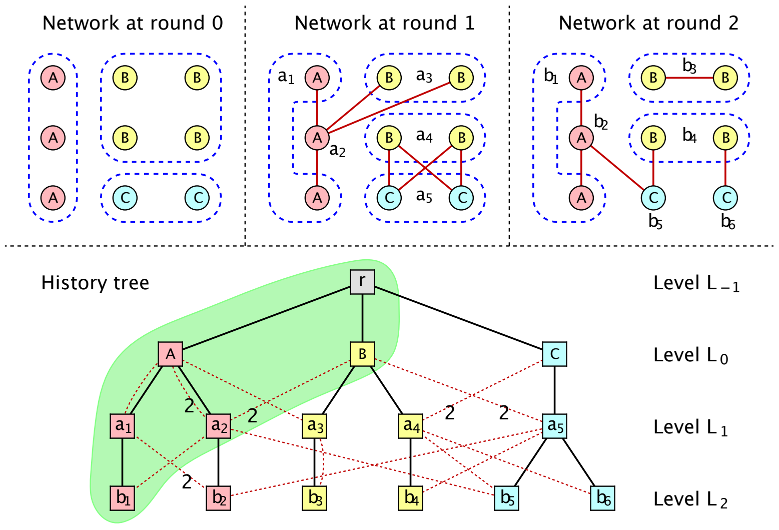


Figure 1 The first rounds of a dynamic network with $n = 9$ processes and the corresponding levels of the history tree. Level L_t consists of all nodes at distance $t + 1$ from the root r. The multiplicities of the red multi-edges of the history tree are explicitly indicated only when greater than 1. The letters A, B, C denote processes' inputs; all other labels have been added for the reader's convenience, and indicate classes of indistinguishable processes (non-trivial classes are also indicated by dashed blue lines). Note that the two processes in b_4 are still indistinguishable at the end of round 2, although they are linked to the distinguishable processes b_5 and b_6. This is because such processes were in the same class a_5 at round 1. The subgraph in the green blob is the *view* of the two processes in b_1.

to use at that round. This justifies the convention that all processes always execute $\mathcal{A}^*$, constructing their local view of the history tree and broadcasting (a representation of) it at every round, regardless of their task. Then, they simply compute their task-dependent outputs as a function of their respective views.

We define the *anonymity* of a node v of the history tree as the number of processes that v represents, and we denote it as $a(v)$. It follows that $\sum_{v \in L_t} a(v) = n$ for all $t \geq -1$, and that the anonymity of a node is equal to the sum of the anonymities of its children. Naturally, a process is not aware of the anonymities of the nodes in its view of the history tree, unless it can somehow infer them from the view's structure itself. In fact, computing the Generalized Counting function is equivalent to determining the anonymities of all the nodes in L_0. Similarly, computing the Frequency function corresponds to determining the value $a(v)/n$ for all $v \in L_0$.

Computation in disconnected networks. Although the network G_t at each individual round may be disconnected, we assume the dynamic network to be T*-union-connected*. That is, there is a *dynamic disconnectivity* parameter $T \geq 1$ such that the sum of any T consecutive G_t's is a connected multigraph. Thus, for all $i \geq 1$, the multigraph $\left(V, \bigcup_{t=i}^{i+T-1} E_t\right)$ is connected (we remark that a union of multisets adds together the multiplicities of equal elements).[3] The next two results are easy to prove (see [19]).

[3] Our T-union-connected networks should not be confused with the T-*interval*-connected networks from [30]. In those networks, the *intersection* (as opposed to the union) of any T consecutive E_t's induces a connected (multi)graph. In particular, a T-interval-connected network is connected at every round, while a T-union-connected network may not be, unless $T = 1$. Incidentally, a network is 1-interval-connected if and only if it is 1-union-connected.

▶ **Proposition 2.** *Any non-trivial function is impossible to compute with termination unless the processes have some knowledge about T. (A function is "trivial" if it can be computed locally.)* ⌟

▶ **Proposition 3.** *A function F can be computed (with termination) within $f(n)$ rounds in any dynamic network with $T = 1$ if and only if F can be computed (with termination) within $T \cdot f(n)$ rounds in any dynamic network with $T \geq 1$, assuming that T is known to all processes.* ⌟

Relationship with the dynamic diameter. A concept closely related to the dynamic disconnectivity T of a network is its *dynamic diameter* (or *temporal diameter*) D, which is defined as the maximum number of rounds it may take for information to travel from any process to any other process at any point in time [9, 32]. It is a simple observation that $T \leq D \leq T(n-1)$.

We chose to use T, as opposed to D, to measure the running times of our algorithms for several reasons. Firstly, T is well defined (i.e., finite) if and only if D is; however, T has a simpler definition, and is arguably easier to directly estimate or enforce in a real network. Secondly, Proposition 3, as well as all of our theorems, remain valid if we replace T with D; nonetheless, stating the running times of our algorithms in terms of T is better, because $T \leq D$.

3 Computation in Leaderless Networks

We will give a stabilizing and a terminating algorithm that efficiently compute the Frequency function F_R in all leaderless networks with finite dynamic disconnectivity T. As a consequence, *all* frequency-based multi-aggregate functions are efficiently computable as well, due to Proposition 1. Moreover, Proposition 16 states that no other functions are computable in leaderless networks, and Proposition 17 shows that our algorithms are asymptotically optimal. All missing proofs are found in [19].

3.1 Stabilizing Algorithm

We will use the procedure in Listing 1 as a subroutine in some of our algorithms. Its purpose is to construct a homogeneous system of $k-1$ independent linear equations involving the anonymities of all the k nodes in a level of a process' view. We will first give some definitions.

In (a view of) a history tree, if a node $v \in L_t$ has exactly one child (i.e., there is exactly one node $v' \in L_{t+1}$ such that $\{v, v'\}$ is a black edge), we say that v is *non-branching*. We say that two non-branching nodes $v_1, v_2 \in L_t$, whose respective children are $v'_1, v'_2 \in L_{t+1}$, are *exposed* with multiplicity (m_1, m_2) if the red edges $\{v'_1, v_2\}$ and $\{v'_2, v_1\}$ are present with multiplicities $m_1 \geq 1$ and $m_2 \geq 1$, respectively. A *strand* is a path $(w_1, w_2, \dots, w_k)$ in (a view of) a history tree consisting of non-branching nodes such that, for all $1 \leq i < k$, the node w_i is the parent of w_{i+1}. We say that two strands P_1 and P_2 are *exposed* if there are two exposed nodes $v_1 \in P_1$ and $v_2 \in P_2$.

Intuitively, the procedure in Listing 1 searches for a long-enough sequence of levels in the given view $\mathcal{V}$, say from L_s to L_t, where all nodes are non-branching. That is, the nodes in $L_s \cup L_{s+1} \cup \dots \cup L_t$ can be partitioned into $k = |L_s| = |L_t|$ strands. Then the procedure searches for pairs of exposed strands, each of which yields a linear equation involving the anonymities of some nodes of L_t, until it obtains $k-1$ linearly independent equations. Note

Listing 1 Constructing a system of equations in the anonymities of some nodes in a view.

```
# Input: a view $\mathcal{V}$ with levels $L_{-1}$, $L_0$, $L_1$, ..., $L_h$
# Output: $(t, S)$, where $t$ is an integer and $S$ is a system of linear equations

Assign $s := 0$
For $t := 0$ to $h$
   If $L_t$ contains a node with no children, return $(-1, \emptyset)$
   If $L_t$ contains a node with more than one child, assign $s := t + 1$
   Else
      Let $k = |L_s| = |L_t|$ and let $u_i$ be the $i$th node in $L_t$
      Let $P_i$ be the strand starting in $L_s$ and ending in $u_i \in L_t$
      Let $\mathcal{P} = \{P_1, P_2, \dots, P_k\}$
      Let $G$ be the graph on $\mathcal{P}$ whose edges are pairs of exposed strands
      If $G$ is connected
         Let $G' \subseteq G$ be any spanning tree of $G$
         Assign $S := \emptyset$
         For each edge $\{P_i, P_j\}$ of $G'$
            Find any two exposed nodes $v_1 \in P_i$ and $v_2 \in P_j$
            Let $(m_1, m_2)$ be the multiplicity of the exposed pair $(v_1, v_2)$
            Add to $S$ the equation $m_1 x_i = m_2 x_j$
         Return $(t, S)$
```

that the search may fail (in which case Listing 1 returns $t = -1$) or it may produce incorrect equations. The following lemma specifies sufficient conditions for Listing 1 to return a correct and non-trivial system of equations for some $t \geq 0$ (the proof is in [19]).

▶ **Lemma 4.** *Let $\mathcal{V}$ be the view of a process in a T-union-connected network of size n taken at round t', and let Listing 1 return (t, S) on input $\mathcal{V}$. Assume that one of the following conditions holds:*

1. *$t \geq 0$ and $t' \geq t + Tn$, or*
2. *$t' \geq 2Tn$.*

Then, $0 \leq t \leq Tn$, and S is a homogeneous system of $k - 1$ independent linear equations (with integer coefficients) in $k = |L_t|$ variables x_1, x_2, ..., x_k. Moreover, S is satisfied by assigning to x_i the anonymity of the ith node of L_t, for all $1 \leq i \leq k$. ⌟

▶ **Theorem 5.** *There is an algorithm that computes F_R in all T-union-connected anonymous networks with no leader and stabilizes in at most $2Tn$ rounds, assuming no knowledge of T or n.* ⌟

3.2 Terminating Algorithm

We will now give a certificate of correctness that can be used to turn the stabilizing algorithm of Theorem 5 into a terminating algorithm. The certificate relies on a-priori knowledge of the dynamic disconnectivity T and an upper bound N on the size of the network n; these assumptions are justified by Proposition 2 and Proposition 18, respectively.

▶ **Theorem 6.** *There is an algorithm that computes F_R in all T-union-connected anonymous networks with no leader and terminates in at most $T(n + N)$ rounds, assuming that T and an upper bound $N \geq n$ are known to all processes.*[4] ⌟

[4] If the dynamic diameter D of the network is known, the termination time improves to $Tn + D$ rounds.

4 Computation in Networks with Leaders

We will give a stabilizing and a terminating algorithm that efficiently compute the Generalized Counting function F_{GC} in all networks with $\ell \geq 1$ leaders and finite dynamic disconnectivity T. Therefore, *all* multi-aggregate functions are efficiently computable as well, due to [17, Theorem 2.1]. Moreover, Proposition 19 states that no other functions are computable in networks with leaders, and Proposition 21 shows that our algorithms are asymptotically optimal for any fixed $\ell \geq 1$.

4.1 Stabilizing Algorithm

We will once again make use of the subroutine in Listing 1, this time assuming that the number of leaders $\ell \geq 1$ is known to all processes. This assumption is justified by Proposition 20. The next theorem uses the same ideas as Theorems 5 and 6, and is proved in [19].

▶ **Theorem 7.** *There is an algorithm that computes F_{GC} in all T-union-connected anonymous networks with $\ell \geq 1$ leaders and stabilizes in at most $2Tn$ rounds, assuming that ℓ is known to all processes, but assuming no knowledge of T or n.* ⌟

4.2 Terminating Algorithm

We will now present the main result of this paper. As already remarked, giving an efficient certificate of correctness for the (Generalized) Counting problem with multiple leaders is a highly non-trivial task for which a radically different approach is required. Note that it is not possible to simply adapt the single-leader algorithm in [17] by setting the anonymity of the leader node in the history tree to ℓ instead of 1. Indeed, as soon as some leaders get disambiguated, the leader node splits into several children nodes whose anonymities are unknown (we only know that their sum is ℓ). There is no way around this difficulty other than developing a new technique.

Our algorithm is rather involved, and the proofs of several technical lemmas are provided in [19], due to lack of space.

The subroutine `ApproxCount`. A large portion of this section is devoted to a subroutine called `ApproxCount`, which will be repeatedly invoked by our main algorithm. The purpose of `ApproxCount` is to compute an approximation n' of the total number of processes n (or report various types of failure). It takes as input a view $\mathcal{V}$ of a process, the number of leaders ℓ, and two integer parameters s and x, representing the index of a level of $\mathcal{V}$ and the anonymity of a leader node in L_s, respectively.

The subroutine roughly follows the general structure of the algorithm in [17, Section 4.2]: namely, anonymities are first "guessed" and then proven correct when some "certificates" are satisfied. However, the way these basic concepts are defined and the way the underlying principles are implemented is entirely new, due to the added difficulty that here we have a strand of leader nodes in the view $\mathcal{V}$ hanging from the first leader node τ in level L_s, where the anonymity $a(\tau)$ is an unknown number not greater than ℓ (as opposed to $a(\tau) = 1$, which is assumed in [17]).

`ApproxCount` begins by assuming that $a(\tau)$ is the given parameter x, and then it makes deductions on the anonymities of other nodes until it is able to make an estimate $n' > 0$ on the total number of processes, or report failure in the form of an *error code* $n' \in \{-1, -2, -3\}$. In particular, since the algorithm requires the existence of a long-enough strand hanging from τ, it reports failure if some descendants of τ (in the relevant levels of $\mathcal{V}$) have more than one child.

Another important difficulty that is unique to the multi-leader case is that, even if $\mathcal{V}$ contains a long-enough strand of leader nodes, some nodes in the strand may still be branching in the history tree (that is, the chain of leader nodes is branching, but only one branch appears in $\mathcal{V}$). We will have to keep this in mind when reasoning about $\mathcal{V}$.

In the following, we give some preliminary definitions and results in order to formally state our subroutine and prove its correctness and running time. We remark that `ApproxCount` assumes that the network is 1-union-connected, as this is sufficient for our main algorithm to work for any T-union-connected network (refer to the proof of Theorem 15).

Discrepancy δ. Suppose that `ApproxCount` is invoked with arguments $\mathcal{V}$, s, x, ℓ, where $1 \leq x \leq \ell$, and let τ be the first leader node in level L_s of $\mathcal{V}$ (if τ does not exist, the procedure immediately returns the error code $n' = -1$). We define the *discrepancy* δ as the ratio $x/a(\tau)$. Clearly, $1/\ell \leq \delta \leq \ell$. Note that, since $a(\tau)$ is not a-priori known by the process executing `ApproxCount`, then neither is δ.

Conditional anonymity. `ApproxCount` starts by assuming that the anonymity of τ is x, and makes deductions on other anonymities based on this assumption. Thus, we will distinguish between the actual anonymity of a node $a(v)$ and the *conditional anonymity* $a'(v) = \delta a(v)$ that `ApproxCount` may compute under the initial assumption that $a'(\tau) = x = \delta a(\tau)$.

Guessing conditional anonymities. Let u be a node of a history tree, and assume that the conditional anonymities of all its children u_1, u_2, ..., u_k have been computed: such a node u is called a *guesser*. If v is not among the children of u but it is at their same level, and the red edge $\{v, u\}$ is present with multiplicity $m \geq 1$, we say that v is *guessable* by u. In this case, we can make a *guess* $g(v)$ on the conditional anonymity $a'(v)$:

$$g(v) = \frac{a'(u_1) \cdot m_1 + a'(u_2) \cdot m_2 + \cdots + a'(u_k) \cdot m_k}{m}, \tag{1}$$

where m_i is the multiplicity of the red edge $\{u_i, v'\}$ for all $1 \leq i \leq k$, and v' is the parent of v (possibly, $m_i = 0$). Note that $g(v)$ may not be an integer. Although a guess may be inaccurate, it never underestimates the conditional anonymity:

▶ **Lemma 8.** *If v is guessable, then $g(v) \geq a'(v)$. Moreover, if v has no siblings, $g(v) = a'(v)$.* ⌟

Heavy nodes. The subroutine `ApproxCount` assigns guesses in a *well-spread* fashion, i.e., in such a way that at most one node per level is assigned a guess.

Suppose now that a node v has been assigned a guess. We define its *weight* $w(v)$ as the number of nodes in the subtree hanging from v that have been assigned a guess (this includes v itself). Recall that subtrees are determined by black edges only. We say that v is *heavy* if $w(v) \geq \lfloor g(v) \rfloor$.

▶ **Lemma 9.** *Assume that $\delta \geq 1$. In a well-spread assignment of guesses, if $w(v) > a'(v)$, then some descendants of v are heavy (the* descendants *of v are the nodes in the subtree hanging from v other than v itself).* ⌟

Correct guesses. We say that a node v has a *correct* guess if v has been assigned a guess and $g(v) = a'(v)$. The next lemma gives a criterion to determine if a guess is correct.

Listing 2 The subroutine ApproxCount invoked in Listing 3.

```
# Input: a view $\mathcal{V}$ and three integers $s$, $x$, $\ell$
# Output: a pair of integers $(n', t)$

Let $L_{-1}$, $L_0$, $L_1$, ... be the levels of $\mathcal{V}$
Assign $t := s$
If $L_s$ does not contain any leader nodes, return $(-1, t)$
Let $\tau$ be the first leader node in $L_s$
Mark all nodes in $\mathcal{V}$ as not guessed and not counted
Assign $u := \tau$; assign $a'(u) := x$; mark $u$ as counted
While $u$ has a unique child $u'$ in $\mathcal{V}$
   Assign $u := u'$; assign $a'(u) := x$; mark $u$ as counted
While there are guessable levels and a counting cut has not been found
   Let $v$ be a guessable non-counted node of smallest depth in $\mathcal{V}$
   Let $L_{t'}$ be the level of $v$; assign $t := \max\{t, t'\}$
   Assign a guess $g(v)$ to $v$ as in Equation (1); mark $v$ as guessed
   Let $P_v$ be the black path from $v$ to its ancestor in $L_s$
   If there is a heavy node in $P_v$
      Let $v'$ be the heavy node in $P_v$ of maximum depth
      If $g(v')$ is not an integer, return $(-3, t)$
      Assign $a'(v') := g(v')$; mark $v'$ as counted and not guessed
      If $v'$ is the root or a leaf of a non-trivial complete isle $I$
         For each internal node $w$ of $I$
            Assign $a'(w) := \sum_{w' \text{ leaf of } I \text{ and descendant of } w} a'(w')$
            Mark $w$ as counted and not guessed
If no counting cut has been found, return $(-2, t)$
Else
   Let $C$ be a counting cut between $L_s$ and $L_t$
   Let $n' = \sum_{v \in C} a'(v)$
   Let $\ell' = \sum_{v \text{ leader node in } C} a'(v)$
   If $\ell' < \ell$, return $(-1, t)$
   If $\ell' > \ell$, return $(-3, t)$
   Return $(n', t)$
```

▶ **Lemma 10.** *Assume that $\delta \geq 1$. In a well-spread assignment of guesses, if a node v is heavy and no descendant of v is heavy, then v has a correct guess or the guess on v is not an integer.* ◀

When the criterion in Lemma 10 applies to a node v, we say that v has been *counted.* So, counted nodes are nodes that have been assigned a guess, which was then confirmed to be the correct conditional anonymity.

Cuts and isles. Fix a view $\mathcal{V}$ of a history tree $\mathcal{H}$. A set of nodes C in $\mathcal{V}$ is said to be a *cut* for a node $v \notin C$ of $\mathcal{V}$ if two conditions hold: (i) for every leaf v' of $\mathcal{V}$ that lies in the subtree hanging from v, the black path from v to v' contains a node of C, and (ii) no proper subset of C satisfies condition (i). A cut for the root r whose nodes are all counted is said to be a *counting cut.*

Let s be a counted node in $\mathcal{V}$, and let F be a cut for v whose nodes are all counted. Then, the set of nodes spanned by the black paths from s to the nodes of F is called *isle*; s is the *root* of the isle, while each node in F is a *leaf* of the isle. The nodes in an isle other than the root and the leaves are called *internal.* An isle is said to be *trivial* if it has no internal nodes.

Listing 3 Solving the Counting problem with $\ell \geq 1$ leaders.

```
# Input: a view $\mathcal{V}$ and a positive integer $\ell$
# Output: either a positive integer $n$ or "Unknown"

Assign $n^* := -1$ and $s := 0$ and $c := 0$
Let $b$ be the number of leader branches in $\mathcal{V}$
While $c \leq \ell - b$
   Assign $t^* := -1$
   For $x := \ell$ downto 1
      Assign $(n', t) :=$ ApproxCount($\mathcal{V}, s, x, \ell$)       # see Listing 2
      Assign $t^* := \max\{t^*, t\}$
      If $n' = -1$, return "Unknown"
      If $n' = -2$, break out of the for loop
      If $n' > 0$
         If $n^* = -1$, assign $n^* := n'$
         Else if $n^* \neq n'$, return "Unknown"
         Assign $c := c + 1$ and break out of the for loop
   Assign $s := t^* + 1$
Let $L_{t'}$ be the last level of $\mathcal{V}$
If $t' \geq t^* + n^*$, return $n^*$
Else return "Unknown"
```

If s is an isle's root and F is its set of leaves, we have $a(s) \geq \sum_{v \in F} a(v)$, because s may have some descendants in the history tree $\mathcal{H}$ that do not appear in the view $\mathcal{V}$. This is equivalent to $a'(s) \geq \sum_{v \in F} a'(v)$. If equality holds, then the isle is said to be *complete*; in this case, we can easily compute the conditional anonymities of all the internal nodes by adding them up starting from the nodes in F and working our way up to s.

Overview of `ApproxCount`. Our subroutine `ApproxCount` is found in Listing 2. It repeatedly assigns guesses to nodes based on known conditional anonymities, starting from τ and its descendants. Eventually some nodes become heavy, and the criterion in Lemma 10 causes the deepest of them to become counted. In turn, counted nodes eventually form isles; the internal nodes of complete isles are marked as counted, which gives rise to more guessers, and so on. In the end, if a counting cut is created, the algorithm checks whether the conditional anonymities of the leader nodes in the cut add up to ℓ.

Algorithmic details of `ApproxCount`. The algorithm `ApproxCount` uses flags to mark nodes as "guessed" or "counted"; initially, no node is marked. Thanks to these flags, we can check if a node $u \in \mathcal{V}$ is a guesser: let $u_1, u_2, \dots, u_k$ be the children of u that are also in $\mathcal{V}$ (recall that a view does not contain all nodes of a history tree); u is a *guesser* if and only if it is marked as counted, all the u_i's are marked as counted, and $a'(u) = \sum_i a'(u_i)$ (which implies $a(u) = \sum_i a(u_i)$, and thus no children of u are missing from $\mathcal{V}$).

`ApproxCount` will ensure that nodes marked as guessed are well-spread at all times; if a level of $\mathcal{V}$ contains a guessed node, it is said to be *locked*. A level L_t is *guessable* if it is not locked and has a non-counted node v that is guessable, i.e., there is a guesser u in L_{t-1} and the red edge $\{v, u\}$ is present in $\mathcal{V}$ with positive multiplicity.

The algorithm starts by assigning a conditional anonymity $a'(\tau) = x$ to the first leader node $\tau \in L_s$. (If no leader node exists in L_s, it immediately returns the error code -1, Line 6.) It also finds the longest strand P_τ hanging from τ, assigns the same conditional anonymity x to all of its nodes (including the unique child of the last node of P_τ) and marks them as counted (Lines 7–11). Then, as long as there are guessable levels and no counting cut has been found yet, the algorithm keeps assigning guesses to non-counted nodes (Line 12).

When a guess is made on a node v, some nodes in the path from v to its ancestor in L_s may become heavy; if so, let v' be the deepest heavy node. If $g(v')$ is not an integer, the algorithm returns the error code -3 (Line 19). (As we will prove later, this can only happen if $\delta \neq 1$ or some nodes in the strand P_τ have children that are not in the view $\mathcal{V}$.) Otherwise, if $g(v')$ is an integer, the algorithm marks v' as counted (Line 20), in accordance with Lemma 10. Furthermore, if the newly counted node v' is the root or a leaf of a complete isle I, then the conditional anonymities of all the internal nodes of I are determined, and such nodes are marked as counted; this also unlocks their levels if such nodes were marked as guessed (Lines 21–24).

In the end, the algorithm performs a "reality check" and possibly returns an estimate n' of n, as follows. If no counting cut was found, the algorithm returns the error code -2 (Line 25). Otherwise, a counting cut C has been found. The algorithm computes n' (respectively, ℓ') as the sum of the conditional anonymities of all nodes (respectively, all leader nodes) in C. If $\ell' = \ell$, then the algorithm returns n' (Line 32). Otherwise, it returns the error code -1 if $\ell' < \ell$ (Line 30) or the error code -3 if $\ell' > \ell$ (Line 31). In all cases, the algorithm also returns the maximum depth t of a guessed or counted node (excluding τ and its descendants), or s if no such node exists.

Consistency condition. In order for our algorithm to work properly, a condition has to be satisfied whenever a new guess is made. Indeed, note that all of our previous lemmas on guesses rest on the assumption that the conditional anonymities of a guesser and all of its children are known. However, while the node τ has a known conditional anonymity (by definition, $a'(\tau) = x$), the same is not necessarily true of the descendants of τ and all other nodes that are eventually marked as counted by the algorithm. This justifies the following definition.

▶ **Condition 1.** *During the execution of* `ApproxCount`*, if a guess is made on a node v at level $L_{t'}$ of $\mathcal{V}$, then τ has a (unique) descendant $\tau' \in L_{t'}$ and $a(\tau) = a(\tau')$.*

As we will prove next, as long as Condition 1 is satisfied during the execution of `ApproxCount`, all of the nodes between levels L_s and L_t that are marked as counted do have correct guesses (i.e., their guesses coincide with their conditional anonymities). Note that in general there is no guarantee that Condition 1 will be satisfied at any point; it is up to the main counting algorithm that invokes `ApproxCount` to ensure that the condition is satisfied often enough for our computations to be successful.

Correctness. In order to prove the correctness of `ApproxCount`, it is convenient to show that it also maintains some *invariants*, i.e., properties that are always satisfied as long as some conditions are met.

▶ **Lemma 11.** *Assume that $\delta \geq 1$. Then, as long as Condition 1 is satisfied, the following hold.*

(i) *The nodes marked as guessed are always well spread.*
(ii) *Whenever Line 13 is reached, there are no heavy nodes.*
(iii) *Whenever Line 13 is reached, all complete isles are trivial.*
(iv) *The conditional anonymity of any node between L_s and L_t that is marked as counted has been correctly computed.* ◀

Running time. We will now study the running time of `ApproxCount`. We will prove two lemmas that allow us to give an upper bound on the number of rounds it takes for the algorithm to return an output, provided that some conditions are satisfied.

▶ **Lemma 12.** *Assume that $\delta \geq 1$. Then, as long as Condition 1 holds, whenever Line 13 is reached, at most δn levels are locked.* ⌟

We say that a node v of the history tree $\mathcal{H}$ is *missing* from level L_i of the view $\mathcal{V}$ if v is at the level of $\mathcal{H}$ corresponding to L_i but does not appear in $\mathcal{V}$. Clearly, if a level of $\mathcal{V}$ has no missing nodes, all previous levels also have no missing nodes.

▶ **Lemma 13.** *Assume that $\delta \geq 1$. Then, as long as level L_t of $\mathcal{V}$ is not missing any nodes (where t is defined and updated as in* `ApproxCount`*), whenever Line 13 is reached, there are at most $n-2$ levels in the range from L_{s+1} to L_t that lack a guessable non-counted node.* ⌟

Main lemma. The following lemma gives some conditions that guarantee that `ApproxCount` has the expected behavior; it also gives some bounds on the number of rounds it takes for `ApproxCount` to produce an approximation n' of n, as well as a criterion to determine if $n' = n$.

▶ **Lemma 14.** *Let* `ApproxCount`$(\mathcal{V}, s, x, \ell)$ *return (n', t). Assume that τ exists and $x \geq a(\tau)$. Let τ' be the (unique) descendant of τ in $\mathcal{V}$ at level L_t, and let $L_{t'}$ be the last level of $\mathcal{V}$. Then:*

(i) *If $x = a(\tau) = a(\tau')$, then $n' \neq -3$.*
(ii) *If $n' > 0$ and $t' \geq t + n'$ and $a(\tau) = a(\tau')$, then $n' = n$.*
(iii) *If $t' \geq s + (\ell+2)n - 1$, then $s \leq t \leq s + (\ell+1)n - 1$ and $n' \neq -1$. Moreover, if $n' = -2$, then L_t contains a leader node with at least two children in $\mathcal{V}$.* ⌟

Terminating algorithm. We are finally able to state our main terminating algorithm. It assumes that all processes know the number of leaders $\ell \geq 1$ and the dynamic disconnectivity T. Again, this is justified by Proposition 20 and Proposition 2.

▶ **Theorem 15.** *There is an algorithm that computes F_{GC} in all T-union-connected anonymous networks with $\ell \geq 1$ leaders and terminates in at most $(\ell^2 + \ell + 1)Tn$ rounds, assuming that ℓ and T are known to all processes, but assuming no knowledge of n.*

Proof. Due to Proposition 3, since T is known and appears as a factor in the claimed running time, we can assume that $T = 1$ without loss of generality. Also, note that determining n is enough to compute F_{GC}. Indeed, if a process determines n at round t', it can wait until round $\max\{t', 2Tn\}$ and run the algorithm in Theorem 7, which is guaranteed to give the correct output by that time.

In order to determine n assuming that $T = 1$, we let each process run the algorithm in Listing 3 with input $(\mathcal{V}, \ell)$, where $\mathcal{V}$ is the view of the process at the current round t'. We will prove that this algorithm returns a positive integer (as opposed to "Unknown") within $(\ell^2 + \ell + 1)n$ rounds, and the returned number is indeed the correct size of the system n.

Algorithm description. Let b be the number of branches in $\mathcal{V}$ representing leader processes (Line 5). The initial goal of the algorithm is to compute $\ell - b + 1$ approximations of n using the information found in as many disjoint intervals $\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_{\ell-b+1}$ of levels of $\mathcal{V}$ (Lines 6–17).

If there are not enough levels in $\mathcal{V}$ to compute the desired number of approximations, or if the approximations are not all equal, the algorithm returns "Unknown" (Lines 11 and 15).

In order to compute an approximation of n, say in an interval of levels $\mathcal{L}_i$ starting at L_s, the algorithm goes through at most ℓ phases (Lines 8–16). The first phase begins by calling `ApproxCount` with starting level L_s and $x = \ell$, i.e., the maximum possible value for the anonymity of a leader node (Line 9). Specifically, `ApproxCount` chooses a leader node in $\tau \in L_s$ and tries to estimate n using as few levels as possible.

Let (n', t) be the pair of values returned by `ApproxCount`. If $n' = -1$, this is evidence that $\mathcal{V}$ is still missing some relevant nodes, and therefore "Unknown" is immediately returned (Line 11). If $n' = -2$, then a descendant of τ with multiple children in $\mathcal{V}$ was found, say at level L_t, before an approximation of n could be determined. As this is an undesirable event, the algorithm moves $\mathcal{L}_i$ after L_t and tries again to estimate n (Line 12). If $n' = -3$, then x may not be the correct anonymity of the leader node τ (see the description of `ApproxCount`), and therefore the algorithm calls `ApproxCount` again, with the same starting level L_s, but now with $x = \ell - 1$. If $n' = -3$ is returned again, then $x = \ell - 2$ is tried, and so on. After all possible assignments down to $x = 1$ have failed, the algorithm just moves $\mathcal{L}_i$ forward and tries again from $x = \ell$.

As soon as $n' > 0$, this approximation of n is stored in the variable n^*. If it is different from the previous approximations, then "Unknown" is returned (Line 15). Otherwise, the algorithm proceeds with the next approximation in a new interval of levels $\mathcal{L}_{i+1}$, and so on.

Finally, when $\ell - b + 1$ approximations of n (all equal to n^*) have been found, a correctness check is performed: the algorithm takes the last level L_{t^*} visited thus far; if the current round t' satisfies $t' \geq t^* + n^*$, then n^* is accepted as correct; otherwise "Unknown" is returned (Lines 18–20).

Correctness and running time. We will prove that, if the output of Listing 3 is not "Unknown", then it is indeed the number of processes, i.e., $n^* = n$. Since the $\ell - b + 1$ approximations of n have been computed on disjoint intervals of levels, there is at least one such interval, say $\mathcal{L}_j$, where no leader node in the history tree has more than one child (because there can be at most ℓ leader branches). With the notation of Lemma 14, this implies that $a(\tau) = a(\tau')$ whenever `ApproxCount` is called in $\mathcal{L}_j$. Also, since the option $x = \ell$ is tried first, the assumption $x \geq a(\tau)$ of Lemma 14 is initially satisfied. Note that `ApproxCount` cannot return $n' = -1$ or $n' = -2$, or else $\mathcal{L}_j$ would not yield any approximation of n. Moreover, by statement (ii) and by the terminating condition (Line 19), if $n' > 0$ while $x \geq a(\tau)$, then $n^* = n' = n$. On the other hand, by statement (i), we necessarily have $n' > 0$ by the time $x = a(\tau)$.

It remains to prove that Listing 3 actually gives an output other than "Unknown" within the claimed number of rounds; it suffices to show that it does so if it is executed at round $t' = (\ell^2 + \ell + 1)n$. It is known that all nodes in the first $t' - n = \ell(\ell + 1)n$ levels of the history tree are contained in the view $\mathcal{V}$ at round t' (cf. [17, Corollary 4.3]). Also, it is straightforward to prove by induction that the assumption of statement (iii) of Lemma 14 holds every time `ApproxCount` is invoked. Indeed, in any interval of $(\ell + 1)n$ levels, either a branching leader node is found or a new approximation of n is computed. Since there can be at most ℓ leader branches, at least one approximation of n is computed within $\ell(\ell+1)n$ levels. Because all nodes in these levels must appear in $\mathcal{V}$, the condition $a(\tau) = a(\tau')$ of Lemma 14 is satisfied in all intervals $\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_{\ell-b+1}$. Reasoning as in the previous paragraph, we conclude that all such intervals must yield the correct approximation of n. So, every time Line 15 is executed, we have $n^* = n'$, and the algorithm cannot return "Unknown". ◀

5 Negative Results

In this section we list several negative results and counterexamples, some of which are well known (in particular, Proposition 16 is implied by [22, Theorem III.1]). The purpose is to justify all of the assumptions made in Sections 3 and 4. All proofs are found in [19].

5.1 Leaderless Networks

▶ **Proposition 16.** *No function other than the frequency-based multi-aggregate functions can be computed with no leader, even when restricted to simple connected static networks.* ◀

▶ **Proposition 17.** *No algorithm can solve the Average Consensus problem in a T-union-connected leaderless network in less than $2Tn - O(T)$ rounds.* ◀

▶ **Proposition 18.** *No algorithm can solve the leaderless Average Consensus problem with explicit termination if nothing is known about the size of the network, even when restricted to simple connected static networks.* ◀

5.2 Networks with Leaders

▶ **Proposition 19.** *No function other than the multi-aggregate functions can be computed (with or without termination), even when restricted to simple connected static networks with a known number of leaders.* ◀

▶ **Proposition 20.** *No algorithm can compute the Counting function F_C (with or without termination) with no knowledge about ℓ, even when restricted to simple connected static networks with a known and arbitrarily small ratio ℓ/n.* ◀

▶ **Proposition 21.** *For any $\ell \geq 1$, no algorithm can compute the Counting function F_C (with or without termination) in all simple T-union-connected networks with ℓ leaders in less than $T(2n - \ell) - O(T)$ rounds.* ◀

6 Conclusions

We have shown that anonymous processes in disconnected dynamic networks can compute all the multi-aggregate functions and no other functions, provided that the network contains a known number of leaders $\ell \geq 1$. If there are no leaders or the number of leaders is unknown, the class of computable functions reduces to the frequency-based multi-aggregate functions. We have also identified the functions F_{GC} and F_R as the complete problems for each class. Notably, the network's dynamic disconnectivity T does not affect the computability of functions, but only makes computation slower.

We also gave efficient stabilizing and terminating algorithms for computing all the above functions. Some of our algorithms make assumptions on the processes' a-priori knowledge about the network; we proved that such assumptions are actually necessary. All our algorithms have optimal linear running times in terms of T and the size of the network n.

In one case, there is still a small gap in terms of the number of leaders ℓ. Namely, for terminating computation with $\ell \geq 1$ leaders, we have a lower bound of $T(2n - \ell) - O(T)$ rounds (Proposition 21) and an upper bound of $(\ell^2 + \ell + 1)Tn$ rounds (Theorem 15). Although these bounds asymptotically match if the number of leaders ℓ is constant (which is a realistic assumption in most applications), optimizing them with respect to ℓ is left as an open problem.

Observe that our stabilizing algorithms use an unbounded amount of memory, as processes keep adding nodes to their view at every round. This can be avoided if the dynamic disconnectivity T (as well as an upper bound on n, in case of a leaderless network) is known: In this case, processes can run the stabilizing and the terminating version of the relevant algorithm in parallel, and stop adding nodes to their views when the terminating algorithm halts. It is an open problem whether a stabilizing algorithm for F_{GC} or F_R can use a finite amount of memory with no knowledge of T.

Our algorithms require processes to send each other explicit representations of their history trees, which have cubic size in the worst case [17]. It would be interesting to develop algorithms that only send messages of logarithmic size, possibly with a trade-off in terms of running time. We are currently able to do so for leaderless networks and networks with a unique leader, but not for networks with more than one leader [18].

References

1 D. Angluin, J. Aspnes, and D. Eisenstat. Fast Computation by Population Protocols with a Leader. *Distributed Computing*, 21(3):61–75, 2008.

2 J. Aspnes, J. Beauquier, J. Burman, and D. Sohier. Time and Space Optimal Counting in Population Protocols. In *Proceedings of the 20th International Conference on Principles of Distributed Systems (OPODIS '16)*, pages 13:1–13:17, 2016.

3 J. Beauquier, J. Burman, S. Clavière, and D. Sohier. Space-Optimal Counting in Population Protocols. In *Proceedings of the 29th International Symposium on Distributed Computing (DISC '15)*, pages 631–646, 2015.

4 J. Beauquier, J. Burman, and S. Kutten. A Self-stabilizing Transformer for Population Protocols with Covering. *Theoretical Computer Science*, 412(33):4247–4259, 2011.

5 D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, Inc., USA, 1989.

6 P. Boldi and S. Vigna. An Effective Characterization of Computability in Anonymous Networks. In *Proceedings of the 15th International Conference on Distributed Computing (DISC '01)*, pages 33–47, 2001.

7 P. Boldi and S. Vigna. Fibrations of Graphs. *Discrete Mathematics*, 243:21–66, 2002.

8 A. Casteigts, F. Flocchini, B. Mans, and N. Santoro. Shortest, Fastest, and Foremost Broadcast in Dynamic Networks. *International Journal of Foundations of Computer Science*, 26(4):499–522, 2015.

9 A. Casteigts, P. Flocchini, W. Quattrociocchi, and N. Santoro. Time-Varying Graphs and Dynamic Networks. *International Journal of Parallel, Emergent and Distributed Systems*, 27(5):387–408, 2012.

10 J. Chalopin, S. Das, and N. Santoro. Groupings and Pairings in Anonymous Networks. In *Proceedings of the 20th International Conference on Distributed Computing (DISC '06)*, pages 105–119, 2006.

11 J. Chalopin, E. Godard, and Y. Métivier. Local Terminations and Distributed Computability in Anonymous Networks. In *Proceedings of the 22nd International Symposium on Distributed Computing (DISC '08)*, pages 47–62, 2008.

12 J. Chalopin, Y. Métivier, and T. Morsellino. Enumeration and Leader Election in Partially Anonymous and Multi-hop Broadcast Networks. *Fundamenta Informaticae*, 120(1):1–27, 2012.

13 B. Charron-Bost and P. Lambein-Monette. Randomization and Quantization for Average Consensus. In *Proceedings of the 57th IEEE Conference on Decision and Control (CDC '18)*, pages 3716–3721, 2018.

14 B. Charron-Bost and P. Lambein-Monette. Computing Outside the Box: Average Consensus over Dynamic Networks. In *Proceedings of the 1st Symposium on Algorithmic Foundations of Dynamic Networks (SAND '22)*, pages 10:1–10:16, 2022.

15 B. Chazelle. The Total s-Energy of a Multiagent System. *SIAM Journal on Control and Optimization*, 49(4):1680–1706, 2011.

16 G. A. Di Luna, P. Flocchini, T. Izumi, T. Izumi, N. Santoro, and G. Viglietta. Population Protocols with Faulty Interactions: The Impact of a Leader. *Theoretical Computer Science*, 754:35–49, 2019.

17 G. A. Di Luna and G. Viglietta. Computing in Anonymous Dynamic Networks Is Linear. In *Proceedings of the 63rd IEEE Symposium on Foundations of Computer Science (FOCS '22)*, pages 1122–1133, 2022.

18 G. A. Di Luna and G. Viglietta. Brief Announcement: Efficient Computation in Congested Anonymous Dynamic Networks. In *Proceedings of the 42nd ACM Symposium on Principles of Distributed Computing (PODC '23)*, pages 176–179, 2023.

19 G. A. Di Luna and G. Viglietta. Optimal Computation in Leaderless and Multi-Leader Disconnected Anonymous Dynamic Networks. *arXiv:2207.08061 [cs.DC]*, pages 1–37, 2023.

20 L. Faramondi, R. Setola, and G. Oliva. Performance and Robustness of Discrete and Finite Time Average Consensus Algorithms. *International Journal of Systems Science*, 49(12):2704–2724, 2018.

21 P. Fraigniaud, A. Pelc, D. Peleg, and S. Pérennes. Assigning Labels in Unknown Anonymous Networks. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing (PODC '00)*, pages 101–111, 2000.

22 J. M. Hendrickx, A. Olshevsky, and J. N. Tsitsiklis. Distributed Anonymous Discrete Function Computation. *IEEE Transactions on Automatic Control*, 56(10):2276–2289, 2011.

23 D. R. Kowalski and M. A. Mosteiro. Polynomial Counting in Anonymous Dynamic Networks with Applications to Anonymous Dynamic Algebraic Computations. In *Proceedings of the 45th International Colloquium on Automata, Languages, and Programming (ICALP '18)*, pages 156:1–156:14, 2018.

24 D. R. Kowalski and M. A. Mosteiro. Polynomial Anonymous Dynamic Distributed Computing Without a Unique Leader. In *Proceedings of the 46th International Colloquium on Automata, Languages, and Programming (ICALP '19)*, pages 147:1–147:15, 2019.

25 D. R. Kowalski and M. A. Mosteiro. Polynomial Counting in Anonymous Dynamic Networks with Applications to Anonymous Dynamic Algebraic Computations. *Journal of the ACM*, 67(2):11:1–11:17, 2020.

26 D. R. Kowalski and M. A. Mosteiro. Supervised Average Consensus in Anonymous Dynamic Networks. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '21)*, pages 307–317, 2021.

27 D. R. Kowalski and M. A. Mosteiro. Efficient Distributed Computations in Anonymous Dynamic Congested Systems with Opportunistic Connectivity. *arXiv:2202.07167 [cs.DC]*, pages 1–28, 2022.

28 D. R. Kowalski and M. A. Mosteiro. Polynomial Anonymous Dynamic Distributed Computing Without a Unique Leader. *Journal of Computer and System Sciences*, 123:37–63, 2022.

29 F. Kuhn, T. Locher, and R. Oshman. Gradient Clock Synchronization in Dynamic Networks. *Theory of Computing Systems*, 49(4):781–816, 2011.

30 F. Kuhn, N. Lynch, and R. Oshman. Distributed Computation in Dynamic Networks. In *Proceedings of the 42nd ACM Symposium on Theory of Computing (STOC '10)*, pages 513–522, 2010.

31 F. Kuhn, Y. Moses, and R. Oshman. Coordinated Consensus in Dynamic Networks. In *Proceedings of the 30th ACM Symposium on Principles of Distributed Computing (PODC '11)*, pages 1–10, 2011.

32 F. Kuhn and R. Oshman. Dynamic Networks: Models and Algorithms. *SIGACT News*, 42(1):82–96, 2011.

33 O. Michail, I. Chatzigiannakis, and P. G. Spirakis. Naming and Counting in Anonymous Unknown Dynamic Networks. In *Proceedings of the 15th International Symposium on Stabilizing, Safety, and Security of Distributed Systems (SSS '13)*, pages 281–295, 2013.

34 O. Michail and P. G. Spirakis. Elements of the Theory of Dynamic Networks. *Communications of the ACM*, 61(2):72, 2018.

35 A. Nedić, A. Olshevsky, A. E. Ozdaglar, and J. N. Tsitsiklis. On Distributed Averaging Algorithms and Quantization Effects. *IEEE Transactions on Automatic Control*, 54(11):2506–2517, 2009.

36 A. Nedić, A. Olshevsky, and M. G. Rabbat. Network Topology and Communication-Computation Tradeoffs in Decentralized Optimization. *Proceedings of the IEEE*, 106(5):953–976, 2018.

37 R. O'Dell and R. Wattenhofer. Information Dissemination in Highly Dynamic Graphs. In *Proceedings of the 5th Joint Workshop on Foundations of Mobile Computing (DIALM-POMC '05)*, pages 104–110, 2005.

38 A. Olshevsky. Linear Time Average Consensus and Distributed Optimization on Fixed Graphs. *SIAM Journal on Control and Optimization*, 55(6):3990–4014, 2017.

39 A. Olshevsky and J. N. Tsitsiklis. Convergence Speed in Distributed Consensus and Averaging. *SIAM Journal on Control and Optimization*, 48(1):33–55, 2009.

40 A. Olshevsky and J. N. Tsitsiklis. A Lower Bound for Distributed Averaging Algorithms on the Line Graph. *IEEE Transactions on Automatic Control*, 56(11):2694–2698, 2011.

41 N. Sakamoto. Comparison of Initial Conditions for Distributed Algorithms on Anonymous Networks. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing (PODC '99)*, pages 173–179, 1999.

42 J. Seidel, J. Uitto, and R. Wattenhofer. Randomness vs. Time in Anonymous Networks. In *Proceedings of the 29th International Symposium on Distributed Computing (DISC '15)*, pages 263–275, 2015.

43 T. Sharma and M. Bashir. Use of Apps in the COVID-19 Response and the Loss of Privacy Protection. *Nature Medicine*, 26(8):1165–1167, 2020.

44 J. N. Tsitsiklis. *Problems in Decentralized Decision Making and Computation*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1984.

45 M. Yamashita and T. Kameda. Computing on an Anonymous Network. In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing (PODC '88)*, pages 117–130, 1988.

46 M. Yamashita and T. Kameda. Computing on Anonymous Networks. I. Characterizing the Solvable Cases. *IEEE Transactions on Parallel and Distributed Systems*, 7(1):69–89, 1996.

47 Y. Yuan, G.-B. Stan, L. Shi, M. Barahona, and J. Goncalves. Decentralised Minimum-Time Consensus. *Automatica*, 49(5):1227–1235, 2013.

A Impact on Fundamental Problems and State of the Art

As a byproduct of the results mentioned in Section 1.1, we are able to optimally solve two popular fundamental problems: *Generalized Counting* for multi-leader networks (because it is a multi-aggregate function) and *Average Consensus* for leaderless networks (because the mean is a frequency-based multi-aggregate function). As summarized in Table 1 and as discussed below, our results improve upon the state of the art on both problems in terms of (i) running time, (ii) assumptions on the network and the processes' knowledge, and (iii) quality of the solution. Altogether, we settle open problems from ICALP 2019 [24], SPAA 2021 [26], and FOCS 2022 [17]. For a more thorough discussion and a comprehensive survey of related literature, refer to [19].

Average Consensus. This problem has been studied for decades by the distributed control and distributed computing communities [5, 13, 14, 15, 26, 35, 38, 40, 44, 47]. In the following, we argue that our results directly improve upon the current state of the art on this problem. A more detailed discussion can be found in the surveys [20, 36, 39] and in [19].

A convergent algorithm with a running time of $O\left(Tn^3 \log(1/\epsilon)\right)$ is given in [35]. The algorithm works in T-union-connected networks with no knowledge of T, but it rests on the assumption that the degree of each process in the network has a known upper bound. Assuming an always connected network, [14] gives an algorithm that converges in $O\left(n^4 \log(n/\epsilon)\right)$ rounds. We remark that both algorithms are only ϵ-convergent; therefore, not only does our stabilizing algorithm improve upon their running times, but it solves a more difficult problem under weaker assumptions.

The algorithm in [13] stabilizes to the actual average in a linear number of rounds, but it is a randomized Monte Carlo algorithm and requires the network to be connected at each round. In contrast, our linear-time stabilizing algorithm is deterministic and works in disconnected networks.

As for terminating algorithms, the one in [26] terminates in $O\left(n^5 \log^3(n)/\ell\right)$ rounds assuming the presence of a known number ℓ of leaders and an always connected network. Since the number of leaders is known, our terminating algorithm for Generalized Counting also solves Average Consensus with a running time that improves upon [26] and does not require the network to be connected. We remark that our algorithm terminates in linear time when ℓ is constant.

Generalized Counting. Our results on this problem are direct generalizations of [17] to the case of multiple leaders and disconnected networks. The best previous counting algorithm with multiple known leaders is the one in [28], which terminates in $O\left(n^4 \log^3(n)/\ell\right)$ rounds and assumes the network to be connected at each round. In the same setting, our stabilizing and terminating algorithms have running times of $2n$ rounds and $(\ell^2 + \ell + 1)n$ rounds, respectively.

The only other result for disconnected networks is the recent preprint [27], which gives an algorithm that terminates in $\widetilde{O}\left(n^{2T+3}/\ell\right)$ rounds using $O(\log n)$-sized messages. Our terminating algorithm has a linear dependence on both n and T, which is an exponential improvement upon the running time of [27], but it requires polynomial-sized messages.

Fast Coloring Despite Congested Relays

Maxime Flin
Reykjavik University, Iceland

Magnús M. Halldórsson
Reykjavik University, Iceland

Alexandre Nolin
CISPA Helmholtz Center for Information Security, Saarbrücken, Germany

Abstract

We provide a $O(\log^6 \log n)$-round randomized algorithm for distance-2 coloring in CONGEST with $\Delta^2 + 1$ colors. For $\Delta \gg \text{poly} \log n$, this improves exponentially on the $O(\log \Delta + \text{poly} \log \log n)$ algorithm of [Halldórsson, Kuhn, Maus, Nolin, DISC'20].

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Mathematics of computing → Graph coloring

Keywords and phrases CONGEST model, distributed graph coloring, power graphs

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.19

Related Version *Full Version*: `https://arxiv.org/abs/2308.01359` [17]

Funding *Maxime Flin*: Funded by the Icelandic Research Fund (grant 2310015).
Magnús M. Halldórsson: Partially supported by the Icelandic Research Fund (grant 217965).

1 Introduction

In the LOCAL model of distributed computing, we are given a communication network in the form of an n-node graph $G = (V, E)$, where each node has a unique $O(\log n)$-bit identifier. Time is divided into discrete intervals called rounds, during which nodes send/receive one message to/from each of their neighbors in G. In the CONGEST model, each message additionally is restricted to $O(\log n)$ bits.

Coloring problems are amongst the most intensively studied problems in the distributed graph literature for they capture the main challenges of symmetry breaking and resolving conflicts (see, e.g., [4]). We consider the *distance-2* $\Delta^2 + 1$-coloring problem in CONGEST, where Δ is the maximum degree of G. The task is to assign each node a color from $\{1, 2, \dots, \Delta^2 + 1\}$ that is different from nodes within distance 2 in G. Namely, we want to color the square graph G^2 while communicating on G with $O(\log n)$-bit messages.

The distance-2 coloring problem is particularly interesting as a petri dish for examining the impact of bandwidth constraints. When we seek a coloring of G, each node can directly communicate with all nodes that it conflicts with, which are precisely its neighbors. In G^2, a node can conflict with Δ^2 other nodes, but only communicate directly with Δ of them. Thus, the bandwidth used by a node is at most $\Delta \log n$ bits, both for incoming and outgoing messages, which can be much smaller than the number of neighbors in G^2. In fact, it is altogether non-trivial to obtain even a $\text{poly}(\log n)$-round algorithm for distance-2 $\Delta^2 + 1$-coloring, which was only achieved in 2020 [27].

Distance-2 coloring is also interesting in its own right. In particular, it arises naturally when assigning frequencies to antennas in wireless networks. More generally, symmetry breaking on power graphs appears naturally in numerous settings [35, 6, 20, 21, 11, 13]. See, e.g., [38, Section 1.2] for a recent treatment.

Finally, the distance-2 coloring problem is the only explicit problem studied when the *conflict graph* H to be colored is different from the *communication graph* G. In the LOCAL model, this distinction is of little concern, as the square graph can be simulated within G with an overhead of factor 2 in the round complexity. This is what motivates the usual assumption that $H = G$. In CONGEST however, bandwidth constraints preclude such local reductions. This is a major challenge toward understanding the complexity landscape in CONGEST. In fact, such local transformations are ubiquitous in the distributed graph literature. Notable examples include reductions to Maximal Independent Set [4, Section 3.9], coloring algorithms based on the Lovász Local Lemma [41, 10, 8], or subroutines working with cluster graphs [24, 40, 13].

Our Contributions. We provide a $\operatorname{poly}\log\log n$-round randomized algorithm to find a distance-2 coloring of G. Our algorithm uses $\Delta^2 + 1$ colors, which is a natural analog to $\Delta + 1$ at distance-1.

▶ **Theorem 1.** *There is a randomized algorithm for distance-2 coloring any n-node graph G with maximum degree Δ, using $\Delta^2 + 1$ colors, and running in $O(\log^6 \log n)$ rounds of* CONGEST*.*

This is an exponential improvement over the previous best known bound $O(\log n)$ [28], as a function of n alone. Interestingly, for more general power graphs G^k with $k \geq 3$, it is provably hard to verify an arbitrary coloring [18]. Thus, any $\operatorname{poly}\log\log n$ algorithm coloring G^k when $k \geq 3$ and $\Delta \gg \operatorname{poly}\log\log n$ would need a different approach.

Theorem 1 requires non-constructive pseudorandom compression techniques, so can be viewed as either existential or requiring exponential local computation. However, we give an explicit and efficient algorithm that achieves such a coloring with $O(\log^2 n)$ bandwidth. We emphasize that even with $O(\log^2 n)$ bandwidth, it is not clear that fast coloring algorithms can be implemented at distance-2. Our $O(\log^2 n)$-bandwidth algorithm preserves the intuition behind our techniques. In fact, reducing the bandwidth to $O(\log n)$ is a technical issue that is almost entirely solved by previous work [32].

1.1 Related Work

Coloring has been extensively studied in the distributed literature [44, 6, 4, 33, 9, 31], and it was the topic of the paper of Linial [36] that defined the LOCAL model. The best round complexity of randomized $(\Delta + 1)$-coloring in LOCAL (as a function of n alone) progressed from $O(\log n)$ in the 80's [37, 2, 34], through $O(\log^3 \log n)$ [6, 33, 9], to the very recent $\widetilde{O}(\log^2 \log n)$ [22]. These algorithms made heavy use of both the large bandwidth and the multiple-message transmission feature of the LOCAL model.

In CONGEST, Halldórsson, Kuhn, Maus, and Tonoyan [29] gave a $O(\log^5 \log n)$-round CONGEST algorithm, later improved to $O(\log^3 \log n)$ in [32, 25]. Very recently, Flin, Ghaffari, Halldórsson, Kuhn, and Nolin [15] provided a $O(\log^3 \log n)$-round algorithm in *broadcast* CONGEST, in which nodes are restricted to broadcast one $O(\log n)$-bit message per round. While these algorithms drastically reduced the bandwidth requirements compared to their earlier LOCAL and CONGEST counterparts, they still use more bandwidth than what distance-2 coloring allows. Indeed, at distance-2 a node cannot receive a distinct message from each neighbor.

Recent years have seen several results for problems on power graphs in CONGEST [26, 27, 28, 38, 7]. Ghaffari and Portmann [26] gave the first sublogarithmic network decomposition algorithm with a mild dependency on the size of identifiers. Keeping a mild dependency on the size of identifiers is crucial in CONGEST as a common technique, called *shattering*, is to reduce the problem to small $\operatorname{poly}\log n$-size instances on which we run a deterministic algorithm, typically a network decomposition algorithm. While the instance size decreases exponentially, identifiers remain of size $O(\log n)$ bit. Hence, deterministic algorithms with linear dependency on the size of identifiers, such as [43], yield no sub-logarithmic algorithms. The later $O(\log^5 n)$ CONGEST algorithm by [24] with mild dependency on the ID space was extended by [40] to work on power graphs with exponentially large IDs in time $O(\log^7 n)$. Very recently, [38] gave a $O(k^2 \log\Delta \log\log n + k^4 \log^5 \log n)$ randomized CONGEST algorithm to compute a maximal independent set in G^k. Along the way, [38] extended the faster $\widetilde{O}(\log^3 n)$ network decomposition of [23] to power graphs in CONGEST with a mild dependency on the ID space.

When $(1+\varepsilon)\Delta^2$ colors are available, the distance-2 coloring problem is much easier and is known to be solvable in $O(\log^4 \log n)$ rounds [30]. The first $\operatorname{poly}(\log n)$-round CONGEST algorithm for distance-2 (Δ^2+1)-coloring was given in [27], while a $O(\log\Delta)+\operatorname{poly}(\log\log n)$-round algorithm was given in [28]. The original publication of this last result had a higher dependence in n, later reduced by improved network decomposition results of [23, 38] and a faster deterministic algorithm of [25]. We state this for later use, and give more details in the full version [17, Appendix F]:

▶ **Proposition 2** ([28, Lemma 3.12+3.15] + [38, Appendix A] + [25])**.** *Let H be a subgraph of G^2 where G is the communication network, and suppose $\Delta(H) \le \operatorname{poly}\log n$. Suppose each node v, of degree $d_H(v)$ in H, knows a list $L(v)$ of $d_H(v)+1$ colors from some color space $|\mathcal{U}| \le \operatorname{poly}(n)$. There is a randomized algorithm coloring H in $O(\log^5 \log n)$ rounds of* CONGEST *such that each node receives a color from its list.*

The best bound known for a deterministic CONGEST algorithm using Δ^2+1 colors is $O(\Delta^2 + \log^* n)$ rounds [27]. Very recently, [7] gave a handful of deterministic coloring algorithms on power graphs, including a $O(\Delta^4)$-coloring algorithm in $O(\log\Delta \cdot \log^* n)$ rounds (which is an adaptation of Linial's algorithm) and a $O(\Delta^2)$-coloring algorithm in $O(\Delta \log\Delta + \log\Delta \cdot \log^* n)$ rounds (which is an adaptation of the $O(\sqrt{\Delta}\operatorname{poly}\log\Delta + \log^* n)$ algorithm of [19, 5, 39]) for the distance-2 setting.

Open Problems. Along the way we introduce various tools whose range of application extends to more general coloring problems. In particular, almost all steps of our algorithm work if each $v \in V$ uses colors $\{1, 2, \dots, \widetilde{d}(v)+1\}$, for $\widetilde{d}(v)$ a locally computable upper bound on degrees. It would be interesting to know if $\widetilde{d}(v)+1$-coloring could be solved. The more difficult list variants of this problem, where nodes must adopt colors from lists of size Δ^2+1 or $\widetilde{d}(v)+1$, are also open. Key aspects of our approach fail for list coloring and, in fact, it is not even known if $\Delta+1$-list-coloring of G is achievable in $\operatorname{poly}\log\log n$ rounds of broadcast CONGEST. It would also be interesting to push the complexity of Δ^2+1-coloring of G^2 down to $O(\log^* n)$ when $\Delta \ge \operatorname{poly}\log n$, to fully match the state of the art at distance one. While we conjecture that minor modifications to our algorithm[1] might be able to reduce its complexity by one or more $\log\log n$ factors, achieving $O(\log^* n)$ would require a new approach, as many steps of our algorithm use $\Omega(\log\log\Delta)$ rounds.

[1] Such as reducing the nodes' uncolored degrees to $O(\log n/\log\log n)$ instead of $O(\log n)$, or slightly reducing the number of layers produced by one of the subroutines (SliceColor).

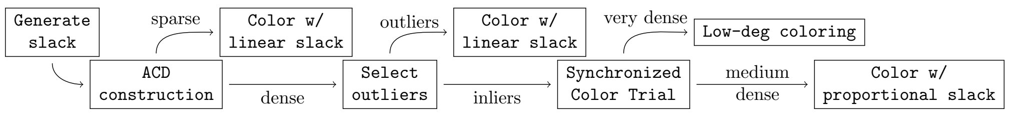


Figure 1 The structure of ultrafast coloring algorithms.

1.2 Our Techniques in a Nutshell

In this section, we highlight the main challenges and sketch the main ideas from our work. Precise definitions are in Section 2 and a more detailed but still high-level overview of our algorithm can be found in Section 3.

Fast Distributed Coloring. All sublogarithmic distributed coloring algorithms [33, 9, 29, 31, 32, 15] follow the overall structure displayed in Figure 1. The key concept is the one of *slack*: the slack of a node is the difference between the number of colors available to that node (i.e., not used by a colored neighbor) and its number of uncolored neighbors (see Definition 3). Nodes with slack proportional to their uncolored degree can be colored fast. The algorithm uses a combination of creating excess colors (by coloring two neighbors with the same color) and reducing the uncolored degree in order for all nodes to get a slack linear in their uncolored degree.

We first generate *slack* by a single-round randomized color trial. We next partition the nodes into the sparse nodes and dense clusters (called *almost-cliques*). Among the dense clusters, we separate a fraction of the nodes as *outliers*. Both the sparse nodes and the outliers can be colored fast using the *linear* slack available to them. The remaining *inliers* then go through a *synchronized color trial* (SCT), where the nodes are assigned a near-uniform random color which avoids color clashes between nodes in the same cluster. The remaining nodes now should have slack proportional to their number of uncolored neighbors. The ultra-dense clusters need a special treatment, but they induce a low-degree graph. The above structure is necessary for high-degree graphs, while for low-degree graphs one can afford less structured methods.

At the outset, several parts of this schema already exist for distance-2 coloring. In particular, generating slack is trivial, a $\text{poly}(\log\log n)$-round algorithm for coloring $\text{poly}(\log n)$-degree graphs is known from [28], and coloring with slack $\Omega(\Delta^2)$ follows from [30]. Almost-clique decompositions (ACD) have been well studied and need only a minor tweak here. We use a particularly simple form of SCT, introduced for the streaming and broadcast CONGEST settings [16, 15]: permute the colors of the *clique palette* – the set of colors not used within the almost-clique – and distribute them uniformly at random among the nodes. We produce the clique palette by giving the nodes data structure access for looking up their assigned color.

Challenges. The biggest challenge in deriving efficient algorithms for the distance-2 setting is that there are no efficient methods known (and possibly not existing) to compute (or approximate) basic quantities like the distance-2 degree of a node. One can easily count the number of 2-paths from a given node, but not how many distinct endpoints they have. This seriously complicates the porting of all previous methods from the distance-1 setting, as we shall see.

A related issue is that a node cannot keep track of all the colors used by its distance-2 neighbors, since it has Δ^2 of them but only bandwidth $O(\Delta \log n)$ bits. Hence, it cannot maintain its true palette (the set of available colors), which means that the standard method of coloring with proportional slack [44, 9] (that can be achieved in $O(\log^* n)$ rounds in distance-1) is not available.

Using Multiple Sources of Slack. We use slack from four sources in our analysis. The (usual) initial slack generation step gives the dense nodes slack proportional to their external degree – their degree to outside of their almost-clique. The method of *colorful matching* [3] provides slack proportional to the average anti-degree of the cluster, where anti-degree counts non-neighbors in one's almost-clique. And finally we get two types of slack for free: the discrepancy between the node's pseudodegree and its true degree on one hand, and the difference between Δ^2 and the pseudodegree on the other hand, where pseudodegrees are easy-to-compute estimates of distance-2 degrees. Only by combining all four sources can we ensure that the final step of coloring with proportional slack can be achieved fast.

Selecting Outliers. The outliers are nodes with exceptionally high degree parameters, either high *external degree* (to the outside of the almost-clique) or *anti-degree* (non-neighbors within the cluster). As we cannot estimate their true values, we work with *pseudodegrees*: the number of 2-paths to external neighbors, or how many additional 2-paths are necessary to reach all anti-neighbors. The selection of outliers is crucial for the success of the last step of the algorithm, where we need to ensure that nodes have true slack proportional to the number of uncolored neighbors. To select outliers, we use a sophisticated filtering technique, giving us bounds in terms of certain related parameters, that then can be linked to the slack that the nodes obtain.

Coloring Fast with Slack. With the right choice of inliers and suitable analysis of SCT, we argue that remaining uncolored nodes have slack proportional to their uncolored degree. We provide a new procedure to color these nodes, extending a method from the first fast CONGEST algorithm [29]. It needs to be adapted to biased sampling and to handle nodes with different ranges of slack. It outputs a series of low-degree graphs, which are then colored by the method of [28].

1.3 Organization of the Paper

After introducing some definitions and results from previous work in Section 2, we give a detailed overview of the full algorithm in Section 3. In Section 4, we go over the technical details involving the coloring of dense nodes, assuming a $O(\log^2 n)$ bandwidth. Various technical parts, as well as details on reducing bandwidth to $O(\log n)$, are in the full version [17].

2 Preliminaries & Definitions

Distributed Graphs. For any integer $k \geq 1$, let $[k]$ represent the set $\{1, 2, \ldots, k\}$. We denote by $G = (V, E)$ the communication network, $n = |V|$ its number of nodes, and Δ its maximum degree. The square graph G^2 has vertices V and edges between pairs $u, v \in V$ if $\text{dist}_G(u, v) \leq 2$. For a node $v \in V$, we denote its unique identifier by $\mathsf{ID}(v)$. For a graph $H = (V_H, E_H)$, the neighborhood in H of v is $N_H(v) = \{u \in V_H : uv \in E_H\}$. A subgraph $K = (V_K, E_K)$ of $H = (V_H, E_H)$ with $V_K \subseteq V_H$ is an *induced* subgraph if

$E_K = \{uv \in E_H : u, v \in V_K\}$, i.e., it contains all edges of E_H between nodes of V_K. We call *anti-edge* in H a pair $u, v \in V_H$ such that $uv \notin E_H$, i.e., an edge missing from H (or, equivalently, in the complement of H).

The degree of v in H is $d_H(v) = |N_H(v)|$, and we shall denote by $N^2(v) = N_{G^2}(v)$ the distance-2 neighbors of v, and the *distance-2 degree* of v by $d(v) = |N^2(v)|$. We also drop the subscript for distance-1 neighbors and write $N(v)$ for $N_G(v)$.

Distributed Coloring. A *partial* c-coloring $\mathcal{C}$ is a function mapping vertices V to colors $[c] \cup \{\perp\}$ such that if $uv \in E$, either $\mathcal{C}(u) \neq \mathcal{C}(v)$ or $\perp \in \{\mathcal{C}(u), \mathcal{C}(v)\}$. The coloring is complete if $\mathcal{C}(v) \neq \perp$ for all $v \in V$ (i.e., all nodes have a color). A deg +1-list-coloring instance is an input graph $H = (V_H, E_H)$ where each node has a list $L(v)$ of $d_H(v) + 1$ colors from some color space $\mathcal{U}$. A valid deg +1-list-coloring is a proper coloring $\mathcal{C} : V_H \to \mathcal{U}$ such that $\mathcal{C}(v) \in L(v)$ for each $v \in V_H$.

Our algorithm computes a monotone sequence of partial colorings until all nodes are colored. In particular, once a node *adopts* a color, it never changes it. The palette of v with respect to the current partial coloring $\mathcal{C}$ is $\Psi(v) \stackrel{\text{def}}{=} [\Delta^2 + 1] \setminus \mathcal{C}(N^2(v))$, i.e., the set of colors that are not used by distance-2 neighbors. For a set $S \subseteq V$, we shall denote the uncolored vertices of S by $S^\circ \stackrel{\text{def}}{=} \{v \in S : \mathcal{C}(v) = \perp\}$ and, reciprocally, the colored vertices of S by $S^\bullet \stackrel{\text{def}}{=} S \setminus S^\circ$. We shall denote the uncolored (distance-2) degree with respect to $\mathcal{C}$ by $d^\circ(v) \stackrel{\text{def}}{=} |N^\circ_{G^2}(v)|$.

2.1 Slack Generation

A key notion to all fast randomized coloring algorithm is the one of slack. It captures the number of excess colors: a node with slack s will always have s available colors, regardless of the colors tried concurrently by neighbors. For our problem, the slack is more simply captured by the following definition.

▶ **Definition 3** (Slack). *Let H be an induced subgraph of G^2. The slack of v in H (with respect to the current coloring of G^2) is*

$$s_H(v) \stackrel{\text{def}}{=} |\Psi(v)| - d^\circ_H(v) \ .$$

There are three ways a node can receive slack: if it has a small degree originally, if two neighbors adopt the same color, or if an uncolored neighbor is inactive (does not belong to H). We consider the first two types of slack *permanent* because a node never increases its degree, and nodes never change their adopted color. On the other hand, the last type of slack is *temporary*: if some inactive neighbors become active, the node loses the slack which was provided by those neighbors.

The sparsity of a node counts the number of missing edges in its neighborhood. We stress that, contrary to previous work in $\Delta + 1$-coloring [9, 29, 15], we use the *local sparsity* – defined in terms of the node's degree $d(v)$ – as opposed to the global sparsity, instead defined in term of Δ. This is to separate the contribution to slack of same-colored neighbors from the *degree slack*, $\Delta^2 - d(v)$. While global sparsity measures both, local sparsity focuses on the former.

▶ **Definition 4** (Local Sparsity, [1, 31]). *The sparsity of v (in the square graph G^2) is*

$$\zeta_v \stackrel{\text{def}}{=} \frac{1}{d(v)}\left(\binom{d(v)}{2} - |E(N^2(v))|\right) .$$

A node v is ζ-sparse if $\zeta_v \geq \zeta$; if $\zeta_v \leq \zeta$ it is ζ-dense.

For a node v, observe that each time that both endpoints of a missing edge in $N^2(v)$ are colored the same, the node v gains slack as its uncolored degree decreases by 2 while its palette loses only 1 color. Therefore, when a node has many missing edges in its neighborhood, it has the potential to gain a lot of slack [42, 12]. This potential for slack is turned into permanent slack by the following simple algorithm (GenerateSlack): each node flips a random coin (possibly with constant bias); each node whose coin flip turned heads picks a color at random and *tries* it, i.e., colors itself with it if none of its neighbors is also trying it. As we state the result with local sparsity (which is in terms of $d(v)$) while nodes try colors in $[\Delta^2+1]$, the next statement has a $d(v)/\Delta^2$ factor compared to previously published versions.

▶ **Proposition 5** (Slack Generation, [42, 12, 29])**.** *There exists a (small) universal constant $\gamma_{\mathsf{slack}} > 0$ such that after* GenerateSlack*, w.p. $exp(-\Omega(\zeta_v \cdot d(v)/\Delta^2))$, node v receives slack $\gamma_{\mathsf{slack}} \cdot \zeta_v \cdot \frac{d(v)}{\Delta^2}$.*

2.2 Sparse-Dense Decomposition

All recent fast randomized distributed coloring algorithms [33, 9, 29, 31, 14, 15] decompose the graph into a set of sparse nodes and several dense clusters. Such a decomposition was first introduced by [42].

▶ **Definition 6.** *For $\varepsilon \in (0, 1/3)$, a distance-2 ε-almost-clique decomposition (ACD) is a partition of $V(G)$ in sets $V_{\mathsf{sparse}}, K_1, \dots, K_k$ such that*

1. *nodes in V_{sparse} either are $\Omega(\varepsilon^2\Delta^2)$-sparse in G^2 or have degree $d(v) \le \Delta^2 - \Omega(\varepsilon^2\Delta^2)$,*
2. *for all $i \in [k]$, sets K_i are called* almost-cliques*, and verify*
   - **a.** *$|K_i| \le (1+\varepsilon)\Delta^2$,*
   - **b.** *for each $v \in K_i$, $|N^2(v) \cap K_i| \ge (1-\varepsilon)\Delta^2$.*

There are several ways to compute this decomposition in CONGEST [28, 29, 32, 16]. We refer the reader to the version of [32, Section 4.2]. The existing distance-2 algorithm of [28] uses $O(\log \Delta)$ rounds and the CONGEST algorithms by [29] require too much bandwidth at distance-2. We mention that [16] implements [32] without representative hash functions and that it can be done here as well. We refer the reader to the full version, [17, Section D], for more details.

▶ **Lemma 7** (Adaptation of [16, Section B.1])**.** *There exists a* CONGEST *randomized algorithm partitioning the graph into $V_{\mathsf{sparse}}, K_1, \dots, K_k$ for some integer $k \ge 0$ such as described in Definition 6. It runs in $O(\varepsilon^{-4})$ rounds.*

▶ **Definition 8** (External and Anti-Degrees)**.** *For a node $v \in K$ and some almost-clique K, we call $e_v = |N^2(v) \setminus K|$ its* external degree *and $a_v = |K \setminus N^2(v)|$ its anti-degree. We shall denote by $\overline{e}_K = \sum_{v\in C} e_v/|K|$ the average external degree and $\overline{a}_K = \sum_{v\in K} a_v/|K|$ the average anti-degree.*

It was first observed by [29] that sparsity bounds external and anti-degrees.

▶ **Lemma 9** ([29, Lemmas 6.2])**.** *There exists two constants $C_{\mathsf{ext}} = C_{\mathsf{ext}}(\varepsilon) > 0$ and $C_{\mathsf{anti}} = C_{\mathsf{anti}}(\varepsilon) > 0$ such that for all $v \in K$, the bounds $e_v \le C_{\mathsf{ext}}\zeta_v$ and $a_v \le C_{\mathsf{anti}}\zeta_v$ holds.*

2.3 Pseudo-degrees

Bandwidth constraints, such as that of the CONGEST model, can severely restrict nodes in their ability to learn information about their neighborhood in a power graph of G. This includes a node's palette (which colors are not yet used by its neighbors in the power graph) but also its degree and related quantities. This motivates the use of similar, but readily computable quantities.

▶ **Definition 10** (Distance-2 Pseudo-Degrees). *In the distance-2 setting, for any node $v \in V$, let its* pseudo-degree $\widetilde{d}(v)$ *and its* uncolored pseudo-degree $\widetilde{d}^{\circ}(v)$ *be*

$$\widetilde{d}(v) \overset{\text{def}}{=} \sum_{u \in N_G(v)} |N_G(u)| \quad \text{and} \quad \widetilde{d}^{\circ}(v) \overset{\text{def}}{=} |N^{\circ}_G(v)| + \sum_{u \in N_G(v)} |N^{\circ}_{G \setminus \{v\}}(u)| \ . \tag{1}$$

For a dense node $v \in K$, its pseudo-external degree $\widetilde{e}_v$ *and its* pseudo-anti degree $\widetilde{a}_v$ *are*

$$\widetilde{e}_v \overset{\text{def}}{=} \sum_{u \in N_G(v)} |N_G(u) \setminus K| \quad \text{and} \quad \widetilde{a}_v \overset{\text{def}}{=} |K| - \sum_{u \in N_G(v)} |N_G(u) \cap K| \ . \tag{2}$$

Note that pseudo-degree and pseudo-external degree are *overestimates* of a node's actual degree and external degree, while pseudo-anti degree is an *underestimate* of a dense node's actual anti-degree. The estimates are accurate for nodes with a tree-like 2-hop neighborhood.

For dense nodes, we also introduce notation for the deviations between the pseudo-degrees and actual G^2-degrees. Such deviations result in slack, which we exploit later in the paper.

$$\theta^{\mathsf{ext}}_v \overset{\text{def}}{=} \widetilde{e}_v - e_v \ , \quad \theta^{\mathsf{anti}}_v \overset{\text{def}}{=} a_v - \widetilde{a}_v \ , \quad \text{and} \quad \theta_v \overset{\text{def}}{=} \theta^{\mathsf{ext}}_v + \theta^{\mathsf{anti}}_v = \widetilde{d}(v) - d(v) \ .$$

We also write $\theta^{\mathsf{ext}}_K = \sum_{v \in K} \theta^{\mathsf{ext}}_v / |K|$ for the average value within a clique.

Pseudo-degrees partially allow nodes to estimate their *degree slack*, the number of colors that v is guaranteed to always have available due to the palette being larger than its degree. Intuitively, the deviations θ^{ext}_v and θ^{anti}_v capture the part of its degree slack that a dense node v does not know about.

$$\underbrace{\Delta^2 + 1 - d(v)}_{\text{degree slack}} = \underbrace{\Delta^2 + 1 - \widetilde{d}(v)}_{\text{known to } v} + \underbrace{\theta^{\mathsf{ext}}_v + \theta^{\mathsf{anti}}_v}_{\text{unknown to } v} \tag{3}$$

3 Detailed Overview of the Full Algorithm

We now give a streamlined overview of our algorithm and describe with some details the technical ideas behind it. See Algorithm 1 for a high-level description of its steps. Since there exists a $O(\log^5 \log n)$-round algorithm when $\Delta \leq \operatorname{poly}\log n$ (Proposition 2), we assume $\Delta \geq \Omega(\log^{3.5} n)$. Henceforth, we assume we are given the almost-clique decomposition $V_{\mathsf{sparse}}, K_1, \ldots, K_k$ (Lemma 7).

Coloring Sparse Nodes (Steps 2 & 3). The coloring of sparse nodes was already handled in [30]. After GenerateSlack, all sparse nodes have slack proportional to Δ^2 (Proposition 5). In particular, their palettes always represent a constant fraction of the color space $[\Delta^2 + 1]$. This allows them to sample colors in their palette efficiently without learning most of their distance-2 neighbors' colors. The algorithm is summarized by the following proposition:

▶ **Proposition 11** (Coloring Nodes with Slack Linear in Δ^2, [30]). *Suppose $\Delta \geq \Omega(\log^{3.5} n)$. Let H be an induced subgraph of G^2 for which all nodes have slack $\gamma \cdot \Delta^2$ for some universal constant $\gamma > 0$ known to all nodes. There exists an algorithm coloring all nodes of H in $O(\log^* n)$ rounds.*

Algorithm 1 High-Level Algorithm.

```
Input  : Graph $G$ with $\Delta \geq \Omega(\log^{3.5} n)$
Output : A distance-2 coloring $\mathcal{C}$ of $G$
1  $V_{\mathsf{sparse}}, K_1, \dots, K_k$ = ComputeACD($\varepsilon$)                      (Section 2.2)
2  GenerateSlack                                                  (Proposition 5)
3  ColoringSparseNodes                                            (Proposition 11)
4  Matching                                                       ([17, Appendix C])
5  ComputeOutliers                                                ([17, Section 6])
6  ColorOutliers                                                  (Proposition 11)
7  SynchColorTrial                                                (Section 4.2)
8  $L_1, \dots, L_\ell \leftarrow$ SliceColor, for some $\ell = O(\log\log n)$   (Section 4.3 and [17, Section 5])
9  foreach $i \in [\ell]$ do
10     LearnPalette                                               (Section 4.4)
11     ColorSmallDegree($L_i$)                                    (Proposition 2)
```

Reducing Degrees with Slack. Since coloring sparse nodes is already known, from now on, we focus our attention on dense nodes. Reducing coloring problems to low-degree instances that one then solves with an algorithm that benefits from the low degree is a common scheme in randomized algorithms for distributed coloring [6, 9]. In particular, when nodes have slack linear in their degree, it was observed by [44, 12, 9] that if nodes try *multiple colors from their palette*, degrees decrease exponentially fast, resulting in a $O(\log^* n)$-round algorithm in LOCAL. This observation motivates the structure of all ultrafast coloring algorithms: 1) generate $\Omega(e_v)$ slack with GenerateSlack, 2) reduce degrees to $O(e_v)$ with SynchColorTrial, and 3) complete the coloring with slack. Unfortunately, this approach is not feasible for us because it requires too much bandwidth. As a result, we do something intermediate that takes advantage of slack but only tries a single color at a time to accommodate our bandwidth limitations. In $O(\log\log n)$ rounds, our method creates $O(\log\log n)$ instances of the maximum degree $O(\log n)$.

Another key technical detail of these methods is that nodes try colors *from their palettes*. At distance-2, perfect sampling in one's palette is not feasible for nodes do not have sufficient bandwidth. We show that they can nevertheless sample colors from a good enough approximation of their palette, in the sense that it preserves the slack. Our involved sampling process requires our degree reduction algorithm to work with weaker guarantees than previous work [29, 31].

▶ **Lemma 12** (Slice Color). *Let $C, \alpha, \kappa > 0$ be some universal constants. Suppose each node knows an upper bound $b(v) \geq d^\circ(v)$ on its uncolored degree. Suppose that for all nodes with $b(v) \geq C \log n$, and a value $s(v) \geq \alpha \cdot b(v)$, there exists an algorithm that samples a color $\mathsf{C}_v \in \Psi(v) \cup \{\bot\}$ (where $\bot$ represents failure) with the following properties:*

$$\Pr(\mathsf{C}_v = \bot) \leq 1/\operatorname{poly}(n) \ , \tag{4}$$

$$\Pr(\mathsf{C}_v = c \mid \mathsf{C}_v \neq \bot) \leq \frac{\kappa}{d^\circ(v) + s(v)} \ . \tag{5}$$

Then, there is a $O(\log\log\Delta + \kappa \cdot \log(\kappa/\alpha))$-rounds algorithm extending the current partial coloring so that uncolored vertices are partitioned into $\ell = O(\log\log\Delta)$ layers $L_1, \dots, L_\ell$ such that each uncolored node knows to which layer it belongs and each $G[L_i]$ has uncolored degree $O(\log n)$.

Coloring Dense Nodes. We assume the sparse nodes are colored (Step 3) and focus on the dense nodes (Steps 4 to 11). Dense nodes receive slack proportional to their external degree (Step 2, Proposition 5 and Lemma 9) in all but the densest almost-cliques.

Steps 4, 5 & 6: Setting up (Section 4.1). We begin by two pre-processing steps to ensure uncolored nodes have useful properties further in the algorithm. Computing a colorful matching (Step 4, Proposition 17) creates $\Theta(\overline{a}_K)$ slack in the clique palette $\Psi(K)$. This is a crucial step to ensure we can approximate nodes palettes (see Step 8). We then compute a (small) fraction $O_K \subseteq K$ of atypical nodes called *outliers* (Step 5, Lemma 14). Outliers have $\Omega(|K|)$ slack from their inactive inlier neighbors, and can thus be colored in $O(\log^* n)$ rounds (Step 6, Proposition 11). *Inliers* $I_K \overset{\text{def}}{=} K \setminus O_K$ verify $\widetilde{e}_v \leq O(\overline{e}_K + \theta_K^{\mathsf{ext}})$ and $\widetilde{a}_v \leq O(\overline{a}_K)$ (Equation (6)).

While the colorful matching algorithm is rather straightforward to implement even at distance-2, computing outliers is a surprisingly challenging task. The reason is that, contrary to distance-1, nodes do not know good estimates for $\overline{a}_K$. Fortunately, a node only overestimates its anti-degree (i.e., $\widetilde{a}_v \geq a_v$) and we know that $0.9|K|$ nodes have $a_v \leq 100\overline{a}_K$. By learning approximately the distributions of anti-degrees, we can set a threshold τ such that all nodes with $\widetilde{a}_v \leq \tau$ verify $\widetilde{a}_v \leq 200\overline{a}_K$.

Step 7: Synchronized Color Trial (Section 4.2). This now standard step exploits the small external degree of dense nodes to color most of them. We distributively sample a permutation π of $[|I_K|]$ such that the i-th node in I_K (with respect to any arbitrary order) knows $\pi(i)$ (Lemma 28). Each node then learns the $\pi(i)$-th color in $\Psi(K)$ and tries that color (Lemma 29). This leaves $O(\overline{a}_K + \overline{e}_K + \log n)$ uncolored node in each almost-clique (Lemma 18). To implement these steps, we split nodes into small random groups to spread the workload. The main technical novelty here is an algorithm to aggregate the partial information of each group (Lemma 20).

Step 8: Slice Color (Section 4.3). In the densest almost-cliques, the synchronized color trial already leaves $O(\log n)$ nodes uncolored with uncolored degree $O(\log n)$. In other almost-cliques, nodes have slack proportional to their uncolored degree $O(\overline{a}_K + \overline{e}_K + \widetilde{e}_v)$: $\Theta(\overline{a}_K)$ slack from the colorful matching (Step 4), $\Omega(e_v) + \theta_v^{\mathsf{ext}} = \Omega(\widetilde{e}_v)$ from slack generation and pseudo-external-degree. If these are not large enough, it must be that $\Delta^2 - \widetilde{d}(v) \geq \Omega(\overline{e}_K)$, i.e., the node has enough slack from its small degree.

While at distance-1 we could use slack to color fast, doing the same at distance-2 requires more work because nodes do not know their palettes. The first key observation, is that *the clique-palette preserves the slack.* More precisely, for all inliers $v \in I_K$, we have $|\Psi(K) \cap \Psi(v)| \geq d^\circ(v) + \Omega(\overline{a}_K + \overline{e}_K + \widetilde{e}_v)$ (Lemma 23). The proof of this statement is very technical and requires careful balancing of all four sources of slack: the colorful matching, the sparsity slack, the pseudo-degree slack and the degree slack. We also emphasize this is why inliers need to verify $\widetilde{a}_v \leq O(\overline{a}_K)$: when we use colors from the clique-palette, we lose up to a_v colors used by anti-neighbors, which we compensate using the colorful matching and pseudo-anti-degree slack $\Theta(\overline{a}_K) + \theta_K^{\mathsf{anti}}$.

It remains to sample uniform colors in $\Psi(K) \cap \Psi(v)$. Based on Lemma 23, it can be observed that $|\Psi(K) \cap \Psi(v)| \geq \Omega(|\Psi(K)|)$. Hence, each node v finds a random color $\mathsf{C}_v \in \Psi(K) \cap \Psi(v)$ to try w.h.p. by sampling $\Theta(\log n)$ uniform colors in $\Psi(K)$. With $\Theta(\log^2 n)$ bandwidth, this step can easily be implemented (Lemma 24) by sampling indices in $|\Psi(K)|$ and using the same tools as for the synchronized color trial (Step 7). With $\Theta(\log n)$

bandwidth, we use *representative hash functions* [30]. Intuitively, we use a poly(n)-sized family of hash functions mapping $[\Delta^2 + 1]$ to some $[\Theta(|\Psi(K)|)]$. To "sample" colors, we take a hash function at random and pick as sampled colors those hashing below $\Theta(\log n)$. Since a hash function h can be described in $O(\log n)$ bits and the hashes $h(\Psi(K)) \cap [\sigma]$ and $h(\mathcal{C}(N(v) \setminus K)) \cap [\sigma]$ can be described using a $O(\log n)$-bitmap, the algorithm works in CONGEST.

The above allows us to apply SliceColor after SynchColorTrial. In $O(\log\log n)$ rounds, we compute $\ell = O(\log\log n)$ layers $L_1, L_2, \dots, L_\ell$ such that the maximum uncolored degree in each induced graph $G[L_i]$ is $O(\log n)$ (Lemma 12).

Steps 10 & 11: Coloring Small Degree Instances (Section 4.4). We go through each layer $L_1, L_2, \dots, L_\ell$ sequentially, each time coloring all nodes in L_i. Actually constructing small degree instances for solving with a deterministic algorithm requires the nodes to learn colors from their palette – a tough ordeal in the distance-2 setting. Our argument is two-fold: in not-too-dense almost-cliques, a simple sampling argument works (Lemma 26). In very dense almost-cliques where $\overline{a}_K, \overline{e}_K, \theta_K^{\mathsf{ext}} \le O(\log n)$, we use a different argument exploiting the very high density of the cluster to disseminate colors fast (Lemma 27). We point out that at this step, it is crucial that uncolored nodes have typical degrees $a_v, e_v \le O(\log n)$, which is ensured by our inlier selection (Step 5). Once nodes know a list of $d^\circ(v) + 1$ colors from their palettes, we can use a small-degree algorithm from [28, 23, 38, 25] to complete the coloring of L_i in $O(\log^5 \log n)$ rounds (Proposition 2). Overall, coloring small degree instances needs $O(\log^6 \log n)$ rounds, which dominates the complexity of our algorithm.

4 Coloring Dense Nodes

Henceforth, we assume that we are given an ε-almost-clique decomposition $V_{\mathsf{sparse}}, K_1, \dots, K_k$ for $\varepsilon = 10^{-5}$ [2], where V_{sparse} is already colored. We further assume we ran GenerateSlack and that each node v with $\zeta_v \ge \Omega(\log n)$ has slack $\Omega(\zeta_v)$ (Proposition 5). In this section, we describe an algorithm that colors dense nodes. More formally, we prove the following result:

▶ **Proposition 13** (Coloring Dense Nodes). *After* GenerateSlack *and coloring sparse nodes, there is a $O(\log^6 \log n)$-round randomized algorithm for completing a $\Delta^2 + 1$-coloring of the dense nodes, w.h.p.*

We assume access to $O(\log^2 n)$ bandwidth throughout the rest of the paper, and defer of how to achieve $O(\log n)$ bandwidth to the full version [17, Section 7]. The use of extra bandwidth is very limited and explicitly stated. The reduction in bandwidth only introduces minor changes to the algorithm, and is mostly achieved through techniques from [32].

4.1 Leader, Outliers & Colorful Matching

A useful property of almost-cliques, used by [29, 31, 15], is their relative uniform sparsity. The first step of these algorithms is to dissociate the typical nodes, called *inliers*, from the atypical ones, called *outliers*. At distance-2, however, detecting outliers is difficult. For instance, the algorithm of [15] requires to keep only nodes with anti-degree $a_v \le O(\overline{a}_K)$. Such a trivial task at distance one requires work at distance-2 because nodes are unable to approximate their degree accurately (up to a constant factor). To circumvent this limitation of the distance-2 setting, we instead compute outliers using *pseudo-degrees* (Definition 10).

[2] Note that we made no attempt to optimize the constants.

▶ **Lemma 14** (Compute Outliers). *We compute in $O(\log\log\Delta)$ rounds a set O_K in each almost-clique K such that $I_K \overset{\text{def}}{=} K \setminus O_K$ has size $0.95|K|$ and each $v \in I_K$ verifies that*

$$\widetilde{e}_v \le 200(\overline{e}_K + \theta_K^{\mathsf{ext}}) \ , \quad \text{and} \quad \widetilde{a}_v \le 200\overline{a}_K \ . \tag{6}$$

The general idea behind Lemma 14 is that a large fraction of the almost-clique has a typical sparsity, external degree and anti-degree. By learning approximately the distribution of pseudo-external degrees and pseudo-anti-degrees, the leader can select a large enough fraction of K verifying Equation (6). As the proof of Lemma 14 is quite technical, we defer it to the full version of the paper, [17, Section 6].

Outliers can be colored in $O(\log^* n)$ rounds, thanks to the $\Omega(\Delta^2)$ slack provided by their inactive inlier neighbors. Starting from Section 4.2, we will assume outliers are all colored, thus focus on coloring inliers.

Colorful Matching. A major issue when coloring dense nodes in G^2 is that they do not know their palette. We overcome this by using the *clique palette* as an approximation.

▶ **Definition 15** (Clique Palette). *For an almost-clique K, define its* clique palette *as $\Psi(K) = [\Delta^2 + 1] \setminus \mathcal{C}(K)$, i.e., the set of colors in $\{1, 2, \dots, \Delta^2 + 1\}$ that are not already used by a node of K.*

This idea was first (implicitly) used by [3] to prove their palette sparsification theorem on almost-cliques. This was since used formally in [16, 15]. Note that in large almost-cliques (such that $|K| = (1 + \varepsilon)\Delta^2$), the clique-palette can be empty after coloring the outliers. To remedy this issue, [3] compute first a colorful-matching:

▶ **Definition 16** (Colorful Matching). *In a clique K, a colorful matching M is a set of anti-edges in K (edges in the complement) such that both endpoints are colored the same.*

Flin, Ghaffari, Halldórsson, Kuhn and Nolin gave a CONGEST algorithm to compute a colorful matching of size $\Theta(\overline{a}_K/\varepsilon)$ in $O(1/\varepsilon)$ rounds in cliques with a high average anti-degree [15]. We review this algorithm and argue it can be implemented on G^2 with constant overhead in the full version of the paper (see [17, Appendix C]).

▶ **Proposition 17** (Distance-2 Colorful Matching). *Let $\beta \le O(1/\varepsilon)$. There exists a $O(\beta)$-round randomized algorithm* Matching *that computes a colorful matching of size $\beta\overline{a}_K$ in all almost-cliques of G^2 with $\overline{a}_K \ge \Omega(\log n)$.*

4.2 Synchronized Color Trial

Synchronizing color trials in dense components is a fundamental part of all known sub-logarithmic algorithm [33, 9, 29, 31]. We implement a variant of [31] where a uniform permutation determines which node tries which color. Contrary to [31], we use colors from the clique palette $\Psi(K)$ (Definition 15), which is easier to implement in our setting. This approach was also used by [15] to implement the synchronized color trial in Broadcast-CONGEST. A major difference with [15] is that at distance-2, nodes cannot learn the whole clique-palette $\Psi(K)$.

▶ **Lemma 18** (Synchronized Color Trial, [31]). *Let K be an almost-clique with $|I_K| \ge \Omega(|K|)$ inliers. Fix the randomness outside K arbitrarily. Let π be a uniform random permutation of $[|I_K|]$. If the i-th node in I_K (for any arbitrary order) tries the $\pi(i)$-th color in $\Psi(K)$ (if it exists), then, with high probability, at most $O(\overline{e}_K + \overline{a}_K + \log n)$ are uncolored in K.*

To implement the synchronized color trial, a node v needs only to know $\pi(v)$ and the $\pi(v)$-th color of $\Psi(K)$. We use an approach similar to [15]: randomly partition nodes into groups $T_1, \dots, T_k$ to spread the workload. Concretely, we use the following fact, which is a straightforward consequence of Chernoff and Definition 6.2b.

▶ **Fact 19.** *Let K be an almost-clique and $k \le |K|/(C \log n)$ for some large enough $C > 0$. Suppose each $v \in K$ samples $t(v) \in [k]$ uniformly at random. Then, w.h.p., each $T_i = \{v \in K : t(v) = i\}$ satisfies that any $u, w \in K$ have $|N^2(u) \cap N^2(w) \cap T_i| \ge (C/4) \log n$. We say set T_i 2-hop connects K.*

Note that the two hops mentioned in Fact 19 are in G^2, i.e., for two nodes $u, w \in K$ where T_i 2-hop connects K, u and w can be at distance 4 in G.

Contrary to [15], at distance-2, nodes do not have the bandwidth to learn the whole clique-palette nor the full random permutation. Fortunately, they only need to know their position in the permutation and the one corresponding color. The main technical novelty in our distance-2 implementation lies in an algorithm to compute prefix-sums $\sum_{j<i} x_j$ where each random group T_i holds a value x_i (Lemma 20). We first explain how to aggregate such prefix sums and then show it is enough for implementing the synchronized color trial.

▶ **Lemma 20** (Prefix Sums). *Let $T_1, \dots, T_k \subseteq K$ be disjoint sets that 2-hop connect K. If each T_i holds a $\operatorname{poly}\log n$-bit integer x_i, then there is a $O(1)$-round algorithm such that for all $i \in [k]$, each $v \in T_i$ learns $\sum_{j<i} x_j$.*

Proof. Compute a BFS tree rooted at some arbitrary $w_K \in K$ and spanning $N^2(w_K) \cap K$. We order distance-2 neighbors of w_K with the *lexicographical order induced by the BFS tree*: distance-2 neighbors $u \in N_{G^2}(w_K)$ are ordered first by $\mathsf{ID}(v)$, where v is the parent of u in the BFS tree, and then by $\mathsf{ID}(u)$. Call $u_1, u_2, \dots, u_{|N_{G^2}(w_K) \cap K|}$ distance-2 neighbors of w_K with respect to that ordering.For each $i \in [k]$, *node u_i learns x_i*. Since T_i 2-hop connects K, there must exist a node $r \in N(u_i) \cap N(T_i)$ which can relay x_i from its neighbor in T_i to u_i. For each distance-1 neighbor $v_j \in N(w_K) \cap K$ of w_K (i.e., depth-1 nodes in the BFS tree), let $u_{i_j}, u_{i_j+1}, \dots, u_{i_{j+1}-1}$ be its children in the BFS tree. Each v_j can learn all values $x_{i_j}, \dots, x_{i_{j+1}-1}$ with a broadcast. Node v_j then sends the sum $S_j \overset{\text{def}}{=} \sum_{k=i_j}^{i_{j+1}-1} x_k$ to w_K, which responds with $\sum_{k<j} S_j = \sum_{k<i_j} x_k$. For each child u_{i_j+t} with $0 \le t \le i_{j+1} - i_j$, the node v_j communicates

$$\sum_{k<j} S_k + \sum_{k=i_j}^{i_j+t-1} x_k = \sum_{k<i_j+t} x_k$$

to u_{i_j+t}, which is exactly the prefix sum it had to learn. Each u_i can then transmit its prefix sum to T_i using the same path it used to learn x_i. ◀

We briefly sketch the algorithm for implementing the synchronized color trial using Lemma 20 and random groups in the following lemma. See Appendix A for more details.

▶ **Lemma 21.** *Let $v_1, \dots, v_{|I_K|} \in I_K$ be the inliers of some almost-clique K. In $O(\log \log n)$ rounds of* CONGEST*, with high probability,*
1. *we can sample a uniform permutation π of $[|I_K|]$ such that v_i knows $\pi(i)$; and*
2. *each v_i can learn i_v-th color of $\Psi(K)$, for any $i_v \in [\Delta + 1]$ (and fail if $|\Psi(K)| < i_v$).*

Proof Sketch. We begin by explaining how to sample the permutation. Each node $v \in I_K$ picks an integer $t(i) \in [\Theta(|K|/\log n)]$ at random. Let $T_i = \{v \in S : t(v) = i\}$. By Chernoff bound, w.h.p., $|T_i| = O(\log n)$ and 2-hop connects K (Fact 19). In particular, each T_i has hop-diameter at most 4 and we can relabel nodes of T_i using $O(\log\log n)$-bit labels in $O(1)$ rounds. Using small labels, each T_i can sample a permutation ρ_i of itself in $O(\log\log n)$ rounds. The permutation of I_K is defined by $\pi(v) = \sum_{j<t(v)} |T_j| + \rho_{t(v)}(v)$; hence, using the prefix sum algorithm (Lemma 20), nodes learn $\sum_{j<t(v)} |T_j|$ and have the required information to compute their position in π.

For learning colors, we split nodes into $\Theta(\Delta^2/\log n)$ groups randomly. Random group T_i is tasked with learning which colors in range $R_i = \{i \cdot \Theta(\log n), \dots, (i+1)\cdot\Theta(\log n) - 1\}$ are used by a node in K. With high probability, each T_i 2-hop connects K; thus, using $O(\log n)$-bitmaps and simple aggregation, nodes in T_i learn $R_i \cap \mathcal{C}(K)$ in $O(1)$ rounds. Hence, each node in T_i learns $R_i \setminus \mathcal{C}(K) = R_i \cap \Psi(K)$. By computing prefix sums $\sum_{j<i} |R_j \cap \Psi(K)|$, nodes of T_i learn which range of queries $i \in [|\Psi(K)|]$ they must respond. Since each T_i 2-hop connects K, each node in K has a relay to the group which must answer its query. ◀

4.3 Slack Color (with extra bandwidth)

After the synchronized color trial, uncolored nodes have degree proportional to the slack they received from GenerateSlack (Proposition 5). Contrary to [9, 29, 31], nodes cannot trivially try colors from their palettes, for they lack direct knowledge of it. In this section, we give a solution that uses $O(\log^2 n)$ bandwidth and refer to [17] for the CONGEST implementation. The idea is to sample $\Theta(\log n)$ colors from the clique palette, which is accessible by Lemma 29. Note that this step is needed only in high-sparsity cliques: if $\overline{a}_K + \overline{e}_K \le O(\log n)$, then its remaining uncolored nodes after the synchronized color trial have degree $O(\log n)$. This motivates the following definition:

▶ **Definition 22** ($\mathcal{K}_{\mathsf{mod}}$, $\mathcal{K}_{\mathsf{very}}$). *Let $C > 0$ be a large enough constant. We say that almost-clique K is* very dense *if $\overline{a}_K < C\log n$ and $\overline{e}_K, \theta^{\mathsf{ext}}_K < 4C\log n$. Reciprocally, we say K is* moderately dense *if it is not very dense. We call $\mathcal{K}_{\mathsf{mod}}$ the set of moderately dense almost-cliques and $\mathcal{K}_{\mathsf{very}}$ the very dense ones.*

Lemma 23 shows that in moderately dense almost-cliques, the clique palette preserves the slack provided by early steps of the algorithm (slack generation, colorful matching and degree slack).

▶ **Lemma 23** (The Clique Palette Preserves Slack). *After* GenerateSlack *and* Matching*, for all inlier $v \in I_K$ with $K \in \mathcal{K}_{\mathsf{mod}}$, we have $|\Psi(v) \cap \Psi(K)| \ge d^\circ(v) + \Omega(\widetilde{e}_v + \overline{e}_K + \overline{a}_K)$. In particular, for any such $v \in I_K$ with $K \in \mathcal{K}_{\mathsf{mod}}$, we have $|\Psi(v) \cap \Psi(K)| \ge \Omega(|\Psi(K)|)$.*

Proof. Clearly, $|K| = |N^2(v) \cap K| + a_v$. We carefully add all contributions to the degree slack of a node

$$\Delta^2 = (\Delta^2 - \widetilde{d}(v)) + \widetilde{d}(v) = |N^2(v) \cap K| + e_v + (\Delta^2 - \widetilde{d}(v)) + \theta^{\mathsf{ext}}_v + \theta^{\mathsf{anti}}_v .$$

The clique palette loses one color for each colored node but saves one for each edge in the colorful matching. Recall that K° denotes the uncolored part of K. The clique palette has size at least

$$|\Psi(K)| \ge \Delta^2 - (|K| - |K^\circ|) + |M| \ge |K^\circ| + e_v + |M| - a_v + (\Delta^2 - \widetilde{d}(v)) + \theta^{\mathsf{ext}}_v + \theta^{\mathsf{anti}}_v .$$

Let s be the slack received w.h.p. by v after GenerateSlack: if $e_v \ge C\log n$ then $s \overset{\text{def}}{=} \Omega(\zeta_v) \ge \Omega(e_v)$ (Proposition 5 and Lemma 9), otherwise $s = 0$. The palette of v is of size at least

$$|\Psi(v)| \ge d^\circ(v) + s + (\Delta^2 - \widetilde{d}(v)) + \theta^{\mathsf{ext}}_v + \theta^{\mathsf{anti}}_v .$$

Notice $|\Psi(v) \setminus \Psi(K)| \le a_v$ and $|\Psi(K) \setminus \Psi(v)| \le e_v^\bullet$ (recall $e_v^\bullet$ is the number of *colored* external neighbors). A double counting argument bounds the number of colors in both v's palette and the clique palette:

$$\begin{aligned} 2|\Psi(v) \cap \Psi(K)| &= |\Psi(v)| + |\Psi(K)| - |\Psi(v) \setminus \Psi(K)| - |\Psi(K) \setminus \Psi(v)| \\ &\ge d^\circ(v) + |K^\circ| + (e_v - e_v^\bullet) + s + |M| + 2(\theta_v^{\mathsf{anti}} - a_v + \theta_v^{\mathsf{ext}} + \Delta^2 - \widetilde{d}(v)) \\ &\ge 2d^\circ(v) + s + |M| - 2\widetilde{a}_v + 2\theta_v^{\mathsf{ext}} + 2(\Delta^2 - \widetilde{d}(v)) \ , \end{aligned} \tag{7}$$

where the second inequality uses $|K^\circ| + (e_v - e_v^\bullet) \ge d^\circ(v)$ and $\theta_v^{\mathsf{anti}} - a_v = (a_v - \widetilde{a}_v) - a_v = -\widetilde{a}_v$.

The remaining of this proof is a careful case analysis to show that Equation (7) implies $|\Psi(v) \cap \Psi(K)| \ge d^\circ(v) + \Omega(\widetilde{e}_v + \overline{e}_K + \overline{a}_K)$. Each case of our analysis corresponds to a regime for $\overline{a}_K$ and $\overline{e}_K$, since v receives slack from the coloring matching only when $\overline{a}_K > \Omega(\log n)$ (Proposition 17) and from slack generation when $e_v \ge \Omega(\log n)$ (Proposition 5). We defer the detail of this case analysis to Appendix B.

Constant density. Observe that if $|\Psi(K)| > 2e_v$ then $|\Psi(v) \cap \Psi(K)| \ge |\Psi(K)| - e_v \ge |\Psi(K)|/2$. Otherwise if $e_v \ge |\Psi(K)|/2$, we use $|\Psi(v) \cap \Psi(K)| \ge \Omega(e_v)$ (which we just proved) to deduce that $|\Psi(v) \cap \Psi(K)| \ge \Omega(e_v) \ge \Omega(|\Psi(K)|)$. ◀

Lemma 24 is the main implication of Lemma 23. It states that we can use random sampling in the clique palette, instead of nodes' palettes, to try colors in SliceColor (Lemma 12). In particular, after SynchColorTrial (Step Algorithm 1 in Algorithm 1), SliceColor with the sampling process described in Lemma 24 reduces degrees to $O(\log n)$ in $O(\log \log n)$ rounds.

▶ **Lemma 24.** *There is an $O(1)$-round algorithm (using $O(\log^2 n)$ bandwidth) that when run after* GenerateSlack *and* Matching*, achieves the following: It samples a random color $\mathsf{C}_v \in \Psi(v) \cup \{\perp\}$ for all uncolored dense nodes $v \in K \in \mathcal{K}_{\mathsf{mod}}$ such that $\mathsf{Pr}(\mathsf{C}_v = \perp) \le 1/\operatorname{poly}(n)$ and $\mathsf{Pr}(\mathsf{C}_v = c) \le \frac{1}{d^\circ(v) + \Omega(\overline{a}_K + \overline{e}_K + \widetilde{e}_v)}$ for all colors $c \in \Psi(v) \cap \Psi(K)$.*

Proof. Fix a node $v \in K$. Nodes of K can learn $|\Psi(K)|$ by Lemma 29. Then v samples $x = \Theta(\log n)$ indices in $[|\Psi(K)|]$. By Lemma 29, using $O(\log^2 n)$ bandwidth, each node can learn in $O(1)$ rounds the colors corresponding to the indices they sampled. They broadcast this list of colors (using $O(\log^2 n)$ bandwidth) and drop all colors used by neighbors (i.e., that are not in their palette). Finally, node v picks C_v uniformly at random among the remaining ones. Since $|\Psi(K) \cap \Psi(v)| \ge \Omega(|\Psi(K)|)$, by sampling $\Theta(\log n)$ colors, we sample at least one color $\Psi(K) \cap \Psi(v)$ with high probability (i.e., $\mathsf{Pr}(\mathsf{C}_v = \perp) \le 1/\operatorname{poly}(n)$). To argue about the uniformity (Equation (5)), we observe that sampling $x = \Theta(\log n)$ indices in $[|\Psi(K)|]$ and then trying a random one of those is equivalent to sampling a uniform permutation π of $[|\Psi(K)|]$ (the x sampled indices are $\pi^{-1}(1), \dots, \pi^{-1}(x)$) and trying the color $c \in \Psi(K) \cap \Psi(v)$ with the smallest $\pi(c)$ (if $\min \pi(\Psi(K) \cap \Psi(v)) < x$). Hence, if we call $Z = \min \pi(\Psi(K) \cap \Psi(v))$, we have

$$\begin{aligned} \mathsf{Pr}(\mathsf{C}_v = c) &= \mathsf{Pr}(Z < x \ \wedge \ \pi(c) = Z) \\ &\le \mathsf{Pr}(\pi(c) = Z) = \frac{1}{|\Psi(K) \cap \Psi(v)|} \\ &\le \frac{1}{d^\circ(v) + \Omega(\overline{a}_K + \overline{e}_K + \widetilde{e}_v)} \ . \qquad \text{(by Lemma 23)} \end{aligned}$$

◀

4.4 Learning Small Palettes (with extra bandwidth)

Assume we are given sets $L_1, \dots, L_\ell$ for some $\ell = O(\log\log n)$ such that the maximum uncolored degree in each $G[L_i]$ is at most $O(\log n)$. We explain how nodes learn a list $L(v)$ of $d^\circ(v) + 1$ colors in their palette, with respect to the current coloring of G^2.

▶ **Lemma 25** (Learn Palette). *Let H be an induced subgraph of G^2 with maximum uncolored degree $O(\log n)$. There is a $O(\log\log n)$-round algorithm at the end of which each node in H knows a set $L(v) \subseteq \Psi(v)$ of $d^\circ_H(v) + 1$ colors with high probability.*

The argument is two-fold, we deal with $v \in K \in \mathcal{K}_{\mathsf{mod}}$ nodes and very dense nodes $v \in K \in \mathcal{K}_{\mathsf{very}}$ separately. We again assume $O(\log^2 n)$ bandwidth. The $O(\log n)$ bandwidth argument can be found in [17].

Moderately Dense Almost-Cliques. Using the sampling algorithm from Lemma 24, nodes can sample $C \log n$ many colors in their palette in $O(1)$ rounds, for any arbitrarily large constant $C > 0$. Since uncolored degrees in H are $O(\log n)$, this suffices for Lemma 25.

▶ **Lemma 26.** *Let H be an induced subgraph of G^2 with maximum uncolored degree $C' \log n$ for a large constant $C' > 0$. There is a $O(1)$-round algorithm (using $O(\log^2 n)$ bandwidth) for $v \in K \in \mathcal{K}_{\mathsf{mod}}$ to learn a list $L(v)$ of $d^\circ_H(v) + 1$ colors from their palettes.*

Very Dense Almost-Cliques. In very dense almost-cliques, the clique palette $\Psi(K)$ does not approximate their palette well enough: Lemma 23 does not apply. We correct for that by adding colors used by anti-neighbors. They filter out colors used by external neighbors with $O(\log^2 n)$ bandwidth, because they have $O(\log n)$ such neighbors.

▶ **Lemma 27.** *Suppose each $K \in \mathcal{K}_{\mathsf{very}}$ has $O(\log n)$ uncolored nodes (hence $d^\circ(v) \le O(\log n)$ for all $v \in K \in \mathcal{K}_{\mathsf{very}}$). There is a $O(\log\log n)$-round randomized algorithm (using $O(\log^2 n)$ bandwidth) for all uncolored nodes $v \in K \in \mathcal{K}_{\mathsf{very}}$ to learn a list $L(v)$ of $d^\circ_H(v) + 1$ colors from their palettes.*

Proof. By repeating messages randomly, we can broadcast any $O(\log^2 n)$-many messages to all nodes in K (see Lemma 31). In particular, when $|\Psi(K)| = O(\log^2 n)$, all nodes in K learn all colors in $\Psi(K)$ in $O(1)$ rounds (see Lemma 32). If $|\Psi(K)| \ge \Omega(\log^2 n)$, nodes learn in $O(1)$ rounds a set $D \subseteq \Psi(K)$ of $\Theta(\log^2 n)$ colors.

Assume first that nodes learned all colors of $\Psi(K)$. Recall nodes have $e_v = O(\log n)$ external neighbors (because $K \in \mathcal{K}_{\mathsf{very}}$ and $v \in I_K$); hence, they can learn all colors used by their external neighbors in $O(1)$ rounds by using $O(\log^2 n)$ bandwidth. Since each uncolored v knows $\Psi(K)$ and the colors of its external neighbors, it thereby knows $\Psi(K) \cap \Psi(v)$.

With a BFS, we can relabel uncolored node of K in the range $[O(\log n)]$. Since uncolored nodes are inliers, they have anti-degree $a_v \le O(\overline{a}_K) \le O(\log n)$ each. At most $O(\log^2 n)$ nodes in K are anti-neighbors of (at least one) uncolored node. We can run $O(\log n)$ BFS in parallel, one rooted at each uncolored node, such that each node knows to which uncolored node it is connected (at distance-2). This takes $O(\log\log n)$ rounds, even with bandwidth $O(\log n)$, because each BFS uses $O(\log\log n)$-bit messages (thanks to the relabeling) and we run $O(\log n)$ of them. Then, the $O(\log^2 n)$ nodes with an uncolored anti-neighbor can describe their list of uncolored anti-neighbors using a $O(\log n)$-bitmap. Using Lemma 31, they broadcast this information as well as their color to all nodes.

Nodes use lists $L(v) \overset{\text{def}}{=} (\Psi(K) \cup \mathcal{C}(K \setminus N^2(v))) \cap \Psi(v)$, i.e., the clique palette augmented with the colors of their anti-neighbors, minus colors used by external neighbors. Adding $\Delta^2+1 \geq |N^2(v) \cap K| + e_v$ and $|K| \leq |N^2(v) \cap K| + a_v$, we get $|\Psi(K)| \geq \Delta^2+1-(|K|-|K^\circ|) \geq |K^\circ| + 1 + e_v - a_v$. Since each colored external neighbor removes as most one color, lists have size (recall $e_v^\bullet$ and $a_v^\bullet$ are the *colored* external degree and anti-degrees respectively)

$$|L(v)| \geq |\Psi(K)| - e_v^\bullet + a_v^\bullet \geq |K^\circ| + (e_v - e_v^\bullet) - (a_v - a_v^\bullet) + 1 = d^\circ(v) + 1 \ .$$

Suppose we are in the second case of Lemma 26, i.e., nodes learn a set $D \subsetneq \Psi(K)$ of $\Theta(\log^2 n)$ colors. Nodes set $L(v) = D \setminus \mathcal{C}(N^2(v) \setminus K) = D \cap \Psi(v)$, i.e., remove colors used by external neighbors. Since they have $e_v \leq O(\log n)$, this provides large enough lists. For the detailed analysis, see the end of Appendix C. ◀

References

1 Noga Alon and Sepehr Assadi. Palette sparsification beyond $(\Delta+1)$ vertex coloring. In *APPROX/RANDOM*, volume 176 of *LIPIcs*, pages 6:1–6:22. LZI, 2020. doi:10.4230/LIPIcs.APPROX/RANDOM.2020.6.

2 Noga Alon, László Babai, and Alon Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *J. of Algorithms*, 7(4):567–583, 1986. doi:10.1016/0196-6774(86)90019-2.

3 Sepehr Assadi, Yu Chen, and Sanjeev Khanna. Sublinear algorithms for $(\Delta+1)$ vertex coloring. In *SODA*, pages 767–786. SIAM, 2019. doi:10.1137/1.9781611975482.48.

4 Leonid Barenboim and Michael Elkin. *Distributed Graph Coloring: Fundamentals and Recent Developments.* Morgan & Claypool Publishers, 2013. doi:10.2200/S00520ED1V01Y201307DCT011.

5 Leonid Barenboim, Michael Elkin, and Uri Goldenberg. Locally-iterative distributed $(\Delta+1)$-coloring and applications. *J. ACM*, 69(1):5:1–5:26, 2022. doi:10.1145/3486625.

6 Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The locality of distributed symmetry breaking. *Journal of the ACM*, 63(3):20:1–20:45, 2016. doi:10.1145/2903137.

7 Leonid Barenboim and Uri Goldenberg. Speedup of distributed algorithms for power graphs in the CONGEST model. Technical Report 2305.04358, arXiv, 2023. doi:10.48550/arXiv.2305.04358.

8 Yi-Jun Chang, Qizheng He, Wenzheng Li, Seth Pettie, and Jara Uitto. The complexity of distributed edge coloring with small palettes. In *SODA*, pages 2633–2652. SIAM, 2018. doi:10.1137/1.9781611975031.168.

9 Yi-Jun Chang, Wenzheng Li, and Seth Pettie. Distributed $(\Delta+1)$-coloring via ultrafast graph shattering. *SIAM J. Computing*, 49(3):497–539, 2020. doi:10.1137/19M1249527.

10 Kai-Min Chung, Seth Pettie, and Hsin-Hao Su. Distributed algorithms for the Lovász local lemma and graph coloring. *Distributed Comput.*, 30(4):261–280, 2017. doi:10.1007/s00446-016-0287-6.

11 Michael Elkin and Shaked Matar. Near-additive spanners in low polynomial deterministic CONGEST time. In *PODC*, pages 531–540. ACM, 2019. doi:10.1145/3293611.3331635.

12 Michael Elkin, Seth Pettie, and Hsin-Hao Su. $(2\Delta-1)$-edge-coloring is much easier than maximal matching in the distributed setting. In *SODA*, pages 355–370. SIAM, 2015. doi:10.1137/1.9781611973730.26.

13 Salwa Faour, Mohsen Ghaffari, Christoph Grunau, Fabian Kuhn, and Václav Rozhoň. Local distributed rounding: Generalized to MIS, matching, set cover, and beyond. In *SODA*, pages 4409–4447. SIAM, 2023. doi:10.1137/1.9781611977554.ch168.

14 Manuela Fischer, Magnús M. Halldórsson, and Yannic Maus. Fast distributed Brooks' theorem. In *SODA*, pages 2567–2588. SIAM, 2023. doi:10.1137/1.9781611977554.ch98.

15 Maxime Flin, Mohsen Ghaffari, Magnús M. Halldórsson, Fabian Kuhn, and Alexandre Nolin. Coloring fast with broadcasts. In *SPAA*, pages 455–465. ACM, 2023. doi:10.1145/3558481.3591095.

16 Maxime Flin, Mohsen Ghaffari, Magnús M. Halldórsson, Fabian Kuhn, and Alexandre Nolin. A distributed palette sparsification theorem. Technical Report 2301.06457, arXiv, 2023. doi:10.48550/arxiv.2301.06457.

17 Maxime Flin, Magnús M. Halldórsson, and Alexandre Nolin. Fast coloring despite congested relays. Technical Report 2308.01359, arXiv, 2023. Full version of this paper. doi:10.48550/arxiv.2308.01359.

18 Pierre Fraigniaud, Magnús M. Halldórsson, and Alexandre Nolin. Distributed testing of distance-k colorings. In *SIROCCO*, volume 12156 of *LNCS*, pages 275–290. Springer, 2020. doi:10.1007/978-3-030-54921-3_16.

19 Pierre Fraigniaud, Marc Heinrich, and Adrian Kosowski. Local conflict coloring. In *FOCS*, 2016. doi:10.1109/FOCS.2016.73.

20 Mohsen Ghaffari. An improved distributed algorithm for maximal independent set. In *SODA*, pages 270–277. SIAM, 2016. doi:10.1137/1.9781611974331.ch20.

21 Mohsen Ghaffari. Distributed maximal independent set using small messages. In *SODA*, pages 805–820. SIAM, 2019. doi:10.1137/1.9781611975482.50.

22 Mohsen Ghaffari and Christoph Grunau. Faster deterministic distributed MIS and approximate matching. *STOC*, abs/2303.16043, 2023. doi:10.48550/arXiv.2303.16043.

23 Mohsen Ghaffari, Christoph Grunau, Bernhard Haeupler, Saeed Ilchi, and Václav Rozhoň. Improved distributed network decomposition, hitting sets, and spanners, via derandomization. In *SODA*, pages 2532–2566. SIAM, 2023. doi:10.1137/1.9781611977554.ch97.

24 Mohsen Ghaffari, Christoph Grunau, and Václav Rozhoň. Improved deterministic network decomposition. In *SODA*, 2021. arXiv:2007.08253.

25 Mohsen Ghaffari and Fabian Kuhn. Deterministic distributed vertex coloring: Simpler, faster, and without network decomposition. In *FOCS*, pages 1009–1020. IEEE Computer Society, 2021. doi:10.1109/FOCS52979.2021.00101.

26 Mohsen Ghaffari and Julian Portmann. Improved network decompositions using small messages with applications on MIS, neighborhood covers, and beyond. In *DISC*, volume 146 of *LIPIcs*, pages 18:1–18:16. LZI, 2019. doi:10.4230/LIPIcs.DISC.2019.18.

27 Magnús M. Halldórsson, Fabian Kuhn, and Yannic Maus. Distance-2 coloring in the CONGEST model. In *PODC*, pages 233–242. ACM, 2020. doi:10.1145/3382734.3405706.

28 Magnús M. Halldórsson, Fabian Kuhn, Yannic Maus, and Alexandre Nolin. Coloring fast without learning your neighbors' colors. In *DISC*, pages 39:1–39:17. LZI, 2020. doi:10.4230/LIPIcs.DISC.2020.39.

29 Magnús M. Halldórsson, Fabian Kuhn, Yannic Maus, and Tigran Tonoyan. Efficient randomized distributed coloring in CONGEST. In *STOC*, pages 1180–1193. ACM, 2021. doi:10.1145/3406325.3451089.

30 Magnús M. Halldórsson and Alexandre Nolin. Superfast coloring in CONGEST via efficient color sampling. *Theor. Comput. Sci.*, 948:113711, 2023. doi:10.1016/j.tcs.2023.113711.

31 Magnús M. Halldórsson, Fabian Kuhn, Alexandre Nolin, and Tigran Tonoyan. Near-optimal distributed degree+1 coloring. In *STOC*, pages 450–463. ACM, 2022. doi:10.1145/3519935.3520023.

32 Magnús M. Halldórsson, Alexandre Nolin, and Tigran Tonoyan. Overcoming congestion in distributed coloring. In *PODC*, pages 26–36. ACM, 2022. doi:10.1145/3519270.3538438.

33 David G. Harris, Johannes Schneider, and Hsin-Hao Su. Distributed $(\Delta + 1)$-coloring in sublogarithmic rounds. *Journal of the ACM*, 65:19:1–19:21, 2018. doi:10.1145/3178120.

34 Öjvind Johansson. Simple distributed $\Delta + 1$-coloring of graphs. *Inf. Process. Lett.*, 70(5):229–232, 1999. doi:10.1016/S0020-0190(99)00064-2.

35 Sven Oliver Krumke, Madhav V. Marathe, and S. S. Ravi. Models and approximation algorithms for channel assignment in radio networks. *Wirel. Networks*, 7(6):575–584, 2001. `doi:10.1023/A:1012311216333`.

36 Nathan Linial. Locality in distributed graph algorithms. *SIAM J. Computing*, 21(1):193–201, 1992. `doi:10.1137/0221015`.

37 M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Computing*, 15:1036–1053, 1986. `doi:10.1137/0215074`.

38 Yannic Maus, Saku Peltonen, and Jara Uitto. Distributed symmetry breaking on power graphs via sparsification. In *PODC*, pages 157–167. ACM, 2023. full version available at arxiv:2302.06878. `doi:10.1145/3583668.3594579`.

39 Yannic Maus and Tigran Tonoyan. Linial for lists. *Distributed Comput.*, 35(6):533–546, 2022. `doi:10.1007/s00446-022-00424-y`.

40 Yannic Maus and Jara Uitto. Efficient CONGEST algorithms for the Lovász local lemma. In *DISC*, volume 209 of *LIPIcs*, pages 31:1–31:19. LZI, 2021. `doi:10.4230/LIPIcs.DISC.2021.31`.

41 Seth Pettie and Hsin-Hao Su. Distributed coloring algorithms for triangle-free graphs. *Inf. Comput.*, 243:263–280, 2015. `doi:10.1016/j.ic.2014.12.018`.

42 Bruce A. Reed. ω, Δ, and χ. *J. Graph Theory*, 27(4):177–212, 1998. `doi:10.1002/(SICI)1097-0118(199804)27:4<177::AID-JGT1>3.0.CO;2-K`.

43 Václav Rozhon and Mohsen Ghaffari. Polylogarithmic-time deterministic network decomposition and distributed derandomization. In *STOC*, pages 350–363. ACM, 2020. `doi:10.1145/3357713.3384298`.

44 Johannes Schneider and Roger Wattenhofer. A new technique for distributed symmetry breaking. In *PODC*, pages 257–266. ACM, 2010. `doi:10.1145/1835698.1835760`.

A Missing Details in Synchronized Color Trial

In this section, we expand on the proof sketch of Lemma 21 in the main text and fill in some of the missing details of how we implement Synchronized Color Trial in the distance-2 setting. Similarly to how we used random groups to compute prefix sums in Lemma 20, permuting the nodes (Lemma 28) and learning the colors used in the almost-clique (Lemma 29) are performed by assigning nodes randomly to groups which each perform a chunk of the workload. Assembling together the work done in each group is done using our algorithm for computing prefix sums (Lemma 20), which was the most novel part of the implementation. Algorithmic ideas behind Lemmas 28 and 29 are very similar to the ones of [15], and we discuss them more briefly.

▶ **Lemma 28** (Permute). *There is an algorithm that samples a uniform permutation π of $[|I_K|]$ in $O(1)$ rounds with high probability. The i-th node in I_K (with respect to any ordering where v knows its index) learns $\pi(i)$.*

Proof. Each node $v \in I_K$ picks an integer $t(i) \in [\Theta(|K|/\log n)]$ at random. Let $T_i = \{v \in S : t(v) = i\}$. By Chernoff bound, w.h.p., $|T_i| = O(\log n)$ and 2-hop connects K (Fact 19). In particular, each T_i has hop-diameter at most 4. Let w_i be the node of minimum ID in T_i. Each T_i computes a spanning tree rooted at its w_i. This is performed in parallel for all groups, by having nodes forward the minimum ID they received from a group T_i to other members of T_i. Note that an edge only needs to send information concerning the two groups of its endpoints. Each T_i then relabels itself using small $O(\log\log n)$-bit identifiers in the range $[|T_i|]$. w_i samples a permutation ρ_i of $|T_i|$ and broadcasts it to T_i. Since the permutation of a group needs $O(\log n) \times O(\log\log n)$ bits, after $O(\log\log n)$ rounds each $v \in T_i$ knows $\rho_i(v)$. Then, using Lemma 20, each v learns $\sum_{j<i} |T_j|$. Finally, node v sets its position to $\pi(v) = \sum_{j<i} |T_j| + \rho_i(v)$. ◀

▶ **Lemma 29** (Free Color). *Suppose each node in $v \in K$ holds an integer $i_v \in [\Delta^2+1]$. There is $O(1)$-round algorithm at the end of which each v knows the i_v-th color of $\Psi(K)$ (with respect to any globally known total order of $\Psi(K)$). Furthermore, all nodes can learn $|\Psi(K)|$ in the process.*

Proof. Each node $v \in K$ picks an integer $t(v) \in [\Theta(\Delta^2/\log n)]$. Let $T_i = \{v \in K : t(v) = i\}$. Again, w.h.p., $|T_i| = O(\log n)$ and T_i 2-hop connects K. Each node broadcasts its color (if it adopted one) and its group number $t(v)$. Let $R_i = \{i \cdot \Theta(\log n), \dots, (i+1)\cdot\Theta(\log n) - 1\}$. Let $S_{u,i} = R_i \cap \mathcal{C}(N(u) \cap K)$ be the colors from range R_i used by neighbors of u. For each $i \in [k]$, node u can describe $S_{u,i}$ to each neighbor in T_i using a $O(\log n)$-bitmap. Since each T_i has diameter 4 and 2-hop connects K, after $O(1)$ rounds of aggregation on bitmaps using a bitwise OR, each node in T_i knows $R_i \cap \mathcal{C}(K)$, i.e., all colors from range R_i used in K. Note that this also allows them to compute $R_i \setminus \mathcal{C}(K) = R_i \cap \Psi(K)$, i.e., the colors of R_i that are *not* used by a node of K. By Lemma 20, nodes of T_i learn $\sum_{j<i} |R_j \cap \Psi(K)|$ in $O(1)$ rounds. Finally each v broadcasts i_v and each $u \in T_i$ broadcasts i, $\sum_{j<i} |R_j \cap \Psi(K)|$ and $R_i \setminus \mathcal{C}(K)$. Since each set T_i 2-hop connects K, if the i_v-th color of $\Psi(K)$ belongs to range R_i (i.e., $\sum_{j<i} |R_j \cap \Psi(K)| \le i_v < \sum_{j\le i} |R_j \cap \Psi(K)|$), then there exists a $u \in T_i$ and $w \in N(u) \cap N(v)$ which knows both i_v and the color it corresponds to. Then w can transmit that information to v.

To learn $|\Psi(K)|$, nodes aggregate the sum of all $|R_i \cap \Psi(K)|$. This can easily done with a BFS (and electing a leader in each group to avoid double counting). ◀

B Missing Details in the Proof of Lemma 23

▶ **Lemma 23** (The Clique Palette Preserves Slack). *After* GenerateSlack *and* Matching*, for all inlier $v \in I_K$ with $K \in \mathcal{K}_{\mathsf{mod}}$, we have $|\Psi(v) \cap \Psi(K)| \ge d^\circ(v) + \Omega(\widetilde{e}_v + \overline{e}_K + \overline{a}_K)$. In particular, for any such $v \in I_K$ with $K \in \mathcal{K}_{\mathsf{mod}}$, we have $|\Psi(v) \cap \Psi(K)| \ge \Omega(|\Psi(K)|)$.*

In this section, we give the details of a case analysis which we skipped in the proof of Lemma 23 in the main text. In Section 4.3, we show Equation (7):

$$2|\Psi(v) \cap \Psi(K)| \ge 2d^\circ(v) + s + |M| - 2\widetilde{a}_v + 2\theta_v^{\mathsf{ext}} + 2(\Delta^2 - \widetilde{d}(v)) \ .$$

To complete the proof, we show that Equation (7) implies that

$$|\Psi(v) \cap \Psi(K)| \ge d^\circ(v) + \Omega(\widetilde{e}_v + \overline{e}_K + \overline{a}_K) \ .$$

Henceforth, slack implicitly refers to the slack in the clique palette, i.e., node v has slack x if $|\Psi(v) \cap \Psi(K)| \ge d^\circ(v) + x$. When both quantities are too small, the following fact implies that nodes must have slack from a low degree.

▶ **Fact 30.** *If $\overline{a}_K, \widetilde{e}_v \le \overline{e}_K/4$, then $\Delta^2 - \widetilde{d}(v) > \overline{e}_K/2$.*

Proof. For all $v \in K$, we have $|K| = |N^2(v) \cap K| + a_v$ and $\Delta^2 = (\Delta^2 - \widetilde{d}(v)) + |N^2(v) \cap K| + \theta_v + e_v$, Equation (8) holds:

$$\Delta^2 - |K| = (\Delta^2 - \widetilde{d}(v)) + \theta_v + e_v - a_v = (\Delta^2 - \widetilde{d}(v)) + \widetilde{e}_v - \widetilde{a}_v \ . \tag{8}$$

Since this holds for all nodes, it also holds on average:

$$\Delta^2 - |K| \ge \theta_K + \overline{e}_K - \overline{a}_K \ . \tag{9}$$

We conclude by replacing Equation (9) in Equation (8):

$$\begin{aligned}
\Delta^2 - \widetilde{d}(v) &\geq (\Delta^2 - |K|) - \widetilde{e}_v && \text{(by Equation (8))}\\
&\geq \overline{e}_K - \overline{a}_K - \widetilde{e}_v && \text{(by Equation (9))}\\
&\geq \overline{e}_K/2\ . && \text{(because } \overline{a}_K, \widetilde{e}_v \leq \overline{e}_K/4\text{)}
\end{aligned}$$

◀

When one or both of the quantities are larger, we do a case analysis with four cases. Large anti-degree implies that the colorful matching provides slack w.h.p. Large external degree implies that GenerateSlack created slack at the beginning of the algorithm, w.h.p. A careful analysis allows to claim that sufficient slack is guaranteed to exist w.h.p.

Case 1 (high anti-degree, low external degree). If $\overline{a}_K > C\log n$ and $\overline{e}_K < 4C\log n$. We compute a colorful matching of size $|M| \geq 402\overline{a}_K$. Thus, all nodes have slack $|M| - 2\widetilde{a}_v \geq 2\overline{a}_K$, because $\widetilde{a}_v \leq 200\overline{a}_K$ for all inliers (Lemma 14). If $e_v \geq \overline{a}_K > C\log n$, then v receives slack $\Omega(e_v)$ from slack generation; hence it has $\Omega(\overline{a}_K + \overline{e}_K + \widetilde{e}_v)$ slack by Equation (7). Otherwise, if $\overline{a}_K > e_v$, it gets enough slack from the colorful matching.

Case 2 (high anti-degree and external degree). If $\overline{a}_K > C\log n$ and $\overline{e}_K \geq 4C\log n$. Similarly to case 1, nodes have slack $\overline{a}_K$. If $\overline{a}_K > \overline{e}_K/4$ or $\theta_v^{\mathsf{ext}} \geq \overline{e}_K/8$, then it has enough slack. Finally, if $e_v > \overline{e}_K/8 > \Omega(\log n)$, then v received $\Omega(e_v)$ slack from GenerateSlack; hence has slack $\Omega(\overline{a}_K + \overline{e}_K + \widetilde{e}_v)$. The only remaining possibility is that $\overline{a}_K, \widetilde{e}_v \leq \overline{e}_K/4$. Then, Fact 30 shows that $\Delta^2 - \widetilde{d}(v) \geq \overline{e}_K/2 \geq \Omega(\overline{a}_K + \overline{e}_K + \widetilde{e}_v)$ and we are done.

Case 3 (low anti-degree, high external degree). If $\overline{a}_K < C\log n$ and $\overline{e}_K > 4C\log n$. If $e_v > \overline{e}_K/8 \geq \Omega(\log n)$, then v has slack $\theta_v^{\mathsf{ext}} + \Omega(e_v) \geq \Omega(\overline{a}_K + \overline{e}_K + \widetilde{e}_v)$ from slack generation, so we are done. If $\theta_K^{\mathsf{ext}} > \overline{e}_K/8$, then again we are done. Otherwise, $\overline{a}_K, \widetilde{e}_v \leq \overline{e}_K/4$ and by Fact 30 we conclude that all nodes have enough degree slack.

Case 4 (low anti-degree and external degree). If $\overline{a}_K < C\log n$ and $\overline{e}_K < 4C\log n$. Since $K \in \mathcal{K}_{\mathsf{mod}}$, it must be that $\theta_K^{\mathsf{ext}} > 4C\log n$. If $\widetilde{e}_v$ is greater than $\theta_K^{\mathsf{ext}}/8$, then v has slack $\Omega(\widetilde{e}_v) \geq \Omega(\widetilde{e}_v + \theta_K^{\mathsf{ext}}) \geq \Omega(\overline{a}_K + \overline{e}_K + \widetilde{e}_v)$ and we are done. So we can assume $\overline{a}_K, \widetilde{e}_v < \theta_K^{\mathsf{ext}}/4$. We argue that the degree slack must be large. Similarly to Fact 30, we have

$$\begin{aligned}
\Delta^2 - \widetilde{d}(v) &\geq (\Delta^2 - |K|) - \widetilde{e}_v && \text{(by Equation (8))}\\
&\geq \theta_K^{\mathsf{ext}} - \overline{a}_K - \widetilde{e}_v && \text{(by Equation (9))}\\
&\geq \theta_K^{\mathsf{ext}}/2 \geq \Omega(\overline{a}_K + \overline{e}_K + \widetilde{e}_v)\ . && \text{(by assumption)}
\end{aligned}$$

C Random Broadcast in Almost-Cliques

In this section, we explain how nodes in an almost-clique can all learn $\Theta(\log^2 n)$ messages each originating from a different node in $O(1)$ rounds, as used in the proof of Lemma 27. We use the following broadcast primitive: Each node forwards along each outgoing edge a (independently) random message received.

▶ **Lemma 31** (Distance-2 Many-to-All Broadcast). *Let K be an almost-clique in G^2 and $S \subseteq K$ be a subset of k vertices such that each $x \in S$ has a message m_x. Suppose $\Delta \geq k\log n$ and $\Delta^2 \geq k^3\log n$. After four rounds of the broadcast primitive, every node in K received all messages $\{m_x\}_{x\in S}$, w.h.p.*

Proof. Let $u \in S$ and $v \in K$. Recall that u and v both have at least $(1-\varepsilon)\Delta^2$ d2-neighbors in K. Since $|K| \leq (1+\varepsilon)\Delta^2$, there are at least $(1-3\varepsilon)\Delta^2$ common d2-neighbors of u and v in K. Let $W \overset{\text{def}}{=} N^2(u) \cap N^2(v)$ be this set.

We attribute a unique "relay" to each node $w \in W$, connecting it to v. For each $w \in W$, let r_w be the common d1-neighbor of w and v of lowest ID. For each d1-neighbor r of v, let $W_r \subseteq W$ be the nodes of W for which r is the chosen relay to v. Assume $\varepsilon < 1/12$. Using that $|W_r| \leq \Delta$, and by a simple Markov-type argument, there are at least $(1-6\varepsilon)\Delta \geq \Delta/2$ d1-neighbors r of v for which $|W_r| \geq \Delta/2$. Let R be the set of those "heavy" relays.

After the first round, each d1-neighbor of u receives m_u. Consider some heavy relay $r \in R$. Each node $w \in W_r$ receives the message m_u from a d1-neighbor it shares with u with probability at least $1/k$, independently from other nodes. Thus, with probability at least $1 - \exp(-\Delta/(24k)) = 1 - 1/\operatorname{poly}(n)$, $\Delta/(4k)$ or more nodes in W_r receive m_u.

Assume at least $\Delta/(4k)$ nodes in W_r received m_u. Then, in the third round of the broadcast primitive, r fails to receive m_u with probability at most:

$$\left(1-\frac{1}{k}\right)^{\Delta/(4k)} \leq e^{-\Delta/(4k^2)} .$$

When $\Delta/(4k^2) \geq 1/2$, this probability is bounded by $e^{-1/2} \leq 2/3$. In that case, $\Delta/6$ or more nodes in R should receive m_u in expectation, and so at least $\Delta/12$ heavy relays receive m_u w.p. $1-\exp(-\Delta/72)$. The probability that those relays all fail in sending m_u to v in the fourth round of the broadcast primitive is at most $(1-1/k)^{\Delta/72} \leq \exp(-\Delta/(72k)) = 1/\operatorname{poly}(n)$.

If $\Delta/(4k^2) \leq 1/2$, then $e^{-\Delta/(4k^2)} \leq 1-\Delta/(8k^2)$. In expectation, at least $\Delta^2/(16k^2)$ heavy relays receive m_u, and $\Delta^2/(32k^2)$ of them receive m_u w.h.p. All those relays fail to send m_u to v with probability at most $(1-1/k)^{\Delta^2/(32k^2)} \leq \exp\bigl(-\Delta^2/(32k^3)\bigr) = 1/\operatorname{poly}(n)$. ◀

In particular, this allows us to learn all colors remaining in the clique palette, because at this step of the algorithm, only $\operatorname{poly}\log n$ colors should remain available in $\Psi(K)$. If not, we learn nonetheless a set of $\operatorname{poly}\log n$ colors from $\Psi(K)$ which will act as a replacement.

▶ **Lemma 32.** *Assume $\Delta \geq \Omega(\log^{3.5} n)$. There is an $O(1)$-round algorithm (with $O(\log n)$ bandwidth) such that, in each almost-clique K, either*
1. *all nodes $v \in K$ learn all colors in $\Psi(K)$, or*
2. *all nodes learn a set $D \subseteq \Psi(K)$ of $\Theta(\log^2 n)$ colors.*

Proof. If $|\Psi(K)| \leq O(\log^2 n)$, then let $D \overset{\text{def}}{=} \Psi(K)$. Otherwise, if $|\Psi(K)| \geq \Omega(\log^2 n)$, let D be the $\Theta(\log^2 n)$ first colors of $\Psi(K)$. Recall that all nodes can learn $|\Psi(K)|$ in $O(1)$ rounds (Lemma 20); hence, nodes know in which of the two case they are.

Assign indices of $[|\Psi(K)|]$ to arbitrary nodes $u_1, \ldots, u_{|D|}$ of K (with a BFS for instance). This is feasible because $|K| \geq \Delta^2/2 > \Theta(\log^2 n) = |D|$. Then, each u_i learns the i-th color of D in $O(1)$ rounds (Lemma 29). Each u_i then crafts a message m_i containing that color and distributes it to all nodes of K by Many-to-All broadcast. By assumption, there are only $|D| = O(\log^2 n)$ messages, and since $|K| \geq \Delta^2/2 \geq \Theta(\log^7 n) = |D|^3 \times \Theta(\log n)$, we meet the requirements of Lemma 31. Thus, in $O(1)$ rounds, all the nodes in K know all colors of D. ◀

This immediately leads to a good approximation $L(v) = D \cap \Psi(v) = D \setminus \mathcal{C}(N^2(v) \setminus K)$ for $v \in K \in \mathcal{K}_{\mathsf{very}}$ after synchronized color trial. Suppose that we are in the second case of Lemma 32, i.e., nodes learn a set $D \subseteq \Psi(K)$ of $\Theta(\log^2 n)$ colors. Since $K \in \mathcal{K}_{\mathsf{very}}$, the average node has few external connections, $\overline{e}_K + \theta_K^{\mathsf{ext}} = O(\log n)$ (Definition 22). Moreover, because uncolored nodes are all inliers, $e_v = O(\overline{e}_K + \theta_K^{\mathsf{ext}})$ (Equation (6)). Finally, node v loses at most one color in D per external neighbor, hence

$$|D \cap \Psi(v)| \geq |D| - e_v \geq \Theta(\log^2 n) - O(\overline{e}_K + \theta_K^{\mathsf{ext}}) \geq \Theta(\log^2 n) \geq d^\circ(v) + 1 \ .$$

The last inequality holds because, at this point of the algorithm, nodes have $d^\circ(v) = O(\log n)$.

D Proof of Theorem 1

In this section, we put together all results from other sections to prove our main theorem. Only missing are the technicalities of reducing the bandwidth from $O(\log^2 n)$ to $O(\log n)$, which we tackle in the full version [17, Section 7].

Proof. Let $C > 0$ be some large universal constant. By Lemma 7, computing the almost-clique decomposition $V_{\mathsf{sparse}}, K_1, \dots, K_k$ of G^2 takes $O(1)$ rounds of CONGEST (Step 1). Generating slack and coloring V_{sparse} takes $O(\log^* n)$ rounds (Propositions 5 and 11). Putting together all results from Section 4, we prove the proposition stated earlier, which implies Theorem 1.

▶ **Proposition 13** (Coloring Dense Nodes). *After* GenerateSlack *and coloring sparse nodes, there is a $O(\log^6 \log n)$-round randomized algorithm for completing a $\Delta^2 + 1$-coloring of the dense nodes, w.h.p.*

Steps 4, 5 & 6. By Proposition 17, we compute a colorful matching of size $402\overline{a}_K$ in $O(1)$ rounds, in all almost-cliques with $\overline{a}_K > C \log n$. By Lemma 14, we can compute sets O_K and $I_K = K \setminus O_K$ in all almost cliques, such that all $v \in I_K$ verify Equation (6) and $|I_K| > (1-5/100)|K|$. Let H_1 be the subgraph of G^2 induced by $\bigcup_K O_K$. Note that each $v \in O_K$, for some almost-clique K, has at least $(1-5/100)|K| - \varepsilon\Delta^2 > (1-5/100)(1-\varepsilon)\Delta^2 - \varepsilon\Delta^2 \geq \Delta^2/2$ neighbors in I_K, for ε small enough. Hence, each outlier has $\Delta^2/2$ slack in H_1 and can thus be colored in $O(\log^* n)$ rounds by Proposition 11.

Step 7 & 8. Order nodes of I_K with a BFS. By Lemma 28, w.h.p., the i-th node of I_K can learn $\pi(i)$, where π is a uniformly random permutation of $[|I_K|]$. By Lemma 29, each node can learn, thus try, the i-th color of $\Psi(K)$ (if it exists). With high probability, by Lemma 18, each almost-clique K has $O(\overline{a}_K + \overline{e}_K + \log n)$ uncolored nodes. This implies, the uncolored degree of a dense node $v \in K$ is $O(e_v + \overline{a}_K + \overline{e}_K + \log n)$. In particular, if $v \in K \in \mathcal{K}_{\mathsf{very}}$ (Definition 22), it has uncolored degree $d^\circ(v) \leq O(\log n)$. Since Step 8 intend to reduce the uncolored degree to $O(\log n)$, we can focus on moderately dense almost-cliques. Let $H_2 = \bigcup_{K \in \mathcal{K}_{\mathsf{mod}}} K$. The following fact shows conditions of Lemma 12 are verified by the sampler of Lemma 24.

▶ **Fact 33.** *There exists a universal constant $\alpha > 0$ such that $s(v) \geq \alpha \cdot b(v)$ for all $v \in H_2$, where $b(v) = \widetilde{e}_v + |K^\circ|$ and $s(v) \geq \Omega(\overline{a}_K + \overline{e}_K + \widetilde{e}_v)$ from Lemma 24 such that* $\Pr(\mathsf{C}_v = c) \leq \frac{1}{d^\circ(v) + s(v)}$.

Proof of Fact 33. Let K be the almost-clique of v. The quantity $b(v)$ only requires $O(1)$ rounds to compute: To compute its pseudo-external degree $\widetilde{e}_v$, a node only needs to receive from each of its direct neighbors $u \in N_G(v)$ the value $|(N_G(u) \cup \{u\}) \setminus K|$; for $|K^\circ|$, the number of uncolored nodes in K, a simple BFS within K suffices to count $|K^\circ|$ and broadcast it to the whole almost-clique.

We now show $b(v)$ satisfies the hypotheses. After SCT, by Lemma 18, at most $O(\overline{e}_K+\overline{a}_K+\log n)$ nodes are left uncolored in K, so $b(v) \in O(\widetilde{e}_v+\overline{e}_K+\overline{a}_K+\log n)$. By Lemma 24, there exist $s(v) \in \Omega(\overline{a}_K+\overline{e}_K+\widetilde{e}_v)$ s.t. Equation (5) holds. If $b(v) \leq C\log n$, then $d^\circ(v) < C\log n$, hence the uncolored degree is already $O(\log n)$. Otherwise, when $b(v) \geq C\log n$, it must be that $b(v) \in \Theta(\widetilde{e}_v+\overline{e}_K+\overline{a}_K)$, and so, there exists a universal constant α s.t. $s(v) \geq \alpha\cdot b(v)$. ◀

Hence, we can use the sampling algorithm of Lemma 24 to run SliceColor (Lemma 12) in H_2. Therefore, in $O(\log\log n)$ rounds, we produce a coloring and a partition $L_1,\dots,L_\ell$ of uncolored nodes in H_2 such that the maximum uncolored degree of each $G[L_i]$ for $i \in [\ell]$ is $O(\log n)$. We also define $L_0 = \bigcup_{K\in\mathcal{K}_{\mathsf{very}}} K$ which has maximum uncolored degree $O(\log n)$ after the synchronized color trial.

Steps 10 & 11. We go through layers $L_0, L_1, \dots, L_\ell$ sequentially. In L_0, nodes learn lists of deg +1 colors from their palette by Lemma 27. In each L_i for $i \in [\ell]$, nodes are moderately dense, hence learn their palette from sampling by Lemma 26. Solve each of these deg +1-list-coloring instance of Proposition 13 with the small degree algorithm of Proposition 2. Since learning palettes takes $O(\log\log n)$ and each deg +1-list-coloring instance is solved in $O(\log^5\log n)$, the total round complexity of this step is $O(\log^6\log n)$, which dominates the complexity of the algorithm. ◀

Distributed Certification for Classes of Dense Graphs

Pierre Fraigniaud
IRIF, Université Paris Cité, CNRS, France

Frédéric Mazoit
LaBRI, Université de Bordeaux, France

Pedro Montealegre
Facultad de Ingeniería y Ciencias, Universidad Adolfo Ibáñez, Santiago, Chile

Ivan Rapaport
DIM-CMM (UMI 2807 CNRS), Universidad de Chile, Santiago, Chile

Ioan Todinca
LIFO, Université d'Orléans and INSA Centre-Val de Loire, France

Abstract

A *proof-labeling scheme* (PLS) for a boolean predicate Π on labeled graphs is a mechanism used for certifying the legality with respect to Π of global network states in a distributed manner. In a PLS, a *certificate* is assigned to each processing node of the network, and the nodes are in charge of checking that the collection of certificates forms a global proof that the system is in a correct state, by exchanging the certificates once, between neighbors only. The main measure of complexity is the *size* of the certificates. Many PLSs have been designed for certifying specific predicates, including cycle-freeness, minimum-weight spanning tree, planarity, etc.

In 2021, a breakthrough has been obtained, as a "meta-theorem" stating that a large set of properties have compact PLSs in a large class of networks. Namely, for every MSO_2 property Π on labeled graphs, there exists a PLS for Π with $O(\log n)$-bit certificates for all graphs of bounded *tree-depth*. This result has been extended to the larger class of graphs with bounded *tree-width*, using certificates on $O(\log^2 n)$ bits.

We extend this result even further, to the larger class of graphs with bounded *clique-width*, which, as opposed to the other two aforementioned classes, includes dense graphs. We show that, for every MSO_1 property Π on labeled graphs, there exists a PLS for Π with $O(\log^2 n)$-bit certificates for all graphs of bounded clique-width. As a consequence, certifying families of graphs such as distance-hereditary graphs and (induced) P_4-free graphs (a.k.a., cographs) can be done using a PLS with $O(\log^2 n)$-bit certificates, merely because each of these two classes can be specified in MSO_1. In fact, we show that certifying P_4-free graphs can be done with certificates on $O(\log n)$ bits only. This is in contrast to the class of C_4-free graphs (which does not have bounded clique-width) which requires $\tilde{\Omega}(\sqrt{n})$-bit certificates.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases CONGEST, Proof Labelling Schemes, clique-width, MSO

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.20

Related Version *Full Version*: https://arxiv.org/abs/2307.14292 [35]

Funding *Pierre Fraigniaud*: Additional support for ANR projects QuData and DUCAT.
Pedro Montealegre: This work was supported by Centro de Modelamiento Matemático (CMM), FB210005, BASAL funds for centers of excellence from ANID-Chile, and ANID-FONDECYT 1230599
Ivan Rapaport: This work was supported by Centro de Modelamiento Matemático (CMM), FB210005, BASAL funds for centers of excellence from ANID-Chile, and ANID-FONDECYT 1220142.

1 Introduction

Checking whether a distributed system is in a legal global state with respect to some boolean predicate occurs in several domains of distributed computing, including the following.

- Fault-tolerance: the occurrence of faults may turn the system into an illegal state that needs to be detected for allowing the system to return to a legal state.
- The use of subroutines as black boxes: some of these subroutines may contain bugs, and produce incorrect outputs that need to be checked before use in the protocol calling the subroutines.
- Algorithm design for specific classes of systems: an algorithm dedicated to some specific class of networks (e.g., algorithms for trees, or for planar networks) may cause deadlocks or live-locks whenever running on a network outside the class. The membership to the class needs to be checked before running the algorithm.

In all three cases above, the checking procedure may be impossible to implement without significant communication overhead. A typical example is bipartiteness, whether it be applied to the network itself, or to an overlay network produced by some subroutine.

1.1 Proof-Labeling Schemes

Proof-labeling scheme (PLS) [46] is a popular mechanisms enabling to certify correctness w.r.t. predicates involving some global property, like bipartiteness. A PLS involves a *prover* and a *verifier*. The prover has access to the global state of the network (including its structure), and has unlimited computational power. It assigns *certificates* to the nodes. The verifier is a distributed algorithm running at each node, performing in a single round, which consists for each node to send its certificate to its neighbors. Upon reception of the certificates of its neighbors, every node performs some local computation and outputs *accept* or *reject*. To be correct, a PLS for a predicate Π must satisfy:

$$\begin{array}{c}\text{the global state of the network satisfies } \Pi \\ \Updownarrow \\ \text{the prover can assign certificates such that the verifier accepts at all nodes.}\end{array}$$

For instance, for bipartiteness, the prover assigns a color 0 or 1 to the nodes, and each node verifies that its color is 0 or 1, and is different from the color of each of its neighbors. If the network is bipartite then the prover can properly 2-color the nodes such that they all accept, and if the network is not bipartite then, for every 2-coloring of the nodes, some of them reject as this coloring cannot be proper.

The PLS certification mechanism has several desirable features. First, if the certificates are small then the verification is performed efficiently, in a single round consisting merely of an exchange of a small message between every pair of adjacent nodes. As a consequence, verification can be performed regularly and frequently without causing significant communication overhead. Second, if the network state does not satisfy the predicate, then at least one node rejects. Such a node can raise an alarm or launch a recovery procedure for allowing the system to return to a correct state, or can stop a program running in an environment for which it was not designed. Third, the prover is an abstraction, for the certificates can be computed offline, either by the nodes themselves in a distributed manner, or by the system provider in a centralized manner. For instance, a protocol constructing an overlay network that is supposed to be bipartite, may properly 2-color the overlay for certifying its bipartiteness. It follows from their features that PLSs are versatile certification mechanisms that are also quite efficient whenever the certificates for legal instances are small.

Many PLSs have been designed for certifying specific predicates on labeled graphs, including cycle-freeness [46], minimum-weight spanning tree (MST) [45], planarity [31], bounded genus [25], H-minor-freeness for small H [6], etc. In 2021, a breakthrough has been obtained, as a "meta-theorem" stating that a large set of properties have compact PLSs in a large class of networks (see [27]). Namely, for every MSO$_2$ property[1] Π, there exists a PLS for Π with $O(\log n)$-bit certificates for all graphs of bounded *tree-depth*, where the tree-depth of a graph intuitively measures how far it is from being a star. This result has been extended to the larger class of graphs with bounded *tree-width* (see [36]), using certificates on $O(\log^2 n)$ bits, where the tree-width of a graph intuitively measures how far it is from being a tree. Although the class of all graphs with bounded tree-width includes many common graph families such as trees, series-parallel graphs, outerplanar graphs, etc., it does not contain families of *dense* graphs. In this paper, we focus on the families of graphs with bounded *clique-width*, which include families of dense graphs.

1.2 Clique-Width

Intuitively, the definition of clique-width is based on a "programming language" for constructing graphs, using only the following four instructions (see [16] for more details):

- Creation of a new vertex v with some color i, denoted by $\mathsf{color}(v, i)$;
- Disjoint union of two colored graphs G and H, denoted by $G \parallel H$;
- Joining by an edge every vertex colored i to every vertex colored $j \neq i$, denoted by $i \bowtie j$;
- Recolor i into color j, denoted by $\mathsf{recolor}(i, j)$.

For instance, the n-node clique can be constructed by creating a first node with color blue, and then repeating $n-1$ times the following: (1) the creation of a new node, with color red, (2) joining red and blue, and (3) recoloring red into blue. Therefore, cliques can be constructed by using two colors only. Similarly, trees can be constructed with three colors only. This can be proved by induction. The induction statement is that, for every tree T, every vertex r of T, and every two colors $c_1, c_2 \in \{\text{blue, red, green}\}$, T can be constructed with colors blue, red, and green such that r is eventually colored c_1, and every other vertex is colored c_2. The statement is trivial for the single-node tree. Let T be a tree with at least two nodes, let r be one of its vertices, and let c_1, c_2 be two colors. Given an arbitrary neighbor s of r, removing the edge $\{r, s\}$ results in two trees T_r and T_s. By induction, construct T_r and T_s separately so that r (resp., s) is eventually colored c_1 (resp., c_2) and all the other nodes of T_r and T_s are colored $c_3 \notin \{c_1, c_2\}$. Then form the graph $T_r \parallel T_s$, and, in this graph, join colors c_1 and c_2, and recolor c_3 into c_2.

The clique-width of a graph G, denoted by $\mathsf{cw}(G)$, is the smallest $k \geq 0$ such that G can be constructed by using k colors. For instance, $\mathsf{cw}(K_n) \leq 2$ for every $n \geq 1$, and, for every tree T, $\mathsf{cw}(T) \leq 3$. A family of graphs has bounded clique-width if there exists $k \geq 0$ such that, for every graph G in the family, $\mathsf{cw}(G) \leq k$. Any graph family with bounded tree-depth or bounded tree-width has bounded clique-width [12, 48]. However, there are important graph families with unbounded tree-width (and therefore unbounded tree-depth) that have bounded clique-width. Typical examples (see [17]) are cliques (i.e., complete

[1] Monadic second-order logic (MSO) is the fragment of second-order logic where the second-order quantification is limited to quantification over sets. MSO$_1$ refers to MSO on graphs with quantification over sets of vertices, whereas MSO$_2$ refers to MSO on graphs with quantification over sets of vertices and sets of edges.

graphs), P_4-free graphs (i.e., graphs excluding a path on four vertices as an induced subgraph, a.k.a., cographs), and distance hereditary graphs (the distances in any connected induced subgraph are the same as they are in the original graph).

Many NP-hard optimization problems can be solved efficiently by dynamic programming in the family of graphs with bounded clique-width. In fact, every MSO_1 property on graphs has a linear-time algorithm for graphs of bounded clique-width [16]. In this paper we show a similar form of "meta-theorem", regarding the size of certificates of PLS for monadic second-order properties of graphs with bounded clique-width.

1.3 Our Results

Our main result is the following. Recall that a labeled graph is a pair (G, ℓ), where G is a graph, and $\ell : V(G) \to \{0,1\}^\star$ is a function assigning a label to every node in G.

▶ **Theorem 1.** *Let k be a non-negative integer, and let Π be an MSO_1 property on node-labeled graphs with constant-size labels. There exists a PLS certifying Π for labeled graphs with clique-width at most k, using $O(\log^2 n)$-bit certificates on n-node graphs.*

The same way several NP-hard problems become solvable in polynomial time in graphs of bounded clique-width, Theorem 1 implies that several predicates for which every PLS has certificates of polynomial size in arbitrary graphs have a PLS with certificates of polylogarithmic size on graphs with bounded clique-width. This is for instance the case of non-3-colorability (which is a MSO_1 predicate), for which every PLS has certificates of size $\tilde{\Omega}(n^2)$ bits in arbitrary graphs [42]. Theorem 1 implies that non-3-colorability has a PLS with certificates on $O(\log^2 n)$ bits in graphs with bounded clique-width, and therefore in graphs of bounded tree-width, cographs, distance-hereditary graphs, etc. This of course is extended to non-k-colorability, as well as other problems definable in MSO_1 such as detecting whether the input graph does not contain a fixed subgraph H as a subgraph, induced subgraph, minor, etc.

In fact, Theorem 1 can be extended to properties including certifying solutions to maximization or minimization problems whose admissible solutions are defined by MSO_1 properties. For instance maximum independent set, minimum vertex cover, minimum dominating set, etc.

In the proof of Theorem 1, we provide a PLS that constructs a particular decomposition using at most $k \cdot 2^{k-1}$ colors (the clique-width of the decomposition). It is through that decomposition that the PLS certifies that the input graph satisfies Π.

An application of Theorem 1 is the certification of certain families of graphs. That is, given a graph family $\mathcal{F}$, designing a PLS for certifying the membership to $\mathcal{F}$. Interestingly, there are some graph classes $\mathcal{F}$ that are expressible in MSO_1 and, at the same time, have clique-width at most k. Theorem 1 provides a PLS for certifying the membership to $\mathcal{F}$ in such cases. Indeed, the PLS first tries to build a decomposition of clique-width at most $k \cdot 2^{k-1}$. If there is no such decomposition, then the input graph does not belong to $\mathcal{F}$. Otherwise, the PLS uses the decomposition to check the MSO_1 property that defines $\mathcal{F}$.

▶ **Corollary 2.** *Let k be a non-negative integer, and let $\mathcal{F}$ be graph family expressible in MSO_1 such that all graphs of the family have clique-width at most k. Membership to $\mathcal{F}$ can be certified with a PLS using $O(\log^2 n)$-bit certificates in n-node graphs.*

For instance, for every $k \geq 0$, the class of graphs with tree-width at most k can be certified with a PLS using $O(\log^2 n)$-bit certificates. Indeed, "tree-width at most k" is expressible in MSO_1, and the class of graphs with tree-width at most k forms a family with clique-width at most $3 \cdot 2^{k-1} + 1$ [12]. Another interesting application is the certification of P_4-free graphs. Indeed, "excluding P_4 as induced subgraph" is expressible in MSO_1, and P_4-free graphs form a family with clique-width at most 2 [17]. It follows that P_4-free graphs can be certified with a PLS using $O(\log^2 n)$-bit certificates. This is in contrast to the class of C_4-free graphs (i.e. graphs not containing a cycle on four vertices, whether it be as induced subgraph or merely subgraph), which requires certificates on $\tilde{\Omega}(\sqrt{n})$ bits [20]. In fact, in the case of cographs, the techniques in the proof of Theorem 1 can be adapted so that to save one log-factor, as stated below.

▶ **Theorem 3.** *The class of (induced) P_4-free graphs can be certified with a PLS using $O(\log n)$-bit certificates in n-node graphs.*

Note that there is a good reason for the huge gap in terms of certificate-size between P_4-free graphs and C_4-free graphs. The point is that, for any graph pattern H, the class of H-free graphs has bounded clique-width if and only if H is an induced subgraph of P_4 [19]. Therefore, C_4-free graphs (as well as triangle-free graphs) do not have bounded clique-width, as opposed to P_4-free graphs (and P_3-free graphs, which are merely cliques).

1.4 Related Work

Proof-Labeling Schemes (PLSs) have been introduced and thoroughly studied in [46]. Variants have been been considered in [42] and [34], which slightly differ from PLSs: the former allows each node to transfer no only its certificates, but also its state, and the latter restricts the power of the oracle, which is bounded to produce certificates independent of the IDs assigned to the nodes. All these forms of distributed certifications have been extended in various directions, including tradeoffs between the size of the certificates and the number of rounds of the verification protocol [30], PLSs with computationally restricted provers [24], randomized PLSs [38], quantum PLSs [33], PLSs rejecting at more nodes whenever the global state is "far" from being correct [28], PLSs using global certificates in addition to the local ones [32], and several hierarchies of certification mechanisms, including games between a prover and a disprover [1, 29], interactive protocols [18, 44, 47], and even recently zero-knowledge distributed certification [3], and distributed quantum interactive protocols [41].

All the aforementioned distributed certification mechanisms have been used for certifying a wide variety of global system states, including MST [45], routing tables [2], and a plethora of (approximated) solutions to optimization problems [11, 23]. A vast literature has also been dedicated to certifying membership to graph classes, including cycle-freeness [46], planarity [31], bounded genus [25], absence of symmetry [42], H-minor-freeness for small H [6], etc. In 2021, a breakthrough has been obtained, as a "meta-theorem" stating that, for every MSO_2 property Π, there exists a PLS for Π with $O(\log n)$-bit certificates for all graphs of bounded *tree-depth* [27]. This result has been extended to the larger class of graphs with bounded *tree-width*, using certificates on $O(\log^2 n)$ bits [36]. To our knowledge, this is the largest class of graphs, and the largest class of boolean predicates on graphs for which it is known that PLSs with polylogarithmic certificates exist.

The class of H-free graphs (i.e., the absence of H as a subgraph), for a given fixed graph H, has attracted lot of attention in the distributed setting, mostly in the CONGEST model. Two main approaches have been considered. One, called distributed property testing, aims at deciding between the case where the input graph is H-free, and the case where the input graph is "far" from being H-free (see, e.g., [7, 9, 26, 39]). In this setting, the objective is to design (randomized) algorithms performing in a constant number of rounds. Such algorithms have been designed for small graphs H, but it is not known whether there is a distributed algorithm for testing K_5-freeness in a constant number of rounds. The other approach aims at designing algorithms deciding H-freeness performing in a small number of rounds. For instance, it is known that deciding C_4-freeness can be done in $\tilde{O}(\sqrt{n})$ rounds, and this is optimal [20]. The $\tilde{\Omega}(\sqrt{n})$-round lower bounds for C_4-freeness also holds for deciding C_{2k}-freeness, for every $k \geq 4$. Nevertheless, the best known algorithm performs in essentially $\tilde{O}(n^{1-\Theta(1/k^2)})$ rounds [22], even if faster algorithms exists for $k = 2, 3, 4, 5$, running in $\tilde{O}(n^{1-\Theta(1/k)})$ rounds [10, 21]. Deciding P_k-freeness (as subgraph) can be done efficiently for all $k \geq 0$ [37]. However, this is not the case of deciding the absence of an *induced* P_k, and no efficient algorithms are known apart for the trivial cases $k = 1, 2, 3$. The first non-trivial case is deciding cographs, i.e., P_4-freeness (as induced subgraph).

The terminology *meta-theorem* is used in logic to refer to a statement about a formal system proven in a language used to describe another language. In the study of graph algorithms, Courcelle's theorem [13] is often referred to as a meta-theorem. It says that every graph property definable in the monadic second-order logic MSO_2 of graphs can be decided in linear time on graphs of bounded treewidth. This theorem was extended to clique-width, but for a smaller set of graph properties. Specifically, every graph property definable in the monadic second-order logic MSO_1 of graphs can be decided in linear-time on graphs of bounded clique-width [16]. Note that the classes of languages in MSO_1 and MSO_2 include languages that are NP-hard to decide (e.g., 3-colorability and Hamiltonicity, respectively). We remind that MSO_2 is as an extension of MSO_1 which also allows quantification on sets of edges – see Footnote 1 for a short description, or [14] for full details. Some graph properties, e.g., Hamiltonicity, are expressible in MSO_2 but not in MSO_1, nevertheless MSO_1 captures a large set of properties, including many classical NP-hard problems as explained above. Eventually, we emphasize again that, when comparing the two most famous meta-theorems, (1) MSO_2 *properties are decidable in linear time on bounded treewidth graphs* vs. (2) MSO_1 *properties are decidable in linear time on bounded clique-width graphs*, the former concerns a larger class of properties, but the latter concerns larger classes of graphs.

2 Models

In this section, we recall the main concepts used in this paper, including proof-labeling scheme, and cographs.

2.1 Proof-Labeling Schemes for MSO Properties

For a fixed integral parameter $\lambda \geq 0$, we consider vertex-labeled graphs (G, ℓ), where $G = (V, E)$ is a connected simple n-node graph, and $\ell : V \to \{0, \dots, \lambda - 1\}$. The label may indicate a solution to an optimization problem, e.g., a minimum dominating set ($\ell(v) = 0$ or 1 depending on whether v is in the set or not), a λ-coloring, an independent set, etc. A labeling may also encode global overlay structures such as spanning trees or spanners, in bounded-degree graphs or in graphs provided by a distance-2 k-coloring, for $k = O(1)$. In the context of distributed computing in networks, nodes are assumed to be assigned distinct identifiers (ID) in $[1, n^c]$ for some $c \geq 1$, so that IDs can be stored on $O(\log n)$ bits. The identifier of a node v is denoted by $\mathsf{id}(v)$. We denote by $N_G(v)$ the set of neighbors of node v in a graph G, and we let $N_G[v] = N_G(v) \cup \{v\}$ be the closed neighborhood of v.

Given a boolean predicate Π on vertex-labeled graphs, a *proof-labeling scheme* (PLS) for Π is a prover-verifier pair. The *prover* is a non-trustable computationally unbounded oracle. Given a vertex-labeled graph (G, ℓ) with ID-assignment id, the prover assigns a *certificate* $c(v)$ to every node v of G. The *verifier* is a distributed algorithm running at every node v of G. It performs a single round of communication consisting of sending $c(v)$ to all neighboring nodes $w \in N_G(v)$, and receiving the certificates of all neighbors. Given $\mathsf{id}(v)$, $\ell(v)$, and $\{c(w) : w \in N_G[v]\}$, every node v outputs *accept* or *reject*. A PLS is correct if the following two conditions are satisfied:

- Completeness: If (G, ℓ) satisfies Π then the prover can assign certificates to the nodes such that the verifier accepts at all nodes;
- Soundness: If (G, ℓ) does not satisfy Π then, for every certificate assignment to the nodes by the prover, the verifier rejects in at least one node.

The main parameter measuring the quality of a PLS is the *size* (i.e., number of bits) of the certificates assigned by the prover to each node of vertex-labeled graphs satisfying the predicate, and leading all nodes to accept.

MSO Predicates. We focus on predicates expressible in MSO_1. Recall that MSO_1 is the fragment of monadic second-order (MSO) logic on (vertex-labeled) graphs that allows quantification on vertices and on sets of (labeled) vertices, and uses the adjacency predicate (adj). For instance non 3-colorability is in MSO_1. Indeed, for every graph $G = (V, E)$, it can be expressed as: for all $A, B, C \subseteq V$, if $A \cup B \cup C = V$ and $A \cap B = A \cap C = B \cap C = \varnothing$ then

$$\exists (u, v) \in (A \times A) \cup (B \times B) \cup (C \times C) : (u \neq v) \wedge \mathsf{adj}(u, v).$$

We shall show that, although some MSO_1 predicates, like non-3-colorability, require certificates on $\tilde{\Omega}(n^2)$ bits in n-node graphs (see [42]), PLSs with certificates of polylogarithmic size can be designed for all MSO_1 predicates in a rich class of graphs, namely all graphs with bounded clique-width.

2.2 Cographs and Cotrees

We conclude this section by introducing a graph class that plays an important role in this paper. Recall that a graph is a cograph (see, e.g., [8]) if it can be constructed by a sequence of parallel operations (disjoint union of two vertex-disjoint graphs) and join operations (connecting two vertex-disjoint graphs G and H by a complete bipartite graphs between $V(G)$ and $V(H)$). Therefore, by definition, cographs have clique-width 2. In particular, cliques are cographs.

It is known [8] that cographs capture precisely the class of induced P_4-free graphs. We shall show that, as opposed to C_4-free graphs, which require $\tilde{\Omega}(\sqrt{n})$-bit certificates to be certified by a PLS [20][2], $O(\log n)$-bit certificates are sufficient for certifying P_4-free graphs. This result is of interest on its own, but proving this result will also play the role of a warmup before establishing our general result about graphs with bounded clique-width. Note that the class of P_4-free graphs (i.e., cographs) can be specified by an MSO_1 formula. Roughly, the formula states that if there exists four vertices v_1, v_2, v_3, v_4 such that $\mathsf{adj}(v_i, v_{i+1})$ for $i = 1, 2, 3$, then $\mathsf{adj}(v_1, v_3) \vee \mathsf{adj}(v_1, v_4) \vee \mathsf{adj}(v_2, v_4)$. C_4-freeness could be expressed in MSO_1 as well. However, P_4-free graphs have clique-width 2 whereas C_4-free graphs have unbounded clique-width – this is because there are $2^{\Omega(n\sqrt{n})}$ different C_4-free graphs of size n, but only $2^{\mathcal{O}(n \log n)}$ n-vertex graphs of bounded clique-width.

[2] The lower bound in [20] is expressed for the CONGEST and Broadcast Congested Clique models, but it extends directly to PLSs since Set-Disjointness has non-deterministic communication complexity $\Omega(N)$ on N-bit inputs.

Given a cograph G, there is actually a canonic way of constructing G by a sequence of parallel and join operations [8]. As explained before, this construction can be described as a tree T whose leaves are the vertices of G, and whose internal nodes are labeled $\|$ or $\bowtie$. This tree is called a *cotree*, and will be used for our PLS.

3 Overview of our Techniques

The objective of this section is to provide the reader with a general idea of our proof-labeling scheme. For a comprehensive description we refer to the full version of this article [35]. Our construction bears some similarities with the approach used in [36] for the certification of MSO_2 properties on graphs of bounded tree-width, with certificates of size $O(\log^2 n)$ bits. However, extending this approach to a proof-labeling scheme for graphs with bounded clique-width requires to overcome several significant obstacles. We therefore start by summarizing the main tools used for the certification of MSO_2 properties on graphs of bounded tree-width (see Section 3.1), and then proceed with the description of the new tools required for extending the result to graphs of bounded clique-width, to the cost of reducing the class of certified properties from MSO_2 to MSO_1 (see Sections 3.2-3.6).

3.1 Certifying MSO_2 Properties in Graphs of Bounded Tree-Width

Recall that a tree-decomposition of a graph G is a tree T where each node x of T, also called *bag*, is a subset of $V(G)$, satisfying the following three conditions: (1) for every vertex $v \in V(G)$ there is a bag $x \in V(T)$ that contains v, (2) for every edge $\{u, v\} \in E(G)$, there is a bag x containing both its endpoints, and (3) for each vertex $v \in V(G)$, the set of bags that contains v forms a (connected) subtree of T. Let Π be an MSO_2 property, and let T be a tree-decomposition of the graph G. The proof-labeling scheme aims at providing each vertex with sufficient information for certifying the correctness of T, as well as the fact that G satisfies Π. To do so, the certificate of each vertex is divided into two parts, one called *main messages*, and the other called *auxiliary messages*.

Main messages. The main message of a node v is a sequence seq_v representing a path of bags in T that connects a leaf with the root, such that v is contained in at least one bag of seq_v. For each bag $x \in \mathrm{seq}_v$, the main message includes, roughly: the set of vertices contained in x, the identifier of a vertex ℓ_x in x, called the *leader* of x, and a data structure c_x used to verify the MSO_2 property Π on G. The leader ℓ_x of x is chosen arbitrarily among the vertices of x that are adjacent to a vertex u belonging to the parent bag $p(x)$ of x in T. The vertex u is said to be *responsible* for x in $p(x)$. Let us assume the following consistency condition: for every bag x of T, every vertex in x received the same information about all the bags from x to the root of T. Under the promise that the consistency condition holds, it is possible to show that the vertices can collectively verify that T is indeed a tree-decomposition of G, and that G satisfies Π.

Auxiliary messages. The role of the auxiliary messages is precisely to check the above consistency condition. For each bag x, let τ_x be a Steiner tree (i.e. a minimal tree connecting a set of vertices denoted *terminals*) in G rooted at the leader ℓ_x, with all the nodes of x as terminals. Every vertex in τ_x receives an auxiliary message containing the certification of τ_x (each vertex of τ_x receives the identifier of a root, of its parent and the distance to the root), and a copy of the information about x given to the nodes in the bag x, through their main messages. By using the auxiliary messages, the leader ℓ_x can verify whether the subgraph $G[x]$ of G induced by the union of the bags in $T[x]$ satisfies Π, where $T[x]$ the subtree of T containing x and all its descendants. Specifically, this verification is performed by simulating the dynamic programming algorithm in Courcelle's Theorem [13] as in the version of Boire, Parker and Tovey [5]. This uses a constant-size data structure c_x stored in the auxiliary messages that "encodes" the predicate $\Pi(G[x])$. Its correctness can be verified by a composition of the values c_y for each child y of x in T. The tree τ_x is actually used to transfer the information about c_y from the node ℓ_y in x responsible for y, to the leader ℓ_x.

Certificate size. If T is of depth d, then the main messages are of size $O(d \log n)$ bits. Crucially, for every graph G, there is a tree-decomposition T satisfying that, for every bag x, there is a Steiner tree τ_x completely contained in $G[x]$. Such a decomposition is called *coherent* in [36] (Lemma 3). It follows that every node participates in a Steiner tree with at most d bags, which implies that the auxiliary messages can be encoded in $O(d \log n)$ bits. Thanks to a construction by Bodlaender [4], it is possible to choose a coherent tree-decomposition with depth $d = O(\log n)$, up to increasing the sizes of the bags by a constant factor only. It follows that the certificates are of size $O(\log^2 n)$ bits.

Our construction also follows the general structure described above. However, each element of this construction has to be adapted in a highly non-trivial way. Indeed, the grammar of clique-with, and the related structure of NLC decomposition, differ in several significant ways from the grammar of tree-width. The rest of the section is dedicated to providing the reader with a rough idea of how this can be done.

3.2 Clique-Width and NLC-Width

First, instead of working with clique-width, it is actually more convenient to work with the NLC-width, where NLC stands for *node-label controlled.* Every graph of clique-width at most k has NLC-width at most k, and every graph of NLC-width at most k has clique-width at most $2k$ [43]. As clique-width, NLC-width can be viewed as the following grammar for constructing graphs, bearing similarities with the grammar for clique-width:

- Creation of a new vertex v with color $i \in \mathbb{N}$, denoted by $\mathsf{newVertex}_i$;
- Given a set S of ordered pairs of colors, and an ordered pair (G, H) of vertex-disjoint colored graphs, create a new graph as the union of G and H, then join by an edge every vertex colored i of G to every vertex colored j of H, for all $(i, j) \in S$; this operation is denoted by $G \bowtie_S H$;
- Recolor the graph, denoted by $\mathsf{recolor}_R$ where $R : \mathbb{N} \to \mathbb{N}$ is any function.

If $k \geq 1$ colors are used, a recoloring function R is a function $R : [k] \to [k]$. When R is used, for every $i \in [k]$, vertices with color i are recolored $R(i) \in [k]$ (all colors are treated simultaneously, in parallel). Note that the recoloring operation in the definition of clique-width is limited to functions R that preserve all colors but one. Note also that, for $S = \varnothing$, the operation $G \bowtie_S H$ is merely the same as $G \parallel H$ for clique-width. We therefore use $G \bowtie_\varnothing H$ or $G \parallel H$ indistinctly. The NLC-width of a graph G is the smallest number of colors such that G can be constructed using the operations above. It is denoted by $\mathsf{nlcw}(G)$. For instance, the n-node clique can be constructed by creating a first node v_1 with color 1, and then repeating, for all $i = 1, \dots, n-1$, (1) the creation of a new node v_{i+1}, with color 1 as well, and (2) applying $v_{i+1} \bowtie_{\{(1,1)\}} K_i$ to get the clique K_{i+1} on $i+1$ vertices. Therefore, cliques can be constructed by using one color only, i.e., $\mathsf{nlcw}(K_n) = 1$ for every $n \geq 1$.

NLC-decomposition. For every $k \geq 1$, the construction of a graph G with $\mathsf{nlcw}(G) \leq k$ can be described by a binary tree T, whose leaves are the (colored) vertices of G. In T, every internal node x has an identified left child x' and an identified right child x'', and is labeled by $\|$ or $\bowtie_S$ for some non-empty set $S \subseteq [k] \times [k]$. This label indicates the operation performed on the (left) graph G' with vertex-set equal to the leaves of the subtree $T_{x'}$ of T rooted at x', and the (right) graph G'' with vertex-set equal to the leaves of the subtree $T_{x''}$ of T rooted at x''. That is, node x corresponds to the operation $G_{x'} \parallel G_{x''}$ or $G_{x'} \bowtie_S G_{x''}$, depending on the label of x. In addition to its label ($\|$ or $\bowtie_S$ for some $S \neq \varnothing$), a node may possibly also include a recoloring function $R : [k] \to [k]$, which indicates a recoloring to be performed *after* the join operation, see Figure 1 for an example.

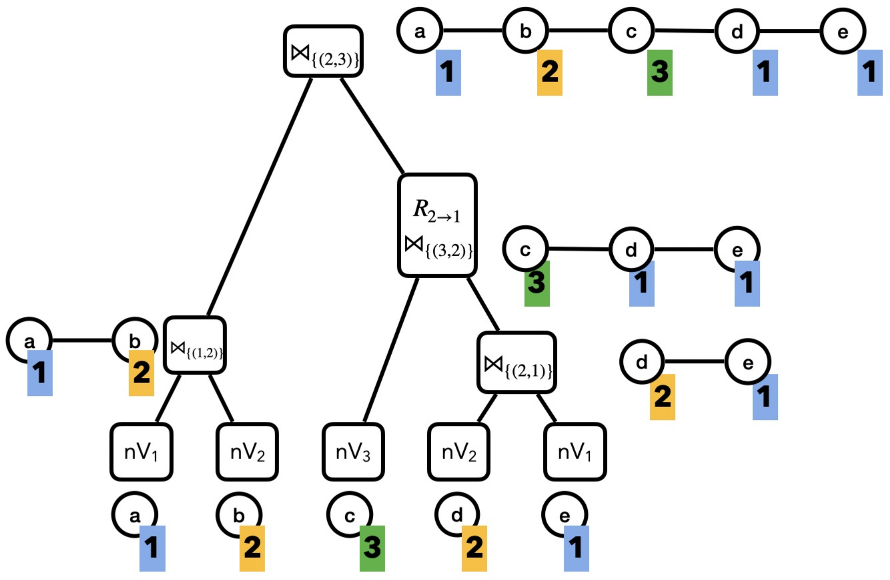


Figure 1 An NLC decomposition tree T. Next to each node x of the tree is displayed the colored graph $G[x]$ corresponding to subtree $T[x]$ of T rooted at x.

3.3 From Tree-Width to NLC-Width: The Main Messages

Let Π be an MSO_1 property, and let T be an NLC-decomposition tree of a graph G with $\mathsf{cw}(G) \leq k$. That is, we can choose the tree T as one using at most k colors. In the following, to avoid confusion, we call *vertices* the elements of the vertex set of G, and *nodes* the elements of the vertex-set of the decomposition tree T. The structure of our certificates differ from the one in [36], and now we decompose the certificate assigned to each node v into three parts: *main messages*, *auxiliary messages*, and *service messages*. This subsection focuses on the main messages.

Our main messages have, to some extent, a structure similar to the main messages used in [36] for the tree-width. In particular, vertex v receives a sequence $\mathsf{path}(v)$, listing all the nodes, i.e., the whole set of operations, in the path from the root of T to the leaf of T where v was created. For each node x in $\mathsf{path}(v)$, the main message also includes the vertex identifier of a leader for x, called *exit vertex of* x, and denoted by $\mathsf{exit}(x)$. The main message also includes a data structure $h(x)$ that encodes the truth value of the MSO_1 property on $G[x]$.

However, unlike the case of tree-width, where the nodes of the tree-decomposition are sets of vertices (i.e., bags) of bounded size, the contents of a non-leaf node in an NLC-decomposition tree T does not necessarily include information about the vertices created in $T[x]$. For that reason, our proof-labeling scheme includes additional information in the main message of v in order to verify the correctness of the given decomposition. It may actually be worth providing a concrete example to explain the need for additional information.

For a node x different from the root, let us denote by $p(x)$ the parent of x in T. The main message of v includes a sequence $\mathsf{links}(v)$ that specifies, for each node x in $\mathsf{path}(v)$ different from the root, whether x is the left or right child of $p(x)$. For instance, in the example of Figure 1, we have $\mathsf{links}(c) = (1, 0)$, indicating that, to reach the leaf creating vertex c from the root, one must follow the right child (1), and then the left child (0). S imilarly, $\mathsf{links}(d) = (1, 1, 0)$. The sequences links are also used to determine the longest common prefixes of the main messages, when the same operations are repeated between two children of a same node (consider for instance the case where the same operation is performed at all the nodes of the decomposition tree). Back to our example above, let us suppose that the sequences $\mathsf{links}(u)$ and $\mathsf{links}(v)$ specify that $x_{t_3+1}(u)$ is the left neighbor of x_{t_3}, and $x_{t_3+1}(v)$ is the right neighbor of x_{t_3}. Using this information, u and v can infer that it is an operation $\bowtie_S$, with $(c(u), c(v)) \in S$ that is specified in the description of x_{t_3}. With the given information, each vertex can thus check that all its incident edges are indeed created at some node of the decomposition tree T.

It remains to check that the decomposition does not define non-existent edges. To do so, the main message of every vertex v also includes, for each node x in $\mathsf{path}(v)$, and for each $i \in [k]$, the integers $\mathsf{color}_i(x)$ representing the number of vertices of $G[x]$ that are colored i in the root of $T[x]$. (Recall that the subgraph $G[x]$ is the subgraph of G induced by the vertices created in the subtree $T[x]$ of T). Returning to our example, vertex v checks that it has exactly $\mathsf{color}_{c(u)}(x_{t_3+1}(u))$ neighbors with the same longest common prefix as u colored $c(u)$ in the left children of x_{t_3}. Also, vertex v checks, for each $i \in [k]$, that the number of vertices colored i in node x_{t_3} corresponds to the sum of the number of vertices colored j in $x_{t_3+1}(u)$ and $x_{t_3+1}(v)$, for each color j that is recolored i by the recoloring operation defined in x_{t_3}. So, let us assume that the following consistency condition (analogous to the one for the certification of tree-decompositions) holds:

C1: For every pair of vertices $u, v \in V(G)$, and for every node x in both $\mathsf{path}(u)$ and $\mathsf{path}(v)$, u and v receive the same information about all nodes in the path from x to the root of T in their main messages, and

C2: If x is the root of T, then the data structure $h(x)$ describes an accepting instance (i.e., G satisfies Π).

Assuming that the consistency condition is satisfied, it is not difficult to show that the vertices can collectively check that the given certificates indeed represent an NLC-decomposition tree, and that G satisfies Π. The difficulty is however in checking that the consistency condition holds. This is the role of the auxiliary and service messages, described next.

3.4 Checking Consistency: Auxiliary, and Service Messages

We use auxiliary and service messages for allowing our proof-labeling scheme to check the first condition **C1** of the consistency condition defined at the end of the previous subsection.

Auxiliary messages can easily be defined for every node x of T satisfying that $G[x]$ is connected. In that case, the auxiliary messages of all the vertices v in $T[x]$ contain the certificates for certifying a spanning tree τ_x of $G[x]$ rooted at the exit vertex of x. Each vertex v can verify that the longest common prefix common to v and its parent in τ_x contains all the nodes from the root up to x, and that the information given in the main messages coincide for all such nodes. Observe that every vertex v may potentially contain one auxiliary message for each node in $\mathsf{path}(v)$.

The case where $G[x]$ is not connected is fairly more complicated, and we need to introduce another type of decomposition.

NLC+ decompositions trees. Observe that G itself is connected. Therefore, there must exist an ancestor z of x for which $G[z]$ is connected. We could provide the vertices in $G[z]$ with a spanning tree of $G[z]$ for checking the consistency in $T[x]$. However, the vertices in $G[z]$ do not necessarily contain x in the prefixes of their node sequences, so we would have to put a copy of the main message associated to x on every node participating in the spanning tree. Since an NLC-decomposition tree does not allow to provide a bound on the distance between z and x in the tree, we have no control on how many copies of main messages a vertex should handle.

Therefore, to cope with the case where $G[x]$ is disconnected, we define a specific type of NLC decompositions trees, called NLC+ decompositions trees. The NLC+ decomposition trees are similar to NLC-decomposition trees, up to two important differences.

- First, we allow the nodes corresponding to a $\|$ operation to have arbitrary large arity, and thus NLC+ decomposition trees are not binary trees, as opposed to NLC-decomposition trees.
- Second, if a node x induces a disconnected subgraph $G[x]$, then its parent node $p(x)$ must satisfy that $G[p(x)]$ is connected. Observe that $p(x)$ must then correspond to a $\bowtie$ operation, and thus $p(x)$ has only two children: x and another child, denoted by y.

Service trees. A *service tree* S_x for a node x such that $G[x]$ is disconnected is a Steiner tree in $G[p(x)]$ rooted at the exit vertex of x, and with all the vertices of $G[x]$ as terminals. Each vertex of S_x (i.e., all vertices in $G[x]$, plus some vertices in $G[y]$ is given a *service message*, which contains the certificate for the tree S_x, as well as a copy of the information about x given in the main messages of the vertices in $G[x]$. Each vertex in S_x can then check that it shares the same information about x than its parent. The properties of the NLC+ decomposition guarantee that a vertex v participates to at most two service trees, for each node x in the sequence $\mathsf{path}(v)$. Indeed, vertex v necessarily participates in S_x when x is of type $\|$, and may also participate in S_y whenever the sibling y of x is of type $\|$. There are significantly more subtle details concerning service trees, but they are described in the full version [35].

It remains to check the second condition **C2** of the consistency condition defined at the end of the previous subsection, which consists in verifying the correctness of $h(x)$, for every node x of T. This is explained next.

3.5 Dealing with MSO_1 Predicates

In their seminal work, Courcelle, Makowsky and Rotics [16] proved that every MSO_1 predicate Π on vertex-labeled graphs can be decided in linear time on graphs of bounded clique-width, and hence on graphs of bounded NLC-width, whenever a decomposition tree is part of the input. The running time of the algorithm is $O(n)$, i.e., linear in the number n of vertices of the input graph, with constants hidden in the big-O notation that depend on the clique-width bound, on the number of labels, and on the MSO_1 formula encoding the predicate Π. Note that this result does not hold for MSO_2 predicates, which is why our proof-labeling scheme applies to MSO_1 predicates only. We discuss the possible extension to MSO_2 properties in the conclusion (see Section 4).

For our purpose it is convenient to see the linear-time decision algorithm as a dynamic programming algorithm over the NLC-decomposition tree of the input graph. We formalize this dynamic programming approach following the vocabulary and notations used by Borie, Parker and Tovey [5]. Note that the latter provided an alternative proof of Courcelle's theorem, but for graphs of bounded tree-width, i.e., specific to a graph grammar defining tree-width. To design our proof-labeling scheme, we adapt their approach to a graph grammar defining NLC-width.

Homomorphism Classes. For a fixed property Π and a fixed parameter k, there is a finite set $\mathcal{C}$ of *homomorphism classes* (whose size depends only on Π and k) such that we can associate to each graph G of clique-width at most k its class $h(G) \in \mathcal{C}$ (for more details see the full version [35]). Whenever G is obtained from two graphs G_1 and G_2 by a $\bowtie_S$ operation potentially followed by a recoloring operation R, the class $h(G)$ only depends on $h(G_1)$, $h(G_2)$, $S \subseteq [k] \times [k]$, and $R : [k] \to [k]$. This property also holds whenever $\bowtie_S$ is replaced by $\|$. Moreover, we also extend the notion to arbitrary arity so that it holds for the NLC+ decomposition trees. Importantly, Courcelle's theorem [16] provides a "compiler" allowing to compute $h(G)$ whenever G is formed by a single vertex of color $j \in [k]$, and to compute $h(G)$ from $h(G_1)$, $h(G_2)$, S and R whenever $G = R(G_1 \bowtie_S G_2)$.

Checking Condition C2. In our proof-labeling scheme, for each node x of the NLC+ decomposition tree, we specify $h(x)$ as the class $h(G[x])$. Following the same principles as before, the consistency of these classes can be checked by simulating a bottom-up parsing of the decomposition tree, in a way very similar to what we described before for checking the consistency of $\mathsf{color}(x)$, but replacing the mere additions by updates of the homomorphism classes as described above.

This completes the rough description of our proof-labeling scheme.

3.6 Certificate Size

For each vertex v, the main, auxiliary, and service messages of v can be encoded using $O(\log n)$ bits for each node x in $\mathsf{path}(v)$, for the following reasons.

- The main message associated to a node x contains the following information. First, the list of operations described in the node, which can be encoded in $O(k^2)$ bits. Second, the corresponding index of links, which is just one bit representing whether x is the left or right children of its parent. Third, the homomorphism class $h(x)$ that can be encoded in $f(k)$ bits for some function f depending on the MSO_1 property under consideration – see the remark further in the text for a discussion about f. Finally, it includes the node identifier of the exit vertex of x, and the integers color_i for each $i \in [k]$. All these latter items can be encoded on $O(\log n)$ bits.
- The auxiliary message associated to node x (whenever $G[x]$ is connected) corresponds to the certification of a spanning tree of $G[x]$, which can be encoded in $O(\log n)$ bits (see [46]).
- For the service messages, note that vertex v participates in at most two service trees associated to x: the one of x (whenever $G[x]$ is disconnected), plus the one of the sibling y of x (when $G[y]$ is disconnected). Again, each of these trees can be certified using $O(\log n)$ bits.

Therefore, the total size of the certificates is $O(d \cdot \log n)$ bits, where d is the depth of the NLC+ decomposition tree T. Our final certificate size depends then on how much we can bound the depth d of T. Courcelle and Kanté [15] show that there always exists an NLC decomposition tree of logarithmic depth, but it comes with a price: the width of the small depth decomposition can be exponentially larger than the width of the original decomposition. Specifically, Courcelle and Kanté have shown that every n-node graph of NLC-width k admits an NLC-decomposition of width $k \cdot 2^{k+1}$ such that the corresponding decomposition tree T has depth $\mathcal{O}(\log n)$. Fortunately, our construction of NLC+ decomposition trees does not increase the depth of a given NLC-decomposition tree. In other words, we can use the result of Courcelle and Kanté to also show that NLC+ decomposition trees have logarithmic depth. Overall, we conclude that the certificate size is $O(\log^2 n)$ bits.

Remark. Our asymptotic bound on the size of the certificates hides a large dependency on the clique-width k of the input graph. For certifying the NLC+ decomposition only, the constant hidden in the big-O notation is single-exponential in k, given that the width of the NLC+ decomposition tree with logarithmic depth grows to $k \cdot 2^{k+1}$. However, for certifying an MSO_1 property, the dependency on k can be much larger, as it depends on the number of homomorphism classes. It is known that, for MSO_1 properties, the number of homomorphism classes is at most a tower of exponentials in k, where the height of the tower depends on the number of quantifiers in the MSO_1 formula. Moreover, this non-elementary dependency on k can not be improved significantly [40]. This exponential or even super-exponential dependency on the clique-width k is however inherent to the theory of algorithms for graphs of bounded clique-width. The same type of phenomenon occurs when dealing with graphs of bounded tree-width (see [40]), and the proof-labeling scheme in [36] is actually subject to the same type of dependencies in the bound k. On the other hand, the certificate size of our proof-labeling scheme grows only polylogarithmically with the size of the graphs.

4 Conclusion

In this paper, we have shown that, for every MSO_1 property Π on labeled graphs, there exists a PLS for Π with $O(\log^2 n)$-bit certificates for all n-node graphs of bounded clique-width. This extends previous results for smaller classes of graphs, namely graphs of bounded tree-depth [27], and graphs of bounded tree-width [36]. Our result also enables to establish a separation, in term of certificate size, between certifying C_4-free graphs and certifying P_4-free graphs.

A natural question is whether the certificate size resulting from our generic PLS construction is optimal. Note that one log-factor is related to the storage of IDs, and of similar types of information related to other nodes in the graph. It seems hard to avoid such a log-factor. The other log-factor is however directly related to the depth of the NLC-decomposition, and our PLS actually uses certificates of size $O(d \cdot \log n)$ bits for graphs supporting an NLC-decomposition of depth d. Nevertheless, the best generic upper bound for the depth d of an NLC-decomposition preserving bounded width is $O(\log n)$. This log-factor seems therefore hard to avoid too. Establishing the existence of a PLS for MSO_1 properties in graphs of bounded clique-width using $o(\log^2 n)$-bit certificates, or proving an $\Omega(\log^2 n)$ lower bound on the certificate size for such PLSs appears to be challenging.

Another interesting research direction is whether our result can be extended to MSO_2 properties. It is known that the meta-theorem from [16] does not extend to MSO_2. Nevertheless, this does not necessarily prevent the existence of compact PLSs for MSO_2 properties on graphs of bounded clique-width. For instance, Hamiltonicity is an MSO_2 property that can be easily certified in *all* graphs, using certificates on just $O(\log n)$ bits. Is there an MSO_2 property requiring certificates of $\Omega(n^\epsilon)$ bits, for some $\epsilon > 0$, on graphs of bounded clique-width? Finally, it might be interesting to study the existence of compact distributed interactive proofs [44] for certifying MSO_1 or even MSO_2 properties on graphs of bounded clique-width. Note that the generic compiler from [47] efficiently applies to sparse graphs only whereas the family of graphs with bounded clique-width includes dense graphs.

References

1 Alkida Balliu, Gianlorenzo D'Angelo, Pierre Fraigniaud, and Dennis Olivetti. What can be verified locally? *J. Comput. Syst. Sci.*, 97:106–120, 2018.

2 Alkida Balliu and Pierre Fraigniaud. Certification of compact low-stretch routing schemes. *Comput. J.*, 62(5):730–746, 2019.

3 Aviv Bick, Gillat Kol, and Rotem Oshman. Distributed zero-knowledge proofs over networks. In *33rd ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2426–2458, 2022.

4 Hans L Bodlaender. Nc-algorithms for graphs with small treewidth. In *Graph-Theoretic Concepts in Computer Science: International Workshop WG'88 Amsterdam, The Netherlands, June 15–17, 1988 Proceedings 14*, pages 1–10. Springer, 1989.

5 Richard B. Borie, R. Gary Parker, and Craig A. Tovey. Automatic generation of linear-time algorithms from predicate calculus descriptions of problems on recursively constructed graph families. *Algorithmica*, 7(5&6):555–581, 1992. `doi:10.1007/BF01758777`.

6 Nicolas Bousquet, Laurent Feuilloley, and Théo Pierron. Local certification of graph decompositions and applications to minor-free classes. In *25th International Conference on Principles of Distributed Systems (OPODIS)*, volume 217 of *LIPIcs*, pages 22:1–22:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

7 Zvika Brakerski and Boaz Patt-Shamir. Distributed discovery of large near-cliques. *Distributed Comput.*, 24(2):79–89, 2011.

8 Andreas Brandstädt, Van Bang Le, and Jeremy P. Spinrad. *Graph Classes: A Survey*. Society for Industrial and Applied Mathematics, 1999.

9 Keren Censor-Hillel, Eldar Fischer, Gregory Schwartzman, and Yadu Vasudev. Fast distributed algorithms for testing graph properties. *Distributed Comput.*, 32(1):41–57, 2019.

10 Keren Censor-Hillel, Orr Fischer, Tzlil Gonen, François Le Gall, Dean Leitersdorf, and Rotem Oshman. Fast Distributed Algorithms for Girth, Cycles and Small Subgraphs. In *34th International Symposium on Distributed Computing (DISC)*, volume 179 of *LIPIcs*, pages 33:1–33:17. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020.

11 Keren Censor-Hillel, Ami Paz, and Mor Perry. Approximate proof-labeling schemes. In *24th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, LNCS 10641, pages 71–89. Springer, 2017.

12 Derek G. Corneil and Udi Rotics. On the relationship between clique-width and treewidth. *SIAM Journal on Computing*, 34(4):825–847, 2005.

13 Bruno Courcelle. The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Inf. Comput.*, 85(1):12–75, 1990. `doi:10.1016/0890-5401(90)90043-H`.

14 Bruno Courcelle and Joost Engelfriet. *Graph structure and monadic second-order logic. A language-theoretic approach.* Encyclopedia of Mathematics and its applications, Vol. 138. Cambridge University Press, June 2012. Collection Encyclopedia of Mathematics and Applications, Vol. 138. URL: `https://hal.science/hal-00646514`.

15 Bruno Courcelle and Mamadou Moustapha Kanté. Graph operations characterizing rank-width and balanced graph expressions. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 66–75. Springer, 2007.

16 Bruno Courcelle, Johann A. Makowsky, and Udi Rotics. Linear time solvable optimization problems on graphs of bounded clique-width. *Theory Comput. Syst.*, 33(2):125–150, 2000. doi:10.1007/s002249910009.

17 Bruno Courcelle and Stephan Olariu. Upper bounds to the clique width of graphs. *Discret. Appl. Math.*, 101(1-3):77–114, 2000. doi:10.1016/S0166-218X(99)00184-5.

18 Pierluigi Crescenzi, Pierre Fraigniaud, and Ami Paz. Trade-offs in distributed interactive proofs. In *33rd International Symposium on Distributed Computing (DISC)*, volume 146 of *LIPIcs*, pages 13:1–13:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

19 Konrad K. Dabrowski and Daniël Paulusma. Clique-width of graph classes defined by two forbidden induced subgraphs. *The Computer Journal*, 59(5):650–666, 2016.

20 Andrew Drucker, Fabian Kuhn, and Rotem Oshman. On the power of the congested clique model. In *33rd ACM Symposium on Principles of Distributed Computing (PODC)*, pages 367–376, 2014.

21 Andrew Drucker, Fabian Kuhn, and Rotem Oshman. On the power of the congested clique model. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, pages 367–376, 2014.

22 Talya Eden, Nimrod Fiat, Orr Fischer, Fabian Kuhn, and Rotem Oshman. Sublinear-time distributed algorithms for detecting small cliques and even cycles. *Distributed Computing*, 35(3):207–234, 2022.

23 Yuval Emek and Yuval Gil. Twenty-two new approximate proof labeling schemes. In *34th International Symposium on Distributed Computing (DISC)*, volume 179 of *LIPIcs*, pages 20:1–20:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

24 Yuval Emek, Yuval Gil, and Shay Kutten. Locally restricted proof labeling schemes. In *36th International Symposium on Distributed Computing (DISC)*, volume 246 of *LIPIcs*, pages 20:1–20:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

25 Louis Esperet and Benjamin Lévêque. Local certification of graphs on surfaces. *Theor. Comput. Sci.*, 909:68–75, 2022.

26 Guy Even, Orr Fischer, Pierre Fraigniaud, Tzlil Gonen, Reut Levi, Moti Medina, Pedro Montealegre, Dennis Olivetti, Rotem Oshman, Ivan Rapaport, and Ioan Todinca. Three notes on distributed property testing. In *31st International Symposium on Distributed Computing (DISC)*, volume 91 of *LIPIcs*, pages 15:1–15:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

27 Laurent Feuilloley, Nicolas Bousquet, and Théo Pierron. What can be certified compactly? compact local certification of MSO properties in tree-like graphs. In *41st ACM Symposium on Principles of Distributed Computing (PODC)*, pages 131–140, 2022.

28 Laurent Feuilloley and Pierre Fraigniaud. Error-sensitive proof-labeling schemes. *J. Parallel Distributed Comput.*, 166:149–165, 2022.

29 Laurent Feuilloley, Pierre Fraigniaud, and Juho Hirvonen. A hierarchy of local decision. *Theor. Comput. Sci.*, 856:51–67, 2021.

30 Laurent Feuilloley, Pierre Fraigniaud, Juho Hirvonen, Ami Paz, and Mor Perry. Redundancy in distributed proofs. *Distributed Comput.*, 34(2):113–132, 2021.

31 Laurent Feuilloley, Pierre Fraigniaud, Pedro Montealegre, Ivan Rapaport, Éric Rémila, and Ioan Todinca. Compact distributed certification of planar graphs. *Algorithmica*, 83(7):2215–2244, 2021.

32 Laurent Feuilloley and Juho Hirvonen. Local verification of global proofs. In *32nd International Symposium on Distributed Computing*, volume 121 of *LIPIcs*, pages 25:1–25:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

33 Pierre Fraigniaud, François Le Gall, Harumichi Nishimura, and Ami Paz. Distributed quantum proofs for replicated data. In *12th Innovations in Theoretical Computer Science Conference (ITCS)*, volume 185 of *LIPIcs*, pages 28:1–28:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

34 Pierre Fraigniaud, Amos Korman, and David Peleg. Towards a complexity theory for local distributed computing. *J. ACM*, 60(5):35:1–35:26, 2013.

35 Pierre Fraigniaud, Frédéric Mazoit, Pedro Montealegre, Ivan Rapaport, and Ioan Todinca. Distributed certification for classes of dense graphs, 2023. `arXiv:2307.14292`.

36 Pierre Fraigniaud, Pedro Montealegre, Ivan Rapaport, and Ioan Todinca. A meta-theorem for distributed certification. In *29th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, volume 13298 of *LNCS*, pages 116–134. Springer, 2022.

37 Pierre Fraigniaud and Dennis Olivetti. Distributed detection of cycles. *ACM Trans. Parallel Comput.*, 6(3):12:1–12:20, 2019.

38 Pierre Fraigniaud, Boaz Patt-Shamir, and Mor Perry. Randomized proof-labeling schemes. *Distributed Comput.*, 32(3):217–234, 2019.

39 Pierre Fraigniaud, Ivan Rapaport, Ville Salo, and Ioan Todinca. Distributed testing of excluded subgraphs. In *30th International Symposium on Distributed Computing (DISC)*, volume 9888 of *LNCS*, pages 342–356. Springer, 2016.

40 Markus Frick and Martin Grohe. The complexity of first-order and monadic second-order logic revisited. *Ann. Pure Appl. Log.*, 130(1-3):3–31, 2004. `doi:10.1016/j.apal.2004.01.007`.

41 François Le Gall, Masayuki Miyamoto, and Harumichi Nishimura. Brief announcement: Distributed quantum interactive proofs. In *36th International Symposium on Distributed Computing (DISC)*, volume 246 of *LIPIcs*, pages 48:1–48:3. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

42 Mika Göös and Jukka Suomela. Locally checkable proofs in distributed computing. *Theory Comput.*, 12(1):1–33, 2016.

43 Ö. Johansson. Clique-decomposition, nlc-decomposition, and modular decomposition - relationships and results for random graphs. *Congressus Numerantium*, 132:39–60, 1998.

44 Gillat Kol, Rotem Oshman, and Raghuvansh R. Saxena. Interactive distributed proofs. In *37th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 255–264. ACM, 2018.

45 Amos Korman and Shay Kutten. Distributed verification of minimum spanning trees. *Distributed Comput.*, 20(4):253–266, 2007.

46 Amos Korman, Shay Kutten, and David Peleg. Proof labeling schemes. *Distributed Comput.*, 22(4):215–233, 2010.

47 Moni Naor, Merav Parter, and Eylon Yogev. The power of distributed verifiers in interactive proofs. In *31st ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1096–115. SIAM, 2020.

48 Jaroslav Nesetril and Patrice Ossona de Mendez. *Sparsity - Graphs, Structures, and Algorithms*, volume 28 of *Algorithms and combinatorics.* Springer, 2012. `doi:10.1007/978-3-642-27875-4`.

The Synchronization Power (Consensus Number) of Access-Control Objects: the Case of AllowList and DenyList

Davide Frey
Inria, IRISA, CNRS, Université de Rennes, France

Mathieu Gestin
Inria, IRISA, CNRS, Université de Rennes, France

Michel Raynal
IRISA, Inria, CNRS, Université de Rennes, France

Abstract

This article studies the synchronization power of AllowList and DenyList objects under the lens provided by Herlihy's consensus hierarchy. It specifies AllowList and DenyList as distributed objects and shows that, while they can both be seen as specializations of a more general object type, they inherently have different synchronization power. While the AllowList object does not require synchronization between participating processes, a DenyList object requires processes to reach consensus on a specific set of processes. These results are then applied to a more global analysis of anonymity-preserving systems that use AllowList and DenyList objects. First, a blind-signature-based e-voting is presented. Second, DenyList and AllowList objects are used to determine the consensus number of a specific decentralized key management system. Third, an anonymous money transfer algorithm using the association of AllowList and DenyList objects is presented. Finally, this analysis is used to study the properties of these application, and to highlight efficiency gains that they can achieve in message passing environment.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Security and privacy → Access control; Security and privacy → Pseudonymity, anonymity and untraceability

Keywords and phrases Access control, AllowList/DenyList, Blockchain, Consensus number, Distributed objects, Modularity, Privacy, Synchronization power

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.21

Related Version *Full Version*: `https://doi.org/10.48550/arXiv.2302.06344v2` [19]

Funding This work was partially funded by the PriCLeSS project and by the SOTERIA H2020 project. PriCLeSS was granted by the Labex CominLabs excellence laboratory of the French ANR (ANR-10-LABX-07-01). SOTERIA received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No101018342. This content reflects only the author's view. The European Agency is not responsible for any use that may be made of the information it contains.

Acknowledgements We wish to thank the anonymous reviewers for their insightful comments and remarks that led to significant improvements to our paper.

1 Introduction

The advent of blockchain technologies increased the interest of the public and industry in distributed applications, giving birth to projects that have applied blockchains in a plethora of use cases. These include e-vote systems [16], naming services [1, 27], Identity Management Systems [18, 31], supply-chain management [30], or Vehicular Ad hoc Network [21]. However, this use of the blockchain as a swiss-army knife that can solve numerous distributed problems highlights a lack of understanding of the actual requirements of those problems. Because of these poor specifications, implementations of these applications are often sub-optimal.

This paper thoroughly studies a class of problems widely used in distributed applications and provides a guideline to implement them with reasonable but sufficient tools.

Differently from the previous approaches, it aims to understand the amount of synchronization required between processes of a system to implement *specific* distributed objects. To achieve this goal it studies such objects under the lens of Herlihy's consensus number [24]. This parameter is inherently associated to shared memory distributed objects, and has no direct correspondence in the message passing environment. However, in some specific cases, this information is enough to provide a better understanding of the objects analyzed, and thus, to gain efficiency in the message passing implementations. For example, recent papers [22, 5] have shown that cryptocurrencies can be implemented without consensus and therefore without a blockchain. In particular, Guerraoui et al. [22] show that k-asset transfer has a consensus number k where k is the number of processes that can withdraw currency from the same account [23]. Similarly, Alpos et al. [3] have studied the synchronization properties of ERC20 token smart contracts and shown that their consensus number varies over time as a result of changes in the set of processes that are approved to send tokens from the same account. These two results consider two forms of asset transfer: the classical one and the one implemented by the ERC20 token, which allows processes to dynamically authorize other processes. The consensus number of those objects depends on specific and well identified processes. From this study, it is possible to conclude that the consensus algorithms only need to be performed between those processes. Therefore, in these specific cases, the knowledge of the consensus number of an object can be directly used to implement more efficient message passing applications. Furthermore, even if this study uses a shared memory model, with crash prone processes, its results can be used to implement more efficient Byzantine resilient algorithm, in a message passing environment. This paper proposes to extend this knowledge to a broader class of applications.

Indeed, the transfer of assets, be them cryptocurrencies or non-fungible tokens, does not constitute the only application in the Blockchain ecosystem. In particular, as previously indicated, a number of applications like e-voting [16], naming [1, 27], or Identity Management [18, 31] use Blockchain as a tool to implement some form of access control. This is often achieved by implementing two general-purpose objects: AllowLists and DenyLists. An AllowList provides an opt-in mechanism. A set of managers can maintain a list of authorized parties, namely the AllowList. To access a resource, a party (user) must prove the presence of an element associated with its identity in the AllowList. A DenyList provides instead an opt-out mechanism. In this case, the managers maintain a list of revoked elements, the DenyList. To access a resource, a party (user) must prove that no corresponding element has been added to the DenyList. In other words, AllowList and DenyList support, respectively, set-membership and set-non-membership proofs on a list of elements.

The proofs carried out by AllowList and DenyList objects often need to offer privacy guarantees. For example, the Sovrin privacy preserving Decentralized Identity-Management System (DIMS) [18] associates an AllowList[1] with each verifiable credential that contains the identifiers of the devices that can use this verifiable credential. When a device uses a credential with a verifier, it needs to prove that the identifier associated with it belongs to the AllowList. This proof must be done in zero knowledge, otherwise the verifier would learn

[1] In reality this is a variant that mixes AllowList and DenyList which we discuss in Appendix A.

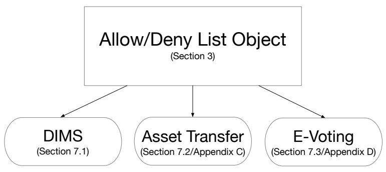
the identity of the device, which in turn could serve as a pseudo-identifier for the user. For this reason, AllowList and DenyList objects support respectively a zero-knowledge proof of set membership or a zero-knowledge proof of set non-membership.

Albeit similar, the AllowList and DenyList objects differ significantly in the way they handle the proving mechanism. In the case of an AllowList, no security risk appears if access to a resource is prohibited to a process, even if a manager did grant this right. As a result, a transient period in which a user is first allowed, then denied, and then allowed again to access a resource poses no problem. On the contrary, with a DenyList, being allowed access to a resource after being denied it poses serious security problems. Hence, the DenyList object is defined with an additional anti-flickering property prohibiting such transient periods. This property is the main difference between an AllowList and a DenyList object and is the reason for their distinct consensus numbers.

Existing systems [16, 1, 27, 18, 31] that employ AllowList and DenyList objects implement them on top of a heavy blockchain infrastructure, thereby requiring network-level consensus to modify their content. As already said, this paper studies this difference under the lens of the consensus number [23]. It shows that (i) the consensus number of an AllowList object is 1, which means that an AllowList can be implemented without consensus; and that (ii) the consensus number of a DenyList is instead equal to the number of processes that can conduct prove operations on the DenyList, and that only these processes need to synchronize. Both data structures can therefore be implemented without relying on the network-level consensus provided by a blockchain, which opens the door to more efficient implementations of applications based on these data structures.

To summarize, this paper presents the following three contributions. We note $CN(X)$ the consensus number of the object X.

1. It formally defines and studies AllowList and DenyList as distributed objects (Section 3).
2. It analyses the consensus number of these objects: it shows that the AllowList does not require synchronization between processes, i.e. $CN(\text{AllowList }) = 1$ (Section 5), while the DenyList requires the synchronization of all the k verifiers of its set-non-membership proofs, i.e $CN(\text{DenyList }) = k$ (Section 6).
3. It uses these theoretical results to give intuitions on their optimal implementations. Namely the implementation of a DIMS, as well as of an e-vote system and an Anonymous Asset-Transfer (AAT) algorithm (Appendix B and in the full version of this paper [19]). More precisely, the consensus number of an AAT algorithm depends on the required anonymity level, i.e. $CN(\text{AAT}) = CN(\text{DenyList}) = k = CN(k\text{-shared Asset Transfer object})$. The consensus number of an e-vote system depends on the number k of vote-casting servers, i.e $CN(\text{e-vote}) = k$. Finally, the consensus number of a the revocation mechanism in a DIMS is 2 in most cases.



To the best of our knowledge, this paper is the first to study the AllowList and DenyList from a distributed algorithms point of view. So we believe our results can provide a powerful tool to identify the consensus number of recent distributed objects that make use of them and to provide more efficient implementations of such objects.

2 Preliminaries

2.1 Computation Model

Model

Let Π be a set of N asynchronous sequential crash-prone processes $p_1, \cdots, p_N$. Sequential means that each process invokes one operation of its own algorithm at a time. We assume the local processing time to be instantaneous, but the system is asynchronous. This means that non-local operations can take a finite but arbitrarily long time and that the relative speeds between the clocks of the different processes are unknown. Finally, processes are crash-prone: any number of processes can prematurely and definitely halt their executions. A process that crashes is called *faulty*. Otherwise, it is called *correct*. The system is eponymous: a unique positive integer identifies each process, and this identifier is known to all other processes.

Communication

Processes communicate via shared objects of type T. Each operation on a shared object is associated with two *events*: an *invocation* and a *response*. An object type T is defined by a tuple (Q, Q_0, O, R, Δ), where Q is a set of states, $Q_0 \subseteq Q$ is the set of initial states, O is the set of operations a process can use to access this object, R is the set of responses to these operations, and $\Delta \subseteq \Pi \times Q \times O \times R \times Q$ is the transition function defining how a process can access and modify an object.

Histories and Linearizability

A *history* [24] is a sequence of invocations and responses in the execution of an algorithm. An invocation with no matching response in a history, H, is called a *pending* invocation. A *sequential history* is one where the first event is an invocation, and each invocation – except possibly the last one – is immediately followed by the associated response. A sub-history is a sub-sequence of events in a history. A process sub-history $H|p_i$ of a history H is a sub-sequence of all the events in H whose associated process is p_i. Given an object x, we can similarly define the object sub-history $H|x$. Two histories H and H' are equivalent if $H|p_i = H'|p_i$, $\forall i \in \{1, \cdots, N\}$.

In this paper, we define the specification of a shared object, x, as the set of all the allowed sub-histories, $H|x$. We talk about a sequential specification if all the histories in this set are sequential. A *legal history* is a history H in which, for all objects x_i of this history, $H|x_i$ belongs to the specification of x_i. The completion $\bar{H}$ of a history H is obtained by extending all the pending invocations in H with the associated matching responses. A history H induces an irreflexive partial order $<_H$ on operations, i.e. $op_0 <_H op_1$ if the response to the operation op_0 precedes the invocation of operation op_1. A history is sequential if $<_H$ is a total order. The algorithm executed by a correct process is *wait-free* if it always terminates after a finite number of steps. A history H is linearizable if a completion $\bar{H}$ of H is equivalent to some legal sequential history S and $<_H \subseteq <_S$.

Consensus number

The consensus number of an object of type T (noted $\text{cons}(T)$) is the largest n such that it is possible to wait-free implement a consensus object from atomic read/write registers and objects of type T in a system of n processes. If an object of type T makes it possible to wait-free implement a consensus object in a system of any number of processes, we say the consensus number of this object is ∞. Herlihy [23] proved the following well-known theorem.

▶ **Theorem 1.** *Let X and Y be two atomic objects type such that cons(X) $= m$ and cons(Y) $= n$, and $m < n$. There is no wait-free implementation of an object of type Y from objects of type X and read/write registers in a system of more than m processes.*

We will determine the consensus number of the DenyList and the AllowList objects using Atomic Snapshot objects and consensus objects in a set of k processes. A Single Writer Multi Reader (SWMR) [2] Atomic Snapshot object is an array of fixed size, which supports two operations: Snapshot and Update. The Snapshot() operation allows a process p_i to read the whole array in one atomic operation. The Update(v, i) operation allows a process p_i to write the value v in the i-th position of the array. Afek et al. showed that a SWMR Snapshot object can be wait-free implemented from read/write registers [2], i.e., this object type has consensus number 1. This paper assumes that all Atomic Snapshot objects used are SWMR. A consensus object provides processes with a single one-shot operation *propose()*. When a process p_i invokes *propose(v)* it proposes v. This invocation returns a *decided* value such that the following three properties are satisfied.

- *Validity*: If a correct process decides value v, then v was proposed by some process;
- *Agreement*: No two correct processes decide differently; and
- *Termination*: Every correct process eventually decides.

A k-consensus object is a consensus object accessed by at most k processes.

2.2 Number theory preliminaries

Cryptographic Commitments

A *cryptographic commitment* is a cryptographic scheme that allows a Prover to commit to a value v while hiding it. The commitment scheme is a two phases protocol. First, the prover computes a binding value known as commitment, C, using a function *Commit*. *Commit* takes as inputs the value v and a random number r. The prover sends this hiding and binding value C to a verifier. In the second phase, the prover reveals the committed value v and the randomness r to the verifier. The verifier can then verify that the commitment C previously received refers to the transmitted values v and r. This commitment protocol is the heart of Zero Knowledge Proof (ZKP) protocols.

Zero Knowledge Proof of set operations

A Zero Knowledge Proof (ZKP) system is a cryptographic algorithm that allows a prover to prove some Boolean statement about a value x to a verifier without leaking any information about x. A ZKP system is initialized for a specific language $\mathcal{L}$ of the complexity class $\mathcal{NP}$. The proving mechanism takes as input $\mathcal{L}$ and outputs a proof π. Knowing $\mathcal{L}$ and π, any verifier can verify that the prover knows a value $x \in \mathcal{L}$[2]. However, the verifier cannot learn the value x used to produce the proof. In the following, it is assumed there exists efficient non interactive ZKP systems of set-(non)-membership (e.g., constructions from [8]).

[2] The notation $x \in \mathcal{L}$ denotes the fact that x is a solution to the instance of the problem expressed by the language $\mathcal{L}$

3 The AllowList and DenyList objects: Definition

Distributed AllowList and DenyList object types are the type of objects that allow a set of managers to control access to a resource. The term ”resource” is used here to describe the goal a user wants to achieve and which is protected by an access control policy. A user is granted access to the resource if it succeeds in proving that it is authorized to access it. First, we describe the AllowList object type. Then we consider the DenyList object type.

The AllowList object type is one of the two most common access control mechanisms. To access a resource, a process $p \in \Pi_V$ needs to prove it knows some element v previously authorized by a process $p_M \in \Pi_M$, where $\Pi_M \subseteq \Pi$ is the set of managers, and $\Pi_V \subseteq \Pi$ is the set of processes authorized to conduct proofs. We call verifiers the processes in Π_V. The sets Π_V and Π_M are predefined and static. They are parameters of the object. Depending on the usage, these subset can either be small, or they can contain all the processes in Π.

A process $p \in \Pi_V$ proves that v was previously authorized by invoking a PROVE(v) operation. This operation is said to be valid if some manager in Π_M previously invoked an APPEND(v) operation. Intuitively, we can see the invocation of APPEND(v) as the action of authorizing some process to access the resource. On the other hand, the PROVE(v) operation, invoked by a prover process, $p \in \Pi_V$, proves to the other processes in Π_V that they are authorized. However, this proof is not enough in itself. The verifiers of a proof must be able to verify that a valid PROVE has been invoked. To this end, the AllowList object type is also equipped with a READ() operation. This operation can be invoked by any process in Π and returns a random permutation of all the valid PROVE invoked, along with the identity of the processes that invoked them. All processes in Π can invoke the READ operation.[3]

An optional anonymity property can be added to the AllowList object to enable privacy-preserving implementations. This property ensures that other processes cannot learn the value v proven by a PROVE(v) operation.

The AllowList object type is formally defined as a sequential object, where each invocation is immediately followed by a response. Hence, the sequence of operations defines a total order, and each operation can be identified by its place in the sequence.

▶ **Definition 2.** The *AllowList* object type supports three operations: APPEND, PROVE, and READ. These operations appear as if executed in a sequence Seq such that:

- *Termination.* A PROVE, an APPEND, or a READ operation invoked by a correct process always returns.
- APPEND *Validity.* The invocation of APPEND(x) by a process p is valid **if** $p \in \Pi_M \subseteq \Pi$ **and** $x \in \mathcal{S}$, where $\mathcal{S}$ is a predefined set. Otherwise, the operation is invalid.
- PROVE *Validity.* **If** the invocation of *op* =PROVE(x) by a process p is valid, **then** $p \in \Pi_V \subseteq \Pi$ **and** a valid APPEND(x) appears before *op* in Seq. Otherwise, the invocation is invalid.
- *Progress.* **If** a valid APPEND(x) is invoked, **then** there exists a point in Seq such that any PROVE(x) invoked after this point by any process $p \in \Pi_V$ will be valid.
- READ *Validity.* The invocation of *op* =READ() by a process $p \in \Pi_V$ returns the list of valid invocations of PROVE that appears before *op* in Seq along with the names of the processes that invoked each operation.

[3] Usually, AllowList objects are implemented in a message-passing setting. In these cases, the READ operation is implicit. Each process knows a local state of the distributed object, and can inspect it any time. In the shared-memory setting, we need to make this READ operation explicit.

- *Optional - Anonymity.* Let us assume the process p invokes a PROVE(v) operation. If the process p' invokes a READ() operation, then p' cannot learn the value v unless p leaks additional information.[4]

The AllowList object is defined in an append-only manner. This definition makes it possible to use it to build all use cases explored in this paper. However, some use cases could need an DenyList with an additional REMOVE operation. This variation is studied in Appendix A.

The DenyList object type can be informally presented as an access policy where, contrary to the AllowList object type, all users are authorized to access the resource in the first place. The managers are here to revoke this authorization. A manager revokes a user by invoking the APPEND(v) operation. A user uses the PROVE(v) operation to prove that it was not revoked. A PROVE(v) invocation is invalid only if a manager previously revoked the value v.

All the processes in Π can verify the validity of a PROVE operation by invoking a READ() operation. This operation is similar to the AllowList's READ operation. It returns the list of valid PROVE invocations along with the name of the processes that invoked it.

There is one significant difference between the DenyList and the AllowList object types. With an AllowList, if a user cannot access a resource immediately after its authorization, no malicious behavior can harm the system – the system's state is equivalent to its previous state. However, with a DenyList, a revocation not taken into account can let a malicious user access the resource and harm the system. In other words, access to the resource in the DenyList case must take into account the "most up to date" available revocation list.

To this end, the DenyList object type is defined with an additional property. The anti-flickering property ensures that if an APPEND operation is taken into account by one PROVE operation, it will be taken into account by every subsequent PROVE operation. Along with the progress property, the anti-flickering property ensures that the revocation mechanism is as immediate as possible. The DenyList object is formally defined as a sequential object, where each invocation is immediately followed by a response. Hence, the sequence of operations define a total order, and each operation can be identified by its place in the sequence.

▶ **Definition 3.** The *DenyList* object type supports three operations: APPEND, PROVE, and READ. These operations appear as if executed in a sequence Seq such that:

- *Termination.* A PROVE, an APPEND, or a READ operation invoked by a correct process always returns.
- APPEND *Validity.* The invocation of APPEND(x) by a process p is valid **if** $p \in \Pi_M \subseteq \Pi$ **and** $x \in \mathcal{S}$, where $\mathcal{S}$ is a predefined set. Otherwise, the operation is invalid.
- PROVE *Validity.* **If** the invocation of a op =PROVE(x) by a correct process p is not valid, **then** $p \notin \Pi_V \subseteq \Pi$ **or** a valid APPEND(x) appears before op_P in Seq. Otherwise, the operation is valid.
- PROVE *Anti-Flickering.* **If** the invocation of a operation op =PROVE(x) by a correct process $p \in \Pi_V$ is invalid, **then** any PROVE(x) that appears after op in Seq is invalid.[5]

[4] The Anonymity property only protects the value v. The system considered is eponymous. Hence, the identity of the processes is already known. However, the anonymity of v makes it possible to hide other information. For example, the identity of a client that issues a request to a process of the system. These example are discussed in Section 7.

[5] The only difference between the AllowList and the DenyList object types is this anti-flickering property. As it is shown in Section 5 and in Section 6, the AllowList object has consensus number 1, and the DenyList object has consensus number $k = |\Pi_V|$. Hence, this difference in term of consensus number is due solely to the anti-flickering property. It is an open question whether a variation of this property could transform any consensus number 1 object into a consensus number k object.

Table 1 Transition function Δ for the PROOF-LIST object.

Process	Operation	Initial state	Response	Final state	Conditions
$p_i \in \Pi_M$	APPEND(y)	$(\textit{listed-values} = \{x \in \mathcal{S}\},$ $\textit{proofs} = (\{(p_j \in \Pi, \widehat{\mathcal{S}} \subseteq \mathcal{S}, \mathsf{P} \in \mathcal{P}_{\mathcal{L}_{\widehat{\mathcal{S}}}})\}))$	True	$(\textit{listed-values} \cup \{y\},$ $\textit{proofs})$	$y \in \mathcal{S}$
p_i	APPEND(y)	$(\textit{listed-values} = \{x \in \mathcal{S}\},$ $\textit{proofs} = (\{(p_j \in \Pi, \widehat{\mathcal{S}} \subseteq \mathcal{S}, \mathsf{P} \in \mathcal{P}_{\mathcal{L}_{\widehat{\mathcal{S}}}})\}))$	False	$(\textit{listed-values}, \textit{proofs})$	$p_i \notin \Pi_M \vee y \notin \mathcal{S}$
$p_i \in \Pi_V$	PROVE(y)	$(\textit{listed-values} = \{x \in \mathcal{S}\},$ $\textit{proofs} = (\{(p_j \in \Pi, \widehat{\mathcal{S}} \subseteq \mathcal{S}, \mathsf{P} \in \mathcal{P}_{\mathcal{L}_{\widehat{\mathcal{S}}}})\}))$	$(\mathcal{A}, \mathsf{P})$	$(\textit{listed-values},$ $\textit{proofs} \cup \{(p_i, \mathcal{A}, \mathsf{P})\})$	$\forall y \in \mathcal{L}_{\mathcal{A}} \wedge \mathcal{A} \subseteq \textit{listed-values}$ $\wedge \forall \mathsf{P} \in \mathcal{P}_{\mathcal{L}_{\mathcal{A}}} \wedge \mathsf{C}(y, \widehat{\mathcal{S}}) = 1$
p_i	PROVE(y)	$(\textit{listed-values} = \{x \in \mathcal{S}\},$ $\textit{proofs} = (\{(p_j \in \Pi, \widehat{\mathcal{S}} \subseteq \mathcal{S}, \mathsf{P} \in \mathcal{P}_{\mathcal{L}_{\widehat{\mathcal{S}}}})\}))$	False	$(\textit{listed-values}, \textit{proofs})$	$\forall y \notin \mathcal{L}_{\mathcal{A}} \vee \mathcal{A} \not\subseteq \textit{listed-values}$ $\vee \forall \mathsf{P} \notin \mathcal{P}_{\mathcal{L}_{\mathcal{A}}} \vee \forall p_i \notin \Pi_V$ $\vee \mathsf{C}(y, \widehat{\mathcal{S}}) = 0$
$p_i \in \Pi$	READ()	$(\textit{listed-values} = \{x \in \mathcal{S}\},$ $\textit{proofs} = (\{(p_j \in \Pi, \widehat{\mathcal{S}} \subseteq \mathcal{S}, \mathsf{P} \in \mathcal{P}_{\mathcal{L}_{\widehat{\mathcal{S}}}})\}))$	*proofs*	$(\textit{listed-values}, \textit{proofs})$	

- READ *Validity.* The invocation of *op* =READ() by a process $p \in \Pi_V$ returns the list of valid invocations of PROVE that appears before *op* in Seq along with the names of the processes that invoked each operation.
- *Optional - Anonymity.* Let us assume the process p invokes a PROVE(v) operation. If the process p' invokes a READ() operation, then p' cannot learn the value v unless p leaks additional information.

4 PROOF-LIST object specification

Section 5 and Section 6 propose an analysis of the synchronization power of the AllowList and the DenyList object types using the notion of consensus number. Both objects share many similarities. Indeed, the only difference is the type of proof performed by the user and the non-flickering properties. Therefore, this section defines the formal specification of the PROOF-LIST object type, a new generic object that can be instantiated to describe the AllowList or the DenyList object type.

The PROOF-LIST object type is a distributed object type whose state is a pair of arrays (*listed-values*, *proofs*). The first array, *listed-values*, represents the list of authorized/revoked elements. It is an array of objects in a set $\mathcal{S}$, where $\mathcal{S}$ is the universe of potential elements. The second array, *proofs*, is a list of assertions about the *listed-values* array. Given a set of managers $\Pi_M \subseteq \Pi$ and a set of verifiers $\Pi_V \subseteq \Pi$, the PROOF-LIST object supports three operations. First, the APPEND(v) operation appends a value $v \in \mathcal{S}$ to the *listed-values* array. Any process in the manager's set can invoke this operation. Second, the PROVE(v) operation appends a valid proof about the element $v \in \mathcal{S}$ relative to the *listed-values* array to the *proofs* array. This operation can be invoked by any process $p \in \Pi_V$. Third, the READ() operation returns the *proofs* array.

The sets Π_V and Π_M are static, predefined subsets of Π. There is no restriction on their compositions. The choice of these sets only depends on the usage of the AllowList or the DenyList. Depending on the usage, they can either contain a small subset of processes in Π or they can contain the whole set of processes of the system.

To express the proofs produced by a process p, we use an abstract language $\mathcal{L}_{\mathcal{A}}$ of the complexity class $\mathcal{NP}$, which depends on a set $\mathcal{A}$. This language will be specified for the AllowList and the DenyList objects in Section 5 and Section 6. The idea is that p produces a proof π about a value $v \in \mathcal{S}$. A PROVE invocation by a process p is valid only if the proof π added to the *proofs* array is valid. The proof π is valid if $v \in \mathcal{L}_{\mathcal{A}}$ – i.e., v is a solution to the instance of the problem expressed by $\mathcal{L}_{\mathcal{A}}$, where $\mathcal{L}_{\mathcal{A}}$ is a language of the complexity class

$\mathcal{NP}$ [6] which depends on a subset $\mathcal{A}$ of the *listed-values* array ($\mathcal{A} \subseteq \mathcal{S}$). We note $\mathcal{P}_{\mathcal{L}_{\mathcal{A}}}$ the set of valid proofs relative to the language $\mathcal{L}_{\mathcal{A}}$. $\mathcal{P}_{\mathcal{L}_{\mathcal{A}}}$ can either represent Zero Knowledge Proofs or explicit proofs.

If a proof π is valid, then the PROVE operation returns $(\mathcal{A}, \mathsf{Acc}.\mathit{Prove}(v, \mathcal{A}))$, where $\mathsf{Acc}.\mathit{Prove}(v, \mathcal{A})$ is the proof generated by the operation, and where $\mathcal{A}$ is a subset of values in *listed-values* on which the proof was applied. Otherwise, the PROVE operation returns “False”. Furthermore, the *proofs* array also stores the name of the processes that invoked PROVE operations.

Formally, the PROOF-LIST object type is defined by the tuple (Q, Q_0, O, R, Δ), where:

- The set of valid state is $Q = (\mathit{listed\text{-}values} = \{x \in \mathcal{S}\}, \mathit{proofs} = \{(p \in \Pi, \widehat{\mathcal{S}} \subseteq \mathcal{S}, \mathsf{P} \in \mathcal{P}_{\mathcal{L}_{\widehat{\mathcal{S}}}})\})$, where *listed-values* is a subset of $\mathcal{S}$ and *proofs* is a set of tuples. Each tuple in *proofs* consists of a proof associated with the set it applies to and to the identifier of the process that issued the proof;
- The set of valid initial states is $Q_0 = (\emptyset, \emptyset)$, the state where the *listed-values* and the *proofs* arrays are empty;
- The set of possible operation is $O = \{\text{APPEND}(x), \text{PROVE}(y), \text{READ}()\}$, with $x, y \in \mathcal{S}$;
- The set of possible responses is $R = \left\{\text{True, False, } (\widehat{\mathcal{S}} \subseteq \mathcal{S}, \mathsf{P} \in \mathcal{P}_{\mathcal{L}_{\widehat{\mathcal{S}}}}), \{(p \in \Pi, \widehat{\mathcal{S}}' \subseteq \mathcal{S}, \mathsf{P}' \in \mathcal{P}_{\mathcal{L}_{\widehat{\mathcal{S}}}})\}\right\}$, where True is the response to a successful APPEND operation, $(\widehat{\mathcal{S}}, \mathsf{P})$ is the response to a successful PROVE operation, $\{(p, \widehat{\mathcal{S}}', \mathsf{P}')\}$ is the response to a READ operation, and False is the response to a failed operation; and
- The transition function is Δ. The PROOF-LIST object type supports 5 possible transitions. We define the 5 possible transitions of Δ in Table 1.

The first transition of the Δ function models a valid APPEND invocation, a value $y \in \mathcal{S}$ is added to the *listed-values* array by a process in the managers' set Π_M. The second transition of the Δ function represents a failed APPEND invocation. Either the process p_i that invokes this function is not authorized to modify the *listed-values* array, i.e., $p_i \notin \Pi_M$, or the value it tries to append is invalid, i.e., $y \notin \mathcal{S}$. The third transition of the Δ function captures a valid PROVE operation, where a valid proof is added to the *proofs* array. The function C will be used to express the anti-flickering property of the DenyList implementation. It is a boolean function that outputs either 0 or 1. The fourth transition of the Δ function represents an invalid PROVE invocation. Either the proof is invalid, or the set on which the proof is issued is not a subset of the *listed-values* array. Finally, the fifth transition represents a READ operation. It returns the *proofs* array and does not modify the object's state.

The language $\mathcal{L}_{\mathcal{A}}$ does not directly depend on the *listed-values* array. Hence, the validity of a PROVE operation will depend on the choice of the set $\mathcal{A}$.

5 The consensus number of the AllowList object

This section provides an AllowList object specification based on the PROOF-LIST object. The specification is then used to analyze the consensus number of the object type.

We provide a specification of the AllowList object defined as a PROOF-LIST object, where $\mathsf{C}(y, \widehat{\mathcal{S}}) = 1$ and $\forall\ y \in \mathcal{S}, y \in \mathcal{L}_{\mathcal{A}} \Leftrightarrow (\mathcal{A} \subseteq \mathcal{S} \wedge y \in \mathcal{A})$.

[6] In this article, $\mathcal{L}_{\mathcal{A}}$ can be one of the following languages: a value v belongs to $\mathcal{A}$ (AllowList), or a value v does not belongs to $\mathcal{A}$ (DenyList).

In other words, y belongs to a set $\mathcal{A}$. Using the third transition of the Δ function, we can see that $\mathcal{A}$ should also be a subset of the *listed-values* array. Hence, this specification supports proofs of set-membership in *listed-values*. A PROOF-LIST object defined for such language follows the specification of the AllowList. To support this statement, we provide an implementation of the object.

To implement the AllowList object, Algorithm 1 uses two Atomic Snapshot objects. The first one represents the *listed-values* array, and the second represents the *proofs* array. These objects are arrays of N entries. Furthermore, we use a function "Proof" that on input of a set $\mathcal{S}$ and an element y outputs a proof that $y \in$ *listed-values*. This function is used as a black box, and can either output an explicit proof – an explicit proof can be the tuple $(y, \mathcal{A})$, where $\mathcal{A} \subseteq$ *listed-values* – or a Zero Knowledge Proof.

Algorithm 1 Implementation of an AllowList object using Atomic-Snapshot objects.

```
Shared variables
  AS-LV ← $N$-dimensions Atomic-Snapshot object, initially $\{\emptyset\}^N$;
  AS-PROOF ← $N$-dimensions Atomic-Snapshot object, initially $\{\emptyset\}^N$;
Operation APPEND($v$) is
 1: If $(v \in \mathcal{S}) \wedge (p \in \Pi_M)$ then
 2:    local-values ← AS-LV.Snapshot()[$p$];
 3:    AS-LV.Update(local-values $\cup$ $v$, $p$);
 4:    Return true;
 5: Else return false;
Operation READ() is
 6: Return AS-PROOF.Snapshot();
Operation PROVE($v$) is
 7: If $p \notin \Pi_V$ then
 8:    Return false;
 9: $\mathcal{A}$ ← AS-LV.Snapshot();
10: If $v \in \mathcal{A}$ then
11:    $\pi_{\text{set-memb}}$ ← Proof($v \in \mathcal{A}$);
12:    proofs ← AS-PROOF.Snapshot()[$p$];
13:    AS-PROOF.Update(proofs $\cup$ $(p, \mathcal{A}, \pi_{\text{set-memb}})$, $p$);
14:    Return $(\mathcal{A}, \pi_{\text{set-memb}})$;
15: Else return false.
```

▶ **Theorem 4.** *Algorithm 1 wait-free implements an AllowList object.*

Proof. The complete proof of this theorem is given in the full version of this paper [19]. ◀

▶ **Corollary 5.** *The consensus number of the AllowList object type is* 1.

6 The consensus number of the DenyList object

In the following, we propose two wait-free implementations establishing the consensus number of the DenyList object type. In this section and in the following, we refer to a DenyList with $|\Pi_V| = k$ as a k-DenyList object. This analysis of this parameter k is the core of the study conducted here. Because it is a statically defined parameter, the knowledge of this parameter can improve efficiency of DenyList implementation by reducing the number of processes that need to synchronize in order to conduct a proof.

6.1 Lower bound

Algorithm 2 presents an implementation of a k-consensus object using a k-DenyList object with $\Pi_M = \Pi_V = \Pi$, and $|\Pi| = k$. It uses an Atomic Snapshot object, AS-LIST, to allow processes to propose values. AS-LIST serves as a helping mechanism [12]. In addition, the algorithm uses the progress and the anti-flickering properties of the PROVE operation of the k-DenyList to enforce the k-consensus agreement property. The PROPOSE operation operates as follows. First, a process p tries to prove that the element 0 is not revoked by invoking PROVE(0). Then, if the previous operation succeeds, p revokes the element 0 by invoking APPEND(0). Then, p waits for the APPEND to be effective. This verification is done by invoking multiple PROVE operations until one is invalid. This behavior is ensured by the progress property of the k-DenyList object. Once the progress has occurred, p is sure that no other process will be able to invoke a valid PROVE(0) operation. Hence, p is sure that the set returned by the READ operation can no longer grow. Indeed, the READ operation returns the set of valid PROVE operation that occurred prior to its invocation. If no valid PROVE(0) operation can be invoked, the set returned by the READ operation is fixed (with regard to the element 0). Furthermore, all the processes in Π share the same view of this set.

Finally, p invokes READ() to obtain the set of processes that invoked a valid PROVE(0) operation. The response to the READ operation will include all the processes that invoked a valid PROVE operation, and this set will be the same for all the processes in Π that invoke the PROPOSE operation. Therefore, up to line 7, the algorithm solved the set-consensus problem. To solve consensus, we use an additional deterministic function $f_i : \Pi^i \to \Pi$, which takes as input any set of size i and outputs a single value from this set.

To simplify the representation of the algorithm, we also use the separator() function, which, on input of a set of proofs $(\{(p \in \Pi, \{\widehat{\mathcal{S}} \subseteq \mathcal{S}, \mathsf{P} \in \mathcal{P}_{\mathcal{L}_\mathcal{S}})\})$, outputs *processes*, the set of processes which conducted the proofs, i.e. the first component of each tuple.

Algorithm 2 k-consensus implementation using one k-DenyList object and one Atomic Snapshot.

```
Shared variables
  k-dlist ← k-DenyList object;
  AS-LIST ← Atomic Snapshot object, initially $\{\emptyset\}^k$
Operation PROPOSE($v$) is
1:  AS-LIST.update($v, p$);
2:  k-dlist.PROVE(0);
3:  k-dlist.APPEND(0);
4:  Do
5:    ret ← k-dlist.PROVE(0);
6:  Until (ret ≠ false);
7:  processes ← separator(k-dlist.READ());
8:  Return AS-LIST.Snapshot()[$f_{|processes|}(processes)$].
```

▶ **Theorem 6.** *Algorithm 2 wait-free implements a k-consensus object.*

Proof. Let us fix an execution E of the algorithm presented in Algorithm 2. The progress property of the k-DenyList object ensures that the while loop in line 4 consists of a finite number of iterations – an APPEND(0) is invoked prior to the loop, hence, the PROVE(0) operation will eventually be invalid. Each invocation of the PROPOSE operation is a sequence of a finite number of local operations, Atomic Snapshot object accesses and k-DenyList object accesses which are assumed atomic. Therefore, each process terminates the PROPOSE operation in a finite number of its own steps. Let H be the history of E. We define $\bar{H}$ the completed history of H, where an invocation of PROPOSE which did not reach line 8 is completed with a line "return false". Line 8 is the linearization point of the algorithm. For convenience, any PROPOSE invocation that returns false is called an failed invocation. Otherwise, it is called a successful invocation.

We now prove that all operations in $\bar{H}$ follow the k-consensus specification:

- The process p that invoked a failed PROPOSE operation in $\bar{H}$ is faulty – by definition, the process prematurely stopped before line 8. Therefore, the fact that p cannot decide does not impact the termination nor the agreement properties of the k-consensus object.
- A successful PROPOSE operation returns AS-LIST.Snapshot()[$f_{|processes|}(processes)$]. Furthermore, a process proposed this value in line 1. All the processes that invoke PROPOSE conduct an APPEND(0) operation, and wait for this operation to be effective using the while loop at line 4 to 6. Thanks to the anti-flickering property of the k-DenyList object, when the APPEND operation is effective for one process – i.e. the Progress happens, in other words,a PROVE(0) operation is invalid – , then it is effective for any other process that would invoke the PROVE(0) operation. Hence, thanks to the

anti-flickering property, when a process obtains an invalid response from the PROPOSE(0) operation at line 5, it knows that no other process can invoke a valid PROVE(0) operation. This implies that the READ operation conducted at line 7 will return a fix set of processes, and all the processes that reach this line will see the same set. Furthermore, because each process invokes a PROPOSE(0) before the APPEND(0) at line 3, at least one valid PROPOSE(0) operation was invoked. Therefore, the *processes* set is not empty. Because each process ends up with the same set *processes*, and thanks to the determinism of the function f_i, all correct processes output the same value v (Agreement property and non-trivial value). The value v comes from the Atomic Snapshot object, composed of values proposed by authorized processes (Validity property). Hence a successful PROPOSE operation follows the k-consensus object specification.

All operations in $\bar{H}$ follow the k-consensus specification. To conclude, the algorithm presented in Algorithm 2 is a wait-free implementation of the k-consensus object type. ◀

▶ **Corollary 7.** *The consensus number of the k-DenyList object type is at least k.*

6.2 Upper bound

This section provides a DenyList object specification based on the PROOF-LIST object. The specification is then used to analyze the upper bound on the consensus number of the object type.

We provide an instantiation of the DenyList object defined as a PROOF-LIST object, where $\forall y \in \mathcal{S}, y \in \mathcal{L}_{\mathcal{A}} \Leftrightarrow (\mathcal{A} \subseteq \mathcal{S} \wedge y \notin \mathcal{A})$ and where:

$$\mathsf{C}(y, \widehat{\mathcal{S}}) = \begin{cases} 1, & \text{if } \forall \mathcal{A}' \in \widehat{\mathcal{S}}, y \notin \mathcal{A}' \\ 0, & \text{otherwise.} \end{cases}$$

In other words, the first equation ensures that y does not belong to a set $\mathcal{A}$, while the second equation ensures that the object fulfills the anti-flickering property. Hence, this instantiation supports proofs of set-non-membership in *listed-values*. A PROOF-LIST object defined for such language follows the specification of the DenyList. To support this statement, we provide an implementation of the object.

To build a k-DenyList object which can fulfill the anonymity property, it is required to build an efficient helping mechanism that preserves anonymity. It is impossible to disclose directly the value proven without disclosing the user's identity. Therefore, we assume that a process p that invokes the PROVE(v) operation can deterministically build a cryptographic commitment to the value v. Let C_v be the commitment to the value v. Then, any process $p' \neq p$ that invokes PROVE(v) can infer that C_v was built using the value v. However, a process that does not invoke PROVE(v) cannot discover to which value C_v is linked. If the targeted application does not require the user's anonymity, it is possible to use the plaintext v as the helping value.

Algorithm 3 presents an implementation of a k-DenyList object using k-consensus objects and Atomic Snapshots. The APPEND and the READ operations are analogous to those of Algorithm 1.

On the other hand, the PROVE operation must implement the anti-flickering property. To this end, a set of k-consensus objects and a helping mechanism based on commitments are used.

When a process invokes the PROVE(v) operation, it publishes C_v, the cryptographic commitment to v, using an atomic snapshot object. This commitment is published along with a timestamp [28] defined as follow. A local timestamp (p, c) is constituted of a process identifier p and a local counter value c. The counter c is always incremented before being reused. Therefore, each timestamp is unique. Furthermore, we build the strict total order relation $\mathcal{R}$ such that $(p, c)\mathcal{R}(p', c') \Leftrightarrow (c < c') \vee ((c = c') \wedge (p < p'))$. The timestamp is used in coordination with the helping value C_v to ensure termination. A process p that invokes the PROVE(v) operation must parse all the values proposed by the other processes. If a PROVE(v') operation was invoked by a process p' earlier than the one invoked by p – under the relation $\mathcal{R}$ – , then p must affect a set "val" for the PROVE operation of p' via the consensus object. The set "val" is obtained by reading the AS-LV object. The AS-LV object is append-only – no operation removes elements from the object. Furthermore, the sets "val" are attributed via the consensus object. Therefore, this mechanism ensures that the sets on which the PROVE operations are applied always grow.

Furthermore, processes sequentially parse the CONS-ARR using the counter$_p$ variable. This behavior, in collaboration with the properties of the consensus, ensures that all the process see the same tuples (winner, val) in the same order.

Finally, if a process p observes that a PROVE operation conducted by a process $p' \neq p$ is associated to a commitment C_v equivalent to the one proposed by p, then p produces the proof of set-non-membership relative to v and the set "val" affected to p' in its name. We consider that a valid PROVE operation is linearized when this proof of set-non-membership is added to AS-PROOF in line 19. Hence, when p produces its own proof – or if another process produces the proof in its name – it is sure that all the PROVE operations that are relative to v and that have a lower index in CONS-ARR compared to its own are already published in the AS-PROOF Atomic Snapshot object. Therefore, the anti-flickering property is ensured. Indeed, because the affected sets "val" are always growing and because of the total order induced by the CONS-ARR array, if p reaches line 25, it previously added a proof to AS-PROOF in the name of each process $p' \neq p$ that invoked a PROVE(v) operation and that was attributed a set at a lower index than p in CONS-ARR. Hence, the operation of p' was linearized prior to the operation of p.

A PROVE operation can always be identified by its published timestamp. Furthermore, when a proof is added to the AS-PROOF object, it is always added to the index counter$_{p_w}$. Therefore, if multiple processes execute line 19 for the PROVE operation labeled counter$_{p_w}$, the AS-PROOF object will only register a unique value.

Furthermore, we use a function "Proof" that on input of a set $\mathcal{S}$ and an element x outputs a proof that $x \notin \mathcal{S}$. This function is used as a black box, and can either output an explicit proof – an explicit proof can be the tuple $(x, \mathcal{S})$ – , or a Zero Knowledge Proof.

▶ **Theorem 8.** *Algorithm 3 wait-free implements a k-DenyList object.*

Proof. The proof of this theorem is given in the full version of this paper [19]. ◀

The following corollary follows from Theorem 6 and Theorem 8.

▶ **Corollary 9.** *The k-DenyList object type has consensus number k.*

7 Discussion

This section presents several applications where the AllowList and the k-DenyList can be used to determine the consensus numbers of more elaborate objects. More importantly, the analysis of the consensus number of these use cases makes it possible to determine if actual implementations achieve optimal efficiency in terms of synchronization. If not, we use the knowledge of the consensus number of the AllowList and DenyList objects to give intuitions

Algorithm 3 k-DenyList implementation using k-consensus objects and Atomic Snapshot objects.

```
Shared variables
  AS-LV ← $N$-dimensions Atomic-Snapshot object, initially $\{\emptyset\}^N$;
  AS-Queue ← $N$-dimensions Atomic-Snapshot object, initially $\{\emptyset\}^N$;
  CONS-ARR$_p$ ← an array of $k$-consensus objects of size $l > 0$;
  AS-PROOF ← $l$-dimensions Atomic-Snapshot object, initially $\{\emptyset\}^l$;
Local variables
  For each $p \in \Pi_V$ :
    evaluated$_p$ ← an array of size $l > 0$, initially $\{\emptyset\}^l$;
    counter$_p$ ← a positive integer, initially 0;
Operation APPEND($v$) is
 1:  If $(v \in \mathcal{S}) \wedge (p \in \Pi_M)$ then
 2:     local-values ← AS-LV.Snapshot()[$p$];
 3:     AS-LV.UPDATE(local-values $\cup$ $v$, $p$);
 4:     Return true;
 5:  Else return false;
Operation PROVE($v$) is
 6:  If $p \notin \Pi_V$ then
 7:     Return false;
 8:  $C_v$ ← Commitment($v$);
 9:  cnt ← counter$_p$;
10:  AS-Queue.UPDATE(((cnt, $p$), $C_v$), $p$);
11:  queue ← AS-Queue.Snapshot() \ evaluated$_p$;
12:  While (cnt, $p$) $\in$ queue do
13:     oldest ← the smallest clock value in queue under $\mathcal{R}$;
14:     prop ← (oldest, AS-LV.snapshot());
15:     (winner, val) ← CONS-ARR[counter$_p$].propose(prop);
16:     ((counter$_{p_w}$, $p_w$), $C^*$) ← winner;
17:     If $C^* = C_v \wedge v \notin$ val then
18:        $\pi_{SNM}$ ← Proof($v \notin val$);
19:        AS-PROOF.Update(($p_w$, val, $\pi_{SNM}$, winner), counter$_{p_w}$);
20:     evaluated$_p$ ← evaluated$_p$ $\cup$ winner;
21:     queue ← queue \ winner;
22:     counter$_p$ ← counter$_p$ + 1;
23:  If $v \notin$ val then
24:     Return (val, $\pi_{\mathrm{SNM}}$);
25:  Else return false;
Operation READ() is
26:  Return AS-PROOF.Snapshot();
```

on how to build more practical implementations. More precisely, the fact that the consensus numbers of AllowList and DenyList objects are (in most cases) smaller than n implies that most implementations can reduce the number of processes that need to synchronize in order to implement such distributed objects. The liveness of many consensus algorithms is only ensured when the network reaches a synchronous period. Therefore, reducing the number of processes that need to synchronize can increase the system's probability of reaching such synchronous periods. Thus, it can increase the effectiveness of such algorithms.

7.1 Revocation of a verifiable credential

We begin by analyzing Sovrin's Verifiable-Credential revocation method using the DenyList object [18]. Sovrin is a privacy-preserving Distributed Identity Management System (DIMS). In this system, users own credentials issued by entities called issuers. A user can employ one such credential to prove to a verifier they have certain characteristics. An issuer may want to revoke a user's credential prematurely. To do so, the issuer maintains an append-only list of revoked credentials. When a user wants to prove that their credential is valid, they must provide to the verifier a valid ZKP of set-non-membership proving that their credential is not revoked, i.e. not in the DenyList. In this application, the set of managers Π_M consists solely of the credential's issuer. Hence, the proof concerns solely the verifier and the user. The way Sovrin implements this verification interaction is by creating an ad-hoc peer-to-peer consensus instance between the user and the verifier for each interaction. Even if the resulting DenyList has consensus number 2, Sovrin implements the APPEND operation using an SWMR stored on a blockchain-backed ledger (which requires synchronizing the N processes of the system). Our results suggest instead that Sovrin's revocation mechanism could be implemented without a blockchain by only using pairwise consensus.

7.2 The Anonymous Asset Transfer object

The anonymous asset transfer object is another application of the DenyList and the AllowList objects. As described in Appendix B, it is possible to use these objects to implement the asset transfer object described in [22]. Our work generalizes the result by Guerroui et al. [22]. Guerraoui et al. show that a joint account has consensus number k where k is the number of agents that can withdraw from the account. We can easily prove this result by observing that withdrawing from a joint account requires a denylist to record the already spent coins.

Nevertheless, our ZKP capable construction makes it possible to show that an asset transfer object where the user is anonymous, and its transactions are unlinkable also has consensus number k, where k is the number of processes among which the user is anonymous. The two main implementations of Anonymous Asset Transfer, ZeroCash and Monero [35, 7], use a blockchain as their main double spending prevention mechanism. While the former provides anonymity on the whole network, the second only provides anonymity among a subset of the processes involved in the system. Hence, this second implementation could reduce its synchronization requirements accordingly.

7.3 Distributed e-vote systems

Finally, another direct application of the DenyList object is the blind-signature-based e-vote system with consensus number k, k being the number of voting servers, which we present in the full version of this paper [19]. Most distributed implementations of such systems also use blockchains, whereas only a subset of the processes involved actually require synchronization.

8 Related Works

Bitcoin and blockchain. Even though distributed consensus algorithms were already largely studied [10, 29, 11, 4, 9], the rise of Ethereum – and the possibilities offered by its versatile smart contracts – led to new ideas to decentralized already known applications. Among those, e-vote and DIMS [18] are two examples.

Blockchains increased the interest in distributed versions of already existing algorithms. However, these systems are usually developed with little concern for the underlying theoretical basis they rely on. A great example lies in trustless money transfer algorithms or crypto money. The underlying distributed asset-transfer object was never studied until recently. A theoretical study proved that a secure asset-transfer algorithm does not need synchrony between network nodes [22]. Prior to this work, all proposed schemes used a consensus algorithm, which cannot be deterministically implemented in an asynchronous network [17]. The result is that many existing algorithms could be replaced by more efficient, Reliable Broadcast [9] based algorithms. This work leads to more efficient implementation proposal for money transfer algorithm [5]. Alpos et al. then extended this study to the Ethereum ERC20 smart contracts [3]. This last paper focuses on the asset-transfer capability of smart contracts. Furthermore, the object described has a dynamic consensus number, which depends on the processes authorized to transfer money from a given account. Furthermore, this work and the one from Guerraoui et al. [22] both analyze a specific object that is not meant to be used to find the consensus number of other applications. In contrast, our work aims to be used as a generic tool to find the consensus number of numerous systems.

E-vote. An excellent example of the usage of DenyList is to implement blind signatures-based e-vote systems [13]. A blind signature is a digital signature where the issuer can sign a message without knowing its content. Some issuer signs a cryptographic commitment – a cryptographic scheme where Alice hides a value while being bound to it [33] – to a message produced by a user. Hence, the issuer does not know the actual message signed. The user can then un-commit the message and present the signature on the plain-text message to a verifier. The verifier then adds this message to a DenyList. A signature present in the DenyList is no longer valid. Such signatures are used in some e-vote systems [20, 32]. In this case, the blind signature enables anonymity during the voting operation. This is the e-vote mechanism that we study in this article. They can be implemented using a DenyList to restrain a user from voting multiple times. This method is explored in the full version of this paper [19].

There exists two other way to provide anonymity to the user of an e-vote system. The first one is to use a MixNet [26, 25, 14]. MixNet is used here to break the correlation between a voter and his vote. Finally, anonymity can be granted by using homomorphic encryption techniques [6, 15].

Each technique has its own advantages and disadvantages, depending on the properties of the specific the e-vote system. We choose to analyze the blind signature-based e-vote system because it is a direct application of the distributed DenyList object we formalize in this paper.

Anonymous Money Transfer. Blockchains were first implemented to enable trustless money transfer algorithms. One of the significant drawbacks of this type of algorithm is that it only provides pseudonymity to the user. As a result, transfer and account balances can be inspected by anyone, thus revealing sensitive information about the user. Later developments proposed hiding the user's identity while preventing fraud. The principal guarantees are double-spending prevention – i.e., a coin cannot be transferred twice by the same user – and *ex nihilo* creation prevention – i.e., a user cannot create money. Zcash [7] and Monero [35] are the best representative of anonymous money transfer algorithms. The first one uses an AllowList to avoid asset creation and a DenyList to forbid double spending, while the second one uses ring signatures. We show in Appendix B that the DenyList and AllowList objects can implement an Anonymous Money Transfer object, and thus, define the synchronization requirements of the processes of the system.

9 Conclusion

This paper presented the first formal definition of distributed AllowList and DenyList object types. These definitions made it possible to analyze their consensus number. This analysis concludes that no consensus is required to implement an AllowList object. On the other hand, with a DenyList object, all the processes that can propose a set-non-membership proof must synchronize, which makes the implementation of a DenyList more resource intensive.

The definition of AllowList and DenyList as distributed objects made it possible to thoroughly study other distributed objects that can use AllowList and DenyList as building blocks. For example, we discussed authorization lists and revocation lists in the context of the Sovrin DIMS. We also provided several additional examples in the Appendix. In particular, we show in Appendix B that an association of DenyList and AllowList objects can implement an anonymous asset transfer algorithm and that this implementation is optimal in terms of synchronization power. This result can also be generalized to any asset transfer algorithm, where the processes act as proxies for the wallet owners. In this case, synchronization is only required between the processes that can potentially transfer money on behalf of a given wallet owner.

References

1 Ethereum name service documentation. online - https://docs.ens.domains/ - accessed 23/11/2022.

2 Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *JACM*, 40(4):873–890, September 1993. doi:10.1145/153724.153741.

3 Orestis Alpos, Christian Cachin, Giorgia Azzurra Marson, and Luca Zanolini. On the synchronization power of token smart contracts. In *41st IEEE ICDCS*, pages 640–651, 2021. doi:10.1109/ICDCS51616.2021.00067.

4 Pierre-Louis Aublin, Sonia Ben Mokhtar, and Vivien Quéma. Rbft: Redundant byzantine fault tolerance. In *IEEE 33rd International Conference on Distributed Computing Systems*, pages 297–306, 2013. doi:10.1109/ICDCS.2013.53.

5 Alex Auvolat, Davide Frey, Michel Raynal, and François Taïani. Money Transfer Made Simple: a Specification, a Generic Algorithm, and its Proof. *Bulletin European Association for Theoretical Computer Science*, 132, October 2020.

6 Olivier Baudron, Pierre-Alain Fouque, David Pointcheval, Jacques Stern, and Guillaume Poupard. Practical multi-candidate election system. In *PODC*, pages 274–283, 2001. doi:10.1145/383962.384044.

7 Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474, May 2014. doi:10.1109/SP.2014.36.

8 Daniel Benarroch, Matteo Campanelli, Dario Fiore, Kobi Gurkan, and Dimitris Kolonelos. Zero-knowledge proofs for set membership: Efficient, succinct, modular. In *Financial Cryptography and Data Security*. Springer Berlin Heidelberg, 2021.

9 Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987. doi:10.1016/0890-5401(87)90054-X.

10 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *OSDI '99*, pages 173–186, 1999.

11 Miguel Castro and Barbara Liskov. Proactive recovery in a Byzantine-Fault-Tolerant system. In *OSDI 2000*, October 2000. URL: https://www.usenix.org/conference/osdi-2000/proactive-recovery-byzantine-fault-tolerant-system.

12 Keren Censor-Hillel, Erez Petrank, and Shahar Timnat. Help! In *PODC '15*, pages 241–250, 2015. doi:10.1145/2767386.2767415.

13 David Chaum. Blind signatures for untraceable payments. In *Advances in Cryptology*, pages 199–203, 1983.

14 Michael R. Clarkson, Stephen Chong, and Andrew C. Myers. Civitas: Toward a secure voting system. *IEEE SSP*, pages 354–368, 2008.

15 Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. A secure and optimally efficient multi-authority election scheme. In *EUROCRYPT '97*, pages 103–118, 1997.

16 Gaby G. Dagher, Praneeth Babu Marella, Matea Milojkovic, and Jordan Mohler. Broncovote: Secure voting system using ethereum's blockchain. In *ICISSP*, 2018.

17 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985. doi:10.1145/3149.214121.

18 Sovrin Foundation. Sovrin: A protocol and token for self-sovereign identity and decentralized trust. Technical report, Sovrin Foundation, 2018.

19 Davide Frey, Mathieu Gestin, and Michel Raynal. The synchronization power (consensus number) of access-control objects: The case of allowlist and denylist, 2023. doi:10.48550/arXiv.2302.06344.

20 Atsushi Fujioka, Tatsuaki Okamoto, and Kazuo Ohta. A practical secret voting scheme for large scale elections. In *AUSCRYPT '92*, pages 244–251, 1993.

21 Jyoti Grover. Security of vehicular ad hoc networks using blockchain: A comprehensive review. *Vehicular Communications*, 34:100458, 2022. doi:10.1016/j.vehcom.2022.100458.

22 Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovič, and Dragos-Adrian Seredinschi. The consensus number of a cryptocurrency. In *PODC '19*, pages 307–316, 2019. doi:10.1145/3293611.3331589.

23 Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991. doi:10.1145/114005.102808.

24 Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

25 Markus Jakobsson, Ari Juels, and Ronald L. Rivest. Making mix nets robust for electronic voting by randomized partial checking. In *11th USENIX Security Symposium*, August 2002. URL: https://www.usenix.org/conference/11th-usenix-security-symposium/making-mix-nets-robust-electronic-voting-randomized.

26 Ari Juels, Dario Catalano, and Markus Jakobsson. Coercion-resistant electronic elections. In *WPES*, pages 61–70, 2005. doi:10.1145/1102199.1102213.

27 Harry A. Kalodner, Miles Carlsten, Paul Ellenbogen, Joseph Bonneau, and Arvind Narayanan. An empirical study of namecoin and lessons for decentralized namespace design. In *Workshop on the Economics of Information Security*, 2015.

28 Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM 21, (7), 558-565*, July 1978. URL: https://www.microsoft.com/en-us/research/publication/time-clocks-ordering-events-distributed-system/.

29 Leslie Lamport. The part-time parliament. In *ACM TOCS*, pages 133–169, 1998. doi:10.1145/279227.279229.

30 Ming K. Lim, Yan Li, Chao Wang, and Ming-Lang Tseng. A literature review of blockchain technology applications in supply chains: A comprehensive analysis of themes, methodologies and industries. *Computers and Industrial Engineering*, 154:107133, 2021. doi:10.1016/j.cie.2021.107133.

31 Nitin Naik and Paul Jenkins. uport open-source identity management system: An assessment of self-sovereign identity and user-centric data platform built on blockchain. In *IEEE International Symposium on Systems Engineering (ISSE)*, pages 1–7, 2020. doi:10.1109/ISSE49799.2020.9272223.

32 Miyako Ohkubo, Fumiaki Miura, Masayuki Abe, Atsushi Fujioka, and Tatsuaki Okamoto. An improvement on a practical secret voting scheme. In *Information Security*, pages 225–234, 1999.

33 Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Advances in Cryptology — CRYPTO '91*, pages 129–140, 1992.

34 Andreas Pfitzmann and Marit Hansen. Anonymity, unlinkability, undetectability, unobservability, pseudonymity, and identity management–a consolidated proposal for terminology. *Version v0*, 31, January 2007.

35 Nicolas van Saberhagen. Cryptonote v 2.0, October 2013.

A Variations on the *listed-values* array

In the previous sections, we assumed the *listed-values* array was append-only. Some use cases might need to use a different configuration for this array. In this section, we explore use cases where the *listed-values* array is no longer append-only.

Let us start by considering the simplest case in which processes can only remove the values they wrote themselves. This results in no conflicts between APPEND and REMOVE operations. The *listed-values* array can be seen as an array of $|\Pi_V|$ values. A process p_i can write the i-th index of the *listed-values* array. As only p_i can modify this value, there are no conflicts upon writing (append/remove). This allows us to easily add a REMOVE operation to an AllowList or DenyList object. In the case of the AllowList, this is particularly interesting because it effectively allows the AllowList to act as a DenyList. Let us assume the managers add all the elements of the universe of the possible identifiers to the AllowList in the first place. Then, this AllowList can implement a DenyList object, where the REMOVE operation of the AllowList is equivalent to the APPEND operation of the DenyList. The resulting AllowList with REMOVE needs an anti-flickering property to prevent concurrent PROVE operations from yielding conflicting results. This makes the AllowList with REMOVE equivalent to a DenyList object: its consensus number is k, where k is the number of processes in Π_V.

A more complex case arises when multiple processes can remove a written value. We associate each process p_i with a predefined authorization set $\mathcal{A}_i \subseteq \Pi_M$, defining which processes can APPEND or REMOVE on p_i's register. We always have $p_i \in \mathcal{A}_i$. If $p_j \in \mathcal{A}_i$, then p_j is allowed to "overwrite" (remove) anything p_i wrote. In this case, APPEND and REMOVE operation can conflict with each other and authorized processes need to synchronize when modifying the *listed-values* array. Specifically, let $k_{\text{AR}_i} = |\mathcal{A}_\rangle|$ be the number of processes that can modify the ith array position and let $k_{\text{AR}} = \mathsf{max}_i(k_{\text{AR}_i})$ be the largest value of k_{AR_i} over all the array positions. Then the consensus number of the APPEND and REMOVE operation is k_{AR_i}.

B Anonymous Asset-Transfer object type

Existing work by Guerraoui et al [22] and Auvolat [5] provides good insight into the problem of asset transfer, but it only studies pseudonymous systems, where all transactions can be linked to a single pseudonym. We now show how our formalization of AllowList and DenyList allows us to reason about anonymous and unlinkable asset transfer solutions [7, 35].

B.1 Problem formalization

The Asset-Transfer object type allows a set of processes to exchange assets via a distributed network. We reformulate the definition proposed by Guerraoui et al. [22]:

▶ **Definition 10.** The (pseudonymous) Asset-Transfer object type proposes two operations, TRANSFER and BALANCE. The object type is defined for a set Π of processes and a set $\mathcal{W}$ of accounts. An account is defined by the amount of assets it contains at time t. Each account is initially attributed an amount of assets equal to $v_0 \in \mathbb{Z}^{+*}$. We define a map $\mu : \mathcal{W} \to \{0,1\}^{|\Pi|}$ which associates each account to the processes that can invoke TRANSFER operations for these wallets. The Asset Transfer object type supports two operations, TRANSFER and BALANCE. When considering a TRANSFER(i, j, v) operation, $i \in \mathcal{W}$ is called the initiator, $j \in \mathcal{W}$ is called the recipient, and $v \in \mathbb{N}$ is called the amount transferred. Let $T(i, j)_t$ be the sum of all valid TRANSFER operations initiated by process i and received by process j before time t. These operations respect three properties:

- (Termination) TRANSFER and BALANCE operations always return if they are invoked by a correct process.
- (TRANSFER Validity) The validity of an operation TRANSFER(x, y, v) invoked at time t by a process p is defined in a recursive way. If no TRANSFER(x, i, v), $\forall i \in \mathcal{W}$ was invoked before time t, then the operation is valid if $v \leq v_0$ and if $p \in \mu(x)$. Otherwise, the operation is valid if $v \leq v_0 + \sum_{i\in\mathcal{W}} T(i, x)_t - \sum_{j\in\mathcal{W}} T(x, j)_t$ and if $p \in \mu(x)$.
- (BALANCE Validity) A BALANCE operation invoked at time t is valid if it returns $v_0 + \sum_{i\in\mathcal{W}} T(i, x)_t - \sum_{j\in\mathcal{W}} T(x, j)_t$ for each account x.

The Asset transfer object is believed to necessitate a double-spending-prevention property. This property is captured by the TRANSFER Validity property of Definition 10. Indeed, the double-spending-prevention property is defined to avoid ex-nihilo money creation. In a wait-free implementation, a valid transfer operation is atomic. Therefore, double spending is already prevented. A TRANSFER operation takes into account all previous transfers from the same account.

The paper by Guerraoui et al. [22] informs us that the consensus number of such an object depends on the map μ. If $\sum_{i \in \{0, \cdots, |\Pi|\}} \mu(w)[i] \leq 1, \forall\ w \in \mathcal{W}$, then the consensus number of the object type is 1. Otherwise, the consensus number is $\max_{w \in \mathcal{W}}(\sum_{i \in \{0, \cdots, |\Pi|\}} \mu(w)[i])$. In other words, the consensus number of such object type is the maximum number of different processes that can invoke a TRANSFER operation on behalf of a given wallet.

From continuous balances to token-based Asset-Transfer

The definition proposed by Guerraoui et al. uses a continuous representation of the balance of each account. Implementing anonymous money transfer with such a representation would require a mechanism to hide the transaction amounts [7]. As such a mechanism would not affect the synchronization properties of the AAT object, we simplify the problem by considering a token-based representation. A transfer in the tokenized version for a value of kV consists of k TRANSFER operations, each transferring a token of value V. The full version of the paper [19] provised a bijection that makes it possible to move to and from the continous and token-based representations.

Anonymity set

Let S be a set of actors. We define "anonymity" as the fact that, from the point of view of an observer, $o \notin S$, the action, v, of an actor, $a \in S$, cannot be distinguished from the action of any other actor, $a' \in S$. We call S the anonymity set of a for the action v [34].

Implementing Anonymous Asset Transfer requires hiding the association between a token and the account or process that owns it. If a "token owner" transfers tokens from the same account twice, these two transactions can be linked together and are no longer anonymous. Therefore, we assume that the "token owner" possesses offline proofs of ownership of tokens. These proofs are associated with shared online elements, allowing other processes to verify the validity of transactions. We call *wallet* the set of offline proofs owned by a specific user. We call the individual who owns this wallet the *wallet owner*. A wallet owner can own multiple wallets, while a wallet is owned by only one owner. Furthermore, we assume each process can invoke TRANSFER operations on behalf of multiple wallet owners. Otherwise, a single process, which is in most cases identified by its ip-address or its public key, would be associated with a single wallet and the system could not be anonymous. With the same reasoning, we can assume that a wallet owner can request many processes to invoke a TRANSFER operation on his or her behalf. Otherwise, the setup would not provide "network anonymity", but only "federated anonymity", where the wallet is anonymous among all other wallets connected to this same process. In our model, processes act as proxies.

The Anonymous Asset-Transfer object type

The first difference between a Pseudonymous Asset Transfer object type and an anonymous one is the absence of a BALANCE operation. The wallet owner can compute the balance of its own wallet using a LOCALBALANCE function that is not part of the distributed object. The TRANSFER operation is also slightly modified. Let us consider a sender that wants to transfer a token T_O to a recipient. The recipient creates a new token T_R with the associated cryptographic offline proofs (in practice, T_R can be created by the sender using the public key of the recipient). Specifically, it associates it with a private key. This private key is known only to the recipient: its knowledge represents, in fact, the possession of the token. Prior to the transfer operation, the recipient sends token T_R to the sender. The sender destroys token T_O and activates token T_R. The destruction prevents double spending, and the creation makes it possible to transfer the token to a new owner while hiding the recipient's identity. Furthermore, this process of destruction and creation makes it possible to unlink the usages of what is ultimately a unique token.

Each agent maintains a local wallet that contains the tokens (with the associated offline proofs) owned by the agent. The owner of a wallet w can invoke TRANSFER operations using any of the processes in $\mu(w)$. A transfer carried out from a process p for wallet w is associated with an anonymity set $\mathcal{AS}_p^w$ of size equal to the number of wallets associated with process p: $|\mathcal{AS}_p^w| = \sum_{i \in \mathcal{W}} \mu(i)[p]$. The setup with the maximal anonymity set for each transaction is an Anonymous Asset Transfer object where each wallet can perform a TRANSFER operation from any process: i.e., $\mu(i) = \{1\}^{|\Pi|}, \forall i \in \mathcal{W}$. The token-based Anonymous Asset Transfer object type is defined as follows:

▶ **Definition 11.** The Anonymous Asset Transfer object type supports only one operation: the TRANSFER operation. It is defined for a set Π of processes and a set $\mathcal{W}$ of wallets. An account is defined by the amount of tokens it controls at time t. Each account is initially attributed an amount v_0 of tokens. We define a map $\mu : \mathcal{W} \to \{0,1\}^{|\Pi|}$ which associates each wallet to the processes that can invoke TRANSFER on behalf of these wallets. When considering a TRANSFER(T_O, T_R) operation, T_0 is the cryptographic material of the initiator that proves the existence of a token T, and T_R is the cryptographic material produced by the recipient used to create a new token. The TRANSFER operation respects three properties:

- (Termination) The TRANSFER operation always returns if it is invoked by a correct process.
- (TRANSFER Validity) A TRANSFER(T_O, T_R) operation invoked at time t is valid if:
  - (Existence) The token T_O already existed before the transaction, i.e., either it is one of the tokens initially created, or it has been created during a valid TRANSFER(T'_O, T_O) operation invoked at time $t' < t$.
  - (Double spending prevention) No TRANSFER$(T_O,\ T'_R)$ has been invoked at time $t'' < t$.
- (Anonymity) A TRANSFER(T_O, T_R) invoked by process p does not reveal information about the owner w and w' of T_O and T_R, except from the fact that w belongs to the anonymity set $\mathcal{AS}_p^w$.

The TRANSFER validity property implies that the wallet owner can provide existence and non-double-spending proofs to the network. It implies that any other owner in the same anonymity set and with the same cryptographic material (randomness and associated element) can require the transfer of the same token. We know the material required to produce a TRANSFER proof is stored in the wallet. Furthermore, we can assume that all the randomness used by a given wallet owner is produced by a randomness Oracle that derives a seed to obtain random numbers. Each seed is unique to each wallet. We assume the numbers output by an oracle seem random to an external observer, but two processes that share the same seed will obtain the same set of random numbers in the same order.

A transaction must be advertised to other processes and wallet owners via the TRANSFER operation. Therefore, proofs of transfer are public. We know these proofs are deterministically computed thanks to our deterministic random oracle model. Furthermore, only one sender and recipient are associated with each transfer operation. Therefore, the public proof cryptographically binds (without revealing them) the sender to the transaction. Hence, the public proof is a cryptographic commitment, which can be opened by the sender or any other actor who knows the same information as the sender.

In order to study the consensus number of this object, we consider that wallet owners can share their cryptographic material with the entire network, thereby giving up their anonymity. This would not make any sense in an anonymous system, but it represents a valuable tool to reason about the consensus number of the object. This sharing process can be implemented by an atomic register (and therefore has no impact on the consensus number).

Processes can derive the sender's identity from the shared information using a local "uncommit" function. The "uncommit" function takes as input an oracle, a random seed, token elements, and an "on-ledger" proof of transfer of a token and outputs a wallet owner ID if the elements are valid. Otherwise, it outputs $\emptyset$.

B.2 Consensus number of the Anonymous Asset-Transfer object type

Lower bound

Algorithm 4 presents an algorithm that implements a k-consensus object, using only k-Anonymous Asset Transfer objects and SWMR registers. The k in k-Anonymous Asset Transfer object refers here to the size of the biggest $\mu(w), \forall\ w \in \mathcal{W}$.

Algorithm 4 Implementation of a k-consensus object using k-Anon-AT objects.

```
Shared variables:
    AT ← $k$-Anonymous-AT object, initialized with $k+1$ wallets,
            each one of the $k$ first wallets possesses the elements
            necessary to transfer one shared token, the $k+1$-th
            wallet is the recipient of the transfers, it is not controlled
            by any process;
    RM-LEDGER ← Atomic Snapshot object, initially $\{\emptyset\}^k$;
    V-LED ← Atomic Snapshot object, initially $\{\emptyset\}^k$;
    O ← A random oracle;
    TokenMat ← secret associated with a unique token;
Local variables:
    seed ← random number;

Operation PROPOSE($v$) is:
1:  RM-LEDGER[p].update(seed, $p$);
2:  V-LED[p].update($v, p$);
3:  res ← AT.transfer(TokenMat, O, seed, $k+1$);
4:  RML ← RM-LEDGER.snapshot();
5:  VL ← V-LED.snapshot();
6:  For $i$ in $\{1, \cdots, k\}$ do:
7:      If uncommit(O, RML[$i$], TokenMat, res) $\neq \emptyset$ then:
8:          Return VL[$i$];
9:  Return False;
```

▶ **Theorem 12.** *Algorithm 4 wait-free implements k-consensus.*

Proof. The proof of Theorem 12 is given in the full version of this paper [19]. ◀

Upper Bound

We give an implementation of the Anon-AT object using only Atomic Snapshot objects, DenyList objects, and AllowList objects. Each wallet owner can request a TRANSFER operation to k different processes. The proposed implementation uses disposable tokens that are either created at the initialization of the system or during the transfer of a token. When a token is destroyed, a new token can be created, and the new owner of the token is the only one to know the cryptographic material associated with this new token. In the following, we use the zero-knowledge version of the DenyList and AllowList object types, where all set-(non-)membership proofs use a zero-knowledge setup. In addition, we use an AllowList object to ensure that a token exists (no ex-nihilo creation), and we use a DenyList object to ensure that the token is not already spent (double-spending protection).

The underlying cryptographic objects used are out of the scope of this paper. However, we assume our implementation uses the ZeroCash [7] cryptographic implementation, which is a sound anonymous asset transfer algorithm. More precisely, we will use a high-level definition of their off-chain functions. It is important to point out that using the ZeroCash implementation, it is possible to transfer value from a pseudonymous asset transfer object to an anonymous one using a special transaction called "Mint". To simplify our construction, we assume that each wallet is created with an initial amount of tokens v_0 and that our object does not allow cross-chain transfers. We, therefore, have no "Mint" operation.

ZeroCash uses a TRANSFER operation called *pour* that performs a transfer operation destroying and creating the associated cryptographic material. Here, we use a modified version of *pour* which does not perform the transfer or any non-local operation. It is a black-box local function that creates the cryptographic material required prove the destruction of the source token (T_O) and the creation of the destination one (T_R). Our modified *pour* function takes as input the source token, the private key of the sender (sk_s), and the public key of the recipient (pk_r): $pour(T_O, \mathsf{pk}_r, \mathsf{sk}_s) \to tx$, tx being the cryptographic material that makes it possible to destroy T_O and create T_R.

There might be multiple processes transferring tokens concurrently. Therefore, we define a deterministic local function ChooseLeader$(\mathcal{A}, tx)$, which takes as input any set $\mathcal{A}$ and a transaction tx, and outputs a single participant p which invoked BL.PROVE(tx).[7]

Algorithm 5 Anon-AT object implementation using SWMR registers, AllowList objects, and DenyList objects.

```
Shared variables:
  DL ← $k$-DenyList object, initially $(\emptyset, \emptyset)$;
  AL ← AllowList object, initially $(\{(token_{(i,j)})_{i=1}^{t}\}_{j=1}^{k}, \emptyset)$
Operation TRANSFER($T_O, \mathsf{pk}_r, \mathsf{sk}_s$) is:
 1:  $tx$ ← Pour($T_O, \mathsf{pk}_r, \mathsf{sk}_s$)
 2:  If verify($tx$) and $tx \in$ AL and $tx \notin$ DL then:
 3:      AL.PROVE($tx$);
 4:      DL.PROVE($tx$);
 5:      DL.APPEND($tx$);
 6:      Do:
 7:          ret ← DL.PROVE(tx);
 8:      While ret $\neq$ false;
 9:      If ChooseLeader(DL.READ(), $tx.T_R$)=$p$ then:
10:          AL.append($tx.T_R$);
11:          Return $tx.T_R$;
12:  Return False;
```

▶ **Theorem 13.** *Algorithm 5 wait-free implements an Anon-AT object.*

Proof. The proof of Theorem 13 is given in the full version of this paper [19]. ◀

▶ **Corollary 14.** *The consensus number upper bound of a k-anon-AT object is k. Using this corollary and Theorem 12, we further deduct that k-anon-AT object has consensus number k.*

[7] In reality, the signature of chooseLeader would be more complicated as the function needs T_O, pk_r, sk_s in addition to tx. These additional elements make it possible to uncommit tx, thereby matching the values of the PROVE operation with $tx.T_R$. Note that this does not pose an anonymity threat as this is a local function invoked by the owner of sk_s. We omit these details to simplify the presentation.

List Defective Colorings: Distributed Algorithms and Applications

Marc Fuchs
University of Freiburg, Germany

Fabian Kuhn
University of Freiburg, Germany

Abstract

The distributed coloring problem is at the core of the area of distributed graph algorithms and it is a problem that has seen tremendous progress over the last few years. Much of the remarkable recent progress on deterministic distributed coloring algorithms is based on two main tools: a) defective colorings in which every node of a given color can have a limited number of neighbors of the same color and b) list coloring, a natural generalization of the standard coloring problem that naturally appears when colorings are computed in different stages and one has to extend a previously computed partial coloring to a full coloring.

In this paper, we introduce *list defective colorings*, which can be seen as a generalization of these two coloring variants. Essentially, in a list defective coloring instance, each node v is given a list of colors $x_{v,1}, \dots, x_{v,p}$ together with a list of defects $d_{v,1}, \dots, d_{v,p}$ such that if v is colored with color $x_{v,i}$, it is allowed to have at most $d_{v,i}$ neighbors with color $x_{v,i}$.

We highlight the important role of list defective colorings by showing that faster list defective coloring algorithms would directly lead to faster deterministic $(\Delta + 1)$-coloring algorithms in the LOCAL model. Further, we extend a recent distributed list coloring algorithm by Maus and Tonoyan [DISC '20]. Slightly simplified, we show that if for each node v it holds that $\sum_{i=1}^{p} \left(d_{v,i} + 1\right)^2 > \deg_G^2(v) \cdot \operatorname{poly}\log \Delta$ then this list defective coloring instance can be solved in a communication-efficient way in only $O(\log \Delta)$ communication rounds. This leads to the first deterministic $(\Delta + 1)$-coloring algorithm in the standard CONGEST model with a time complexity of $O(\sqrt{\Delta} \cdot \operatorname{poly}\log \Delta + \log^* n)$, matching the best time complexity in the LOCAL model up to a $\operatorname{poly}\log \Delta$ factor.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Mathematics of computing → Graph coloring

Keywords and phrases distributed coloring, list coloring, defective coloring

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.22

Related Version *Full Version*: `https://arxiv.org/abs/2304.09666` [15]

Funding This work was partially supported by the German Research Foundation (DFG) under the project number 491819048.

1 Introduction and Related Work

Distributed graph coloring is one of the core problems in the area of distributed graph algorithms. One typically assumes that the graph $G = (V, E)$ to be colored represents a communication network of n nodes with maximum degree Δ and that the nodes (or edges) of G must be colored in a distributed way by exchanging messages over the edges of G. The nodes typically interact with each other in synchronous rounds. If the size of messages is not restricted, this is known as the LOCAL model and if in every round, every node can send an $O(\log n)$-bit message to every neighbor, it is known as the CONGEST model [33]. The problem was first studied by Linial in a paper that pioneered the whole area of distributed graph algorithms [26]. Linial in particular showed that coloring a ring network with $O(1)$

colors (and thus coloring a graph with $f(\Delta)$ colors) requires $\Omega(\log^* n)$ rounds and that in $O(\log^* n)$ rounds, one can color any graph with $O(\Delta^2)$ colors. Subsequently, there has been a plethora of work on distributed coloring algorithms, e.g., [19, 1, 31, 36, 32, 25, 11, 7, 12, 5, 14, 22, 13, 24, 30, 29, 18, 20, 21]. The most standard variant of the distributed coloring problem asks for a proper coloring of the nodes V of G with $\Delta + 1$ colors. Note that this is what can be achieved by a simple sequential greedy algorithm.

Over the last approximately 15 years, we have seen remarkable progress on randomized and on deterministic distributed coloring algorithms. Much of the progress on deterministic algorithms (which are the focus of the present paper) has been achieved by studying and using two generalizations of the standard coloring problem, *defective colorings* and *list colorings*. In the following, we briefly discuss the history and significance of defective colorings and of list colorings in the context of deterministic distributed coloring algorithms. For lack of space, we do not discuss, the very early work on deterministic distributed coloring [26, 19, 36, 25], the deterministic algorithms that directly result from computing network decomposition [1, 32, 34, 17, 16], or the vast literature on randomized distributed coloring algorithms, e.g., [28, 35, 12, 22, 13, 20, 21].

Defective Coloring. Given integers $d \geq 0$ and $c > 0$, a d-defective c-coloring of a graph $G = (V, E)$ is an assignment of colors $\{1, \ldots, c\}$ to the nodes in V such that the subgraph induced by each color class has a maximum degree of at most d [27]. Defective colorings were introduced to distributed algorithms independently by Barenboim and Elkin [6] and by Kuhn [23] in 2009. Both papers give distributed algorithms to compute d-defective colorings with $O(\Delta^2/d^2)$ colors. The algorithm of [23] extends the classic $O(\Delta^2)$-coloring algorithm by Linial to achieving this in $O(\log^* n)$ time. Both papers use defective colorings to compute proper colorings in a divide-and-conquer fashion, leading to algorithms to compute a $(\Delta + 1)$-coloring in $O(\Delta + \log^* n)$ rounds.[1] This idea was pushed further by Barenboim and Elkin in [7] and [8]. In [7], they introduce the notion of *arbdefective colorings*: Instead of decomposing a graph into color classes of bounded degree, the aim is to decompose a graph into color classes of bounded arboricity. More specifically, the output of an arbdefective c-coloring algorithm with arbdefect d is a coloring of nodes with colors $\{1, \ldots, c\}$ together with an orientation of the edges such that every node has at most d outneighbors of the same color. For this more relaxed version of defective coloring, they show that for a given oriented graph with maximum outdegree β, one can efficiently compute a d-arbdefective coloring with $O(\beta/d)$-colors (in time $O(\beta^2/d^2 \cdot \log n)$). Applying this recursively, for example allows us to obtain a $\Delta^{1+o(1)}$-coloring in time $\operatorname{poly}\log \Delta \cdot \log n$. In [8], it is shown that for a family of graphs that includes line graphs, one can efficiently compute a standard d-defective coloring with only $O(\Delta/d)$ colors in time $O(\Delta^2/d^2 + \log^* n)$. This in particular implies that a $\Delta^{1+o(1)}$-edge coloring can be computed in time $\operatorname{poly}\log \Delta + O(\log^* n)$. All the algorithms of [6, 23, 7, 8] use defective coloring in the following basic way. If a computed defective coloring has p colors, the space of available colors is divided into p parts that can then be assigned to the p color classes and handled in parallel on the respective lower degree/arboricity graphs. When doing this, one inherently has to use more than $\Delta + 1$ colors because in each defective coloring step, the maximum degree (or outdegree) goes down at a factor that is somewhat smaller than the number of colors of the defective coloring. In [6, 23], this is compensated by reducing the number of colors at the end of each recursion level. However, this leads to algorithms with time complexity at least linear in Δ.

[1] Earlier algorithms were based on simple round-by-round color reduction schemes and required $O(\Delta^2 + \log^* n)$ [26, 19] and $O(\Delta \log \Delta + \log^* n)$ rounds [36, 25], respectively.

List Coloring. The key to obtaining $(\Delta+1)$-coloring algorithms with a better time complexity is to explicitly consider the more general $(\mathit{degree}+1)$-list coloring problem. In this problem, every node v receives a list L_v of at least $\deg(v)+1$ colors as input and an algorithm has to properly color the graph in such a way that each node v is colored with a color from its list L_v. Note that this problem can still be solved by a simple sequential greedy algorithm. Note also that the problem appears naturally when solving the standard $(\Delta+1)$-coloring problem in different phases. If a subset $S \subseteq V$ is already colored, then each node $v \in V \setminus S$ needs to be colored with a color that is not already taken by some neighbor in S. If v has degree Δ and all already colored neighbors of v have chosen different colors, the list of the remaining available colors for v is exactly of length $\deg(v)+1$. The first paper that explicitly considered list coloring in the context of deterministic distributed coloring is by Barenboim [5]. In combination with the improved arbdefective coloring algorithm of [10], the algorithm of the paper obtains a $(1+\varepsilon)\Delta$-coloring in $O(\sqrt{\Delta}+\log^* n)$ rounds by first computing a $O(\sqrt{\Delta})$-arbdefective $O(\sqrt{\Delta})$-coloring and by afterwards iterating over the $O(\sqrt{\Delta})$ color classes of this arbdefective coloring and solving the corresponding list coloring problem in $O(1)$ time. With the same technique, the paper also gets an $O(\Delta^{3/4}+\log^* n)$-round algorithm for $(\Delta+1)$-coloring. This algorithm also works in the CONGEST model, i.e., by exchanging messages of at most $O(\log n)$ bits. For algorithms with a round complexity of the form $f(\Delta)+O(\log^* n)$, this still is the fastest known $(\Delta+1)$-coloring algorithm in the CONGEST model. The algorithm was improved by Fraigniaud, Kosowski, and Heinrich [14]. In combination with the subsequent results of [10, 30], the algorithm of [14] leads to an $O(\sqrt{\Delta\log\Delta}+\log^* n)$-round distributed algorithm for $(\mathit{degree}+1)$-list coloring and thus also for $(\Delta+1)$-coloring. As one of the main results of this paper, we give a CONGEST algorithm that almost matches this and that has a time complexity of $O(\sqrt{\Delta}\operatorname{poly}\log\Delta+\log^* n)$. List colorings and defective colorings have also been explicitly used in all later deterministic distributed coloring algorithms [24, 4, 18, 3]. We next discuss an idea that was introduced in [24] and that is particularly important in the context of the present paper.

Distributed Color Space Reduction. The objective of [24] was to extend the coloring algorithms of [7, 8] to list colorings. The algorithms of [7, 8] are based on computing arbdefective or defective colorings to recursively divide the graph into low (out)degree parts that use disjoint sets of colors. This leads to fast coloring algorithms, however, the number of required colors grows exponentially with the number of recursion levels. While it is not clear how to efficiently turn a standard distributed coloring algorithm that uses significantly more than $\Delta+1$ colors into a $(\Delta+1)$-coloring algorithm, by using the techniques introduced in [5, 14], we can do this if we have a list coloring algorithm. Essentially, if we have a list coloring algorithm that uses lists of size $O(\alpha(\Delta+1))$, it can be turned into a $(\mathit{degree}+1)$-list coloring algorithm in only $\tilde{O}(\alpha^2)$ rounds (and in some cases even in $O(\alpha)$ rounds). However, if the nodes have different lists, a defective coloring does not easily split the graph into independent coloring problems. As a generalization of defective colorings, [24] introduces a tool called *color space reduction.* Assuming that all lists consist of colors of some color space $\mathcal{C}$. For a given partition of $\mathcal{C}$ into disjoint parts $\mathcal{C}_1,\dots,\mathcal{C}_p$, a color space reduction algorithm partitions the nodes V into p parts $V_1,\dots,V_p$ such that for every node v, with $v \in V_i$ and v has $\deg_i(v)$ neighbors in V_i, the algorithm tries to keep the ratio $|L_v \cap \mathcal{C}_i|/\deg_i(v)$ as close as possible to the initial list-degree ratio $|L_v|/\deg(v)$. In [24], it is shown that the arbdefective and defective coloring algorithms of [7, 8] can be generalized to compute a color space reduction. If the size of the color space is polynomial in Δ, this lead to $(\mathit{degree}+1)$-coloring algorithms with time complexities of $2^{O(\sqrt{\log\Delta})}\log n$ in general graphs and of $2^{O(\sqrt{\log n})}+O(\log^* n)$ in graphs of bounded neighborhood independence, a family of graphs that includes line graphs of bounded rank hypergraphs. The complexity of the $(\mathit{degree}+1)$-edge coloring problem was later improved to $(\log\Delta)^{O(\log\log\Delta)} + O(\log^* n)$ in [4] and to $\operatorname{poly}\log\Delta + O(\log^* n)$ in [3]. In both cases, this was achieved by designing better distributed color space reduction algorithms for line graphs. The $(\Delta+1)$-coloring algorithm of [24] for general graphs was later subsumed by a deterministic $O(\log^2\Delta\cdot\log n)$-round algorithm for the $(\Delta+1)$-coloring problem in [18].

List Defective Colorings. Color space reductions can be seen as a special case of the following list variant of defective colorings. Each node v has a list of possible colors that it can choose (e.g., which color subspace $\mathcal{C}_i$ to use). Depending on what color v chooses, it can tolerate different defects (e.g., depending on the size $|L_{v,i}\cap\mathcal{C}_i|$ of the remaining color list when choosing color subspace $\mathcal{C}_i$). In the following, we formally define *list defective colorings*. One of the objectives of this paper is to understand the relation of list defective colorings to each other and to other coloring problems, and we will see in particular that better list defective coloring algorithms can directly lead to better algorithms for standard coloring problems.

In a list defective coloring problem, as input, each node v obtains a *color list* $L_v \subseteq \mathcal{C}$, where $\mathcal{C}$ is the space of possible colors. Each node v further has a defect function $d_v : L_v \to \mathbb{N}_0$ that assigns a non-negative integral defect value to each color in v's list L_v. Given vertex lists L_v, a list vertex coloring is an assignment $\varphi : V \to \mathcal{C}$ that assigns each node $v \in V$ a color $\varphi(v) \in L_v$. In the following, we formally define three variants of list defective coloring.

▶ **Definition 1** (List Defective Coloring). *Let $G=(V,E)$ be a graph, let $\mathcal{C}$ be a color space, and assume that each node $v\in V$ is given a color list $L_v\subseteq\mathcal{C}$ and a defect function $d_v : L_v\to\mathbb{N}_0$. Further, assume that we are given a list vertex coloring $\varphi : V\to\mathcal{C}$.*

- *The coloring φ is a* list defective coloring *iff every $v\in V$ has at most $d_v(\varphi(v))$ neighbors of color $\varphi(v)\in L_v$.*
- *If G is a directed graph, φ is called an* oriented list defective coloring *iff every $v\in V$ has at most $d_v(\varphi(v))$ out-neighbors of color $\varphi(v)\in L_v$.*
- *In combination with an edge orientation σ, φ is called a* list arbdefective coloring *iff it is an oriented list defective coloring w.r.t. the directed graph induced by the edge orientation σ.*

An (oriented) list (arb)defective coloring is called an (oriented) p-list (arb)defective coloring *for some integer $p>0$ if for all $v\in V$, $|L_v|\le p$.*

Note that the difference between an oriented list defective coloring and a list arbdefective coloring is that in an oriented list defective coloring, the edge orientation of G is given as part of the input and in a list arbdefective coloring, the edge orientation is a part of the output. By a result of Lovász [27], it is well-known that a d-defective c-coloring of a graph G with maximum degree Δ always exists if $c(d+1)>\Delta$. Note that this condition is also necessary if $G=K_{\Delta+1}$. By computing a balanced orientation of the edges of each color class of such a coloring, one can also deduce that a d-arbdefective c-coloring always exists if $c(2d+1)>\Delta$. Again, this condition is necessary if $G=K_{\Delta+1}$. By generalizing the potential function argument of [27], in the full version of this paper [15], we prove that the natural generalization of both existential statements also holds for the respective list defective coloring variants. Specifically, we show that for given color lists L_v and defect functions d_v, a list defective coloring always exists if

$$\forall v\in V:\ \sum_{x\in L_v}\big(d_v(x)+1\big)>\Delta \tag{1}$$

and a list arbdefective coloring always exists if

$$\forall v \in V \;:\; \sum_{x \in L_v} \big(2d_v(x)+1\big) > \Delta. \tag{2}$$

Both conditions are necessary if the graph is a $(\Delta+1)$-clique and if all nodes have the same color list and the same defect function. For arbdefective colorings, it has further been shown in [2] that Condition (1) is necessary and sufficient to compute such colorings in time $f(\Delta)+O(\log^* n)$. Whenever (1) does not hold, there is an $\Omega(\log_\Delta n)$-round lower bound for deterministic distributed list arbdefective coloring algorithms.

1.1 Our Contributions

In the following, whenever we consider a graph $G=(V,E)$, we assume that n denotes the number of nodes of G, Δ denotes the maximum degree of G, and $\deg(v)$ denotes the degree of a node v. Also, if G is a directed graph, β_v refers to the outdegree of node v and β denotes the maximum outdegree. Further, if we discuss any list defective coloring problem, unless stated otherwise, we assume that the colors come from space $\mathcal{C}$, $L_v \subseteq \mathcal{C}$ denotes the list of node v, and d_v denotes the defect function of nodes v. We further assume that $\Lambda := \max_{v\in V} |L_v|$ denotes the maximum list size. As it is common in the distributed setting, we do not analyze the complexity of internal computations at nodes. We briefly discuss the complexity of internal computations for our algorithms in the full version [15].

Oriented List Defective Coloring. As our main technical contribution, we give an efficient deterministic distributed algorithm for computing oriented list defective colorings. This algorithm is an adaptation of the techniques developed by Maus and Tonoyan [30] to obtain a 2-round algorithm for proper vertex colorings in directed graphs of small maximum outdegree.

▶ **Theorem 2.** *Let $G=(V,E)$ be a properly m-colored directed graph and assume that we are given an oriented list defective coloring instance on G. Assume that for every node $v \in V$, for a sufficiently large constant $\alpha > 0$, it holds that*

$$\sum_{x \in L_v} \big(d_v(x)+1\big)^2 \geq \alpha \cdot \beta_v^2 \cdot \kappa(\beta, \mathcal{C}, m), \tag{3}$$

where $\kappa(\beta,\mathcal{C},m) = (\log\beta + \log\log|\mathcal{C}| + \log\log m)\cdot(\log\log\beta + \log\log m)\cdot \log^2\log\beta$.

Then, there is a deterministic distributed algorithm that solves this oriented list defective coloring instance in $O(\log\beta)$ rounds using $O\big(\min\{|\mathcal{C}|, \Lambda\cdot\log|\mathcal{C}|\} + \log\beta + \log m\big)$-bit messages.

Recursive Color Space Reduction. We have already discussed that we can use list defective colorings to recursively divide the color space. We next elaborate on power of doing recursive color space reduction directly for list defective coloring problems. The following theorem shows that in this way, at the cost of solving a somewhat weaker problem, we can sometimes significantly improve the time complexity or the required message size. In the following, we assume that we have an oriented list defective coloring algorithm $\mathcal{A}$, where the complexity is in particular a function of the maximum list size Λ. More specifically, the following theorem specifies the properties of $\mathcal{A}$ by an arbitrary parameter $\nu \geq 0$ and by arbitrary non-decreasing functions $\kappa(\Lambda)$, $T(\Lambda)$, and $M(\Lambda)$. Note that the functions $\kappa(\Lambda)$, $T(\Lambda)$, and $M(\Lambda)$ can in principle also depend on other global properties such as the maximum degree Δ, the maximum outdegree β, or the number of nodes n. When applying $\mathcal{A}$ to obtain algorithm $\mathcal{A}'$, we then however treat those other parameters as fixed quantities.

▶ **Theorem 3.** *Let $\nu \geq 0$ be a parameter and let $\kappa(\Lambda)$, $T(\Lambda)$, and $M(\Lambda)$ be non-decreasing functions of the maximum list size Λ. Assume that we are given a deterministic distributed algorithm $\mathcal{A}$ that solves oriented list defective coloring instances for which*

$$\forall v \in V\ :\ \sum_{x \in L_v} \big(d_v(x)+1\big)^{1+\nu} \geq \beta_v^{1+\nu} \cdot \kappa(\Lambda).$$

Assume further that the round complexity of $\mathcal{A}$ is $T(\Lambda)$ and that $\mathcal{A}$ requires messages of $M(\Lambda)$ bits.

Then, for any integer $p \in (1, |\mathcal{C}|]$, there exists a deterministic distributed algorithm $\mathcal{A}'$ that solves oriented list defective coloring instances for which

$$\forall v \in V\ :\ \sum_{x \in L_v} \big(d_v(x)+1\big)^{1+\nu} \geq \beta_v^{1+\nu} \cdot \kappa(p)^{\lceil \log_p |\mathcal{C}| \rceil}$$

in time $O(T(p) \cdot \log_p |\mathcal{C}|)$ and with $M(p)$-bit messages.

When replacing β_v by $\deg(v)$, the same theorem also holds for list defective colorings (in undirected graphs). One can easily see this as a list defective coloring can be turned into an equivalent oriented list defective problem, by replacing every edge $\{u, v\}$ of an undirected graph by the two directed edges (u, v) and (v, u).

Note that the number of colors of a standard defective coloring corresponds to the maximum list size Λ of a list defective coloring, i.e., a standard defective coloring with c colors is a special case of list defective coloring with lists of size c. Many of the existing defective and arbdefective coloring algorithms have a round complexity that is of the form $\mathrm{poly}(c) + O(\log^* n)$ [8, 11, 10]. For a concrete application of Theorem 3, we therefore assume that the time complexity of algorithm $\mathcal{A}$ is of the form $T(\Lambda) = \mathrm{poly}(\Lambda) + O(\log^* n)$. For simplicity, we further assume that the size of the color space $\mathcal{C}$ is at most polynomial in β. By setting $p = 2^{O(\sqrt{\log\beta \cdot \log\kappa(\Lambda)})}$, we then get an algorithm that solves oriented list defective coloring problems with $\forall v \in V\ :\ \sum_{x \in L_v}(d_v(x)+1)^{1+\nu} \geq \beta_v^{1+\nu} \cdot 2^{O(\sqrt{\log\beta \cdot \log\kappa(\Lambda)})}$ in time $2^{O(\sqrt{\log\beta \cdot \log\kappa(\Lambda)})} + O(\log^* n)$ rounds. For details, we refer to Corollary 13 in Section 4.

As a second application of Theorem 3, consider the oriented list defective coloring algorithm given by Theorem 2. The round complexity of this algorithm is $O(\log\beta)$ and we cannot hope to get a time improvement by recursively subdividing the color space. We can however improve the necessary message size. The message size of the algorithm is essentially linear in the maximum list size Λ. If we choose $p \ll \Lambda$, the message size becomes essentially linear in p. Assume for example that Λ and the color space are both polynomial in β. We then only need a constant number of recursion levels to reduce the message size to $O(\beta^{\varepsilon} + \log m)$ for any constant $\varepsilon > 0$ (see Corollary 14). We will apply this idea to obtain our new CONGEST algorithm for the $(\Delta + 1)$-coloring problem.

Degree + 1 and List Arbdefective Colorings. The remaining contributions deal with applying (oriented) list defective coloring algorithms to solve the standard $(\mathit{degree}+1)$-coloring problem and more general other coloring problems. The following theorem shows that in combination with (oriented) list defective coloring algorithms, the general technique of [5, 14] cannot only be used to solve standard $(\mathit{degree}+1)$-coloring instances, but more generally also to solve list arbdefective coloring instances for which for all nodes v, $\sum_{x \in L_v}(d_v(x)+1) \geq \deg(v)+1$. Further, if we assume (oriented) list defective coloring algorithms of a certain quality (which is better than what we currently know), we directly obtain algorithms that potentially significantly improve the state of the art for the standard $(\mathit{degree}+1)$-coloring problem. For the following theorem, we assume that for two parameters $\nu > 0$ and

$\kappa > 0$, we have an oriented list defective coloring algorithm $\mathcal{A}^O_{\nu,\kappa}$ or a list defective coloring algorithm $\mathcal{A}^D_{\nu,\kappa}$ to solve instances for which for all v, $\sum_{x\in L_v}(d_v(x)+1)^{1+\nu} \geq \beta_v^{1+\nu}\cdot\kappa$ or $\sum_{x\in L_v}(d_v(x)+1)^{1+\nu} \geq \deg(v)^{1+\nu}\cdot\kappa$. We use $T^O_{\nu,\kappa}$ and $T^D_{\nu,\kappa}$ to denote the time complexities of the two algorithms. Note that the parameter κ can depend (monotonically) on global properties such as the maximum list size Λ, the maximum degree Δ, or the maximum outdegree β. We then however treat those global parameters as fixed quantities when applying the algorithms $\mathcal{A}^O_{\nu,\kappa}$ and $\mathcal{A}^D_{\nu,\kappa}$ recursively.

▶ **Theorem 4.** *Let $\nu \geq 0$ and $\kappa > 0$ be two parameters, let $G = (V, E)$ be an undirected graph with maximum degree Δ, and assume that we are given a list arbdefective coloring instance of G for which $\forall v \in V : \sum_{x\in L_v}(d_v(x)+1) > \deg(v)$. Using the oriented list defective coloring algorithm $\mathcal{A}^O_{\nu,\kappa}$, the given list arbdefective coloring problem can be solved in $O\big(\Lambda^{\frac{\nu}{1+\nu}}\cdot\kappa^{\frac{1}{1+\nu}}\cdot\log(\Delta)\cdot T^O_{\nu,\kappa}+\log^* n\big)$ rounds. Using the list defective coloring algorithm $\mathcal{A}^D_{\nu,\kappa}$, the given list arbdefective coloring problem can be solved in $O\big(\Lambda^{\nu}\cdot\kappa^2\cdot\log(\Delta)\cdot T^D_{\nu,\kappa}+\log^* n\big)$ rounds. If $\nu \geq \nu_0$ for some constant $\nu_0 > 0$, in both time bounds, the $\log(\Delta)$ term can be substituted by $\log(\Delta/\Lambda)$. If $\mathcal{A}^D_{\nu,\kappa}$ (or $\mathcal{A}^O_{\nu,\kappa}$) uses messages of at most B bits, then the resulting list arbdefective coloring algorithm uses messages of $O(B + \log n)$ bits.*

Note that the algorithm of Theorem 2 satisfies the requirements of algorithm $\mathcal{A}^O_{\nu,\kappa}$ for $\nu = 1$. If we assume that we first compute an $O(\Delta^2)$-coloring of G in time $O(\log^* n)$ by using a standard algorithm of [26] and if we assume the size of the color space is at most exponential in Δ, then $\kappa = O(\log\Delta\cdot\log^3\log\Delta)$ and $T^O_{\nu,\kappa} = O(\log\Delta)$. When using the algorithm of Theorem 2 as algorithm $\mathcal{A}^O_{\nu,\kappa}$ in Theorem 4, Theorem 4 therefore implies that the given arbdefective coloring instance can be solved in $O\big(\sqrt{\Lambda}\cdot\log^{3/2}\Delta\cdot\log^{3/2}\log\Delta+\log^* n\big)$ rounds. Hence, the theorem in particular entails that a d-arbdefective $\lfloor\frac{\Delta}{d+1}+1\rfloor$-coloring can be computed in $O(\sqrt{\Delta/(d+1)}\cdot\log^{3/2}\Delta\cdot\log^{3/2}\log\Delta+\log^* n)$ rounds. Even when using $O(\Delta/d)$ colors, the best previous algorithm for this problem required $O(\Delta/d + \log^* n)$ rounds [10]. Theorem 4 further shows that if we could get a fast oriented list defective coloring algorithm for a condition of the form $\sum_{x\in L_v}(d_v(x)+1)^{2-\varepsilon} \geq \beta_v^{2-\varepsilon}\operatorname{poly}\log\Delta$, we would already significantly improve the existing $O(\sqrt{\Delta\log\Delta}+\log^* n)$-round algorithm of [14, 10, 30] to compute a $(\Delta+1)$-coloring. The same would be true in case we could get a fast list defective coloring algorithm for a condition of the form $\sum_{x\in L_v}(d_v(x)+1)^{3/2-\varepsilon} \geq \deg(v)^{3/2-\varepsilon}\cdot\operatorname{poly}\log\Delta$. This indicates that the current obstacle for significantly improving the $O(\sqrt{\Delta\log\Delta}+\log^* n)$-round algorithm (in case this is possible) is to improve our understanding of the complexity of computing defective colorings and possible list defective colorings.

Faster Coloring in the CONGEST Model. Finally, we show that by combining Theorems 2–4, we obtain a faster (*degree* + 1)-list coloring for the CONGEST model.

▶ **Theorem 5.** *Let $G = (V, E)$ be an n-node graph and assume that we are given a (degree+1)-list coloring instance on G. If the color space $\mathcal{C}$ of the problem is of size $|\mathcal{C}| \leq \operatorname{poly}(\Delta)$, there exists a deterministic* CONGEST *algorithm for solving the (degree + 1)-list coloring instance in time $\sqrt{\Delta}\cdot\operatorname{poly}\log\Delta+O(\log^* n)$. If the color space is of size $|\mathcal{C}| = O(\Delta)$ (such as, e.g., for the standard $(\Delta+1)$-coloring problem), the time complexity of the algorithm is $O(\sqrt{\Delta}\cdot\log^2\Delta\cdot\log^6\log\Delta+\log^* n)$.*

Note that there is a $O(\log^2\Delta\cdot\log n)$-round CONGEST algorithm for solving (*degree* + 1)-list coloring instances [18]. Further, by [14, 10, 30] there is an $O(\sqrt{\Delta\log\Delta}+\log^* n)$ algorithm for the problem that uses messages of size $\tilde{O}(\Delta)$. Thus, $(\Delta+1)$-coloring algorithms running in $\sqrt{\Delta}\cdot\operatorname{poly}\log\Delta+O(\log^* n)$ rounds in the CONGEST model are already known as long as $\Delta = O(\log n)$ or $\Delta = \Omega(\log^2 n)$. Our results fill this gap and give such an algorithm for $\Delta \in [\omega(\log n), o(\log^2 n)]$. A rough explanation of why the previous deterministic CONGEST algorithms fail to compute $(\Delta+1)$-colorings efficiently when $\Delta \in [\omega(\log n), o(\log^2 n)]$ is the following. In the algorithm of [14, 10, 30], every node has to *learn* the color lists of its neighbors, which requires that $\Omega(\Delta \cdot \log \Delta)$ bits have to be sent over every edge (which only works in CONGEST if $\Delta = O(\log n)$). For other algorithms (such as for the algorithm of [18]), the round complexity of the algorithms is at least $\Omega(\log n)$, which only leads to efficient time complexities in $\tilde{O}(\sqrt{\Delta})$ if $\Delta = \Omega(\log^2 n)$.

1.2 Organization of the paper

The remainder of the paper is organized as follows. In Section 2, we formally define the communication model and we introduce the necessary mathematical notations and definitions. Section 3 is the main technical section. It discusses our oriented list defective coloring algorithms, leading to the proof of Theorem 2. Subsequently, Section 4 discusses how to improve existing list (defective) coloring algorithms by recursively reducing the color space. Section 5 then shows how (oriented) list defective coloring algorithms can be applied to efficiently solve the standard $(\mathit{degree}+1)$-coloring problem and even list arbdefective coloring problems. The section also shows how this, in combination with the results in Section 3 and Section 4, leads to our new $(\mathit{degree}+1)$-coloring algorithm for the CONGEST model. Due to lack of space, all formal proofs are deferred to the full version of the paper [15].

2 Model and Preliminaries

Communication Model. In the LOCAL model and the CONGEST model [33], the network is abstracted as an n-node graph $G = (V, E)$ in which each node is equipped with a unique $O(\log n)$-bit identifier. Communication happens in synchronous rounds. In every round, every node of G can send a potentially different message to each of its neighbors, receive the messages from the neighbors and perform some arbitrary internal computation. Even if G is a directed graph, we assume that communication can happen in both directions. All nodes start an algorithm at time 0 and the time or round complexity of an algorithm is defined as the total number of rounds needed until all nodes terminate (i.e., output their color in a coloring problem). In the LOCAL model, nodes are allowed to exchange arbitrary messages, whereas in the CONGEST model, messages must consist of at most $O(\log n)$ bits.

Mathematical Notation. Let $G = (V, E)$ be a graph. Throughout the paper, we use $\deg_G(v)$ to denote the degree of a node $v \in V$ in G and $\Delta(G)$ to denote the maximum degree of G. If G is a directed graph, we further use $\beta_{v,G}$ to denote the outdegree of a node $v \in V$. More specifically, for convenience, we define $\beta_{v,G}$ as the maximum of 1 and the outdegree of v, i.e., we also set $\beta_{v,G} = 1$ if the outdegree of v is 0. The maximum outdegree β_G of G is defined as $\beta_G := \max_{v \in V} \beta_{v,G}$. We further use $N_G(v)$ to denote the set of neighbors of a node v and if G is a directed graph, we use $N_G^{out}(v)$ to denote the set of outneighbors of v. In all cases, if G is clear from the context, we omit the subscript G. When discussing one of the list defective coloring problems on a graph $G = (V, E)$, we will typically assume that $\mathcal{C}$ denotes the space of possible colors, and we use L_v and d_v for $v \in V$ to denote the color list and defect function of node v. Throughout the paper, we will w.l.o.g. assume that $\mathcal{C} \subseteq \mathbb{N}$ is a subset of the natural numbers. When clear from the context, we do not explicitly introduce this notation each time. Further, for convenience, for an integer $k \geq 1$, we use $[k] := \{1, \ldots, k\}$ to denote the set of integers from 1 to k. Further, for a finite set A and an integer $k \geq 0$, we use 2^A to denote the power set of A and $\binom{A}{k}$ to denote the set of subsets of size k of A. Finally, we use $\log(x) := \log_2(x)$ and $\ln(x) := \log_e(x)$.

3 Distributed Oriented List Defective Coloring Algorithms

3.1 Fundamentals

Our algorithm in based on the list coloring approach of Maus and Tonoyan [30] that we sketch next. As input, each node of $G = (V, E)$ obtains a color list $L_v \subseteq \mathcal{C}$ of size $|L_v| \geq \alpha\beta^2\tau$ for a sufficiently large constant $\alpha > 0$ and some integer parameter $\tau > 0$. In addition, the nodes are equipped with an initial proper m-coloring of G. The "highlevel" idea is based on the classic one-round $O(\beta^2 \log m)$-coloring algorithm of Linial [26]. As an intermediate step of the algorithm of [30], every node v chooses a subset $C_v \subseteq L_v$ of size $|C_v| = \beta\tau$ of its list such that for every outneighbor u of v, it holds that $|C_u \cap C_v| < \tau$. To obtain a proper coloring of G, node v can then choose a color $x \in C_v$ that does not appear in any of the sets C_u of one of the $\leq \beta$ outneighbors u of v. If all nodes have to pick a color from $\{1, \dots, \alpha\beta^2\tau\}$ and if $\tau = O(\log\beta + \log m)$ is chosen sufficiently large, Linial shows that such sets C_v for all nodes v can be computed in 0 rounds without communication. However, this is not true for the list coloring variant of the problem considered in [30].

Before we show how the authors of [30] overcome the problems of list, we introduce some terminology. Let P_0 be the original list coloring problem that we need to solve and let P_1 be the intermediate problem of choosing a set C_v from $\binom{L_v}{\beta\cdot\tau}$ s.t. $|C_v \cap C_u| < \tau$ for all outneighbors u of v. As discussed, after solving P_1, P_0 can be solved in a single round, each node v just needs to learn the sets C_u of all its outneighbors u. To solve P_1, the authors of [30] introduce a new problem P_2, that can be seen as a "higher-dimensional" variant of Linial's algorithm. P_2 is defined in such a way that it can be solved without communication in 0 rounds and such that after solving P_2, P_1 can be solved in a single round. In problem P_2, every node v computes a set of possible candidates for the set C_v. For a more detailed description we need to define the following conflict relation.

▶ **Definition 6** (Conflict relation $\Psi(\tau', \tau)$). *Let $\tau', \tau > 0$ be two parameters. The relation $\Psi(\tau', \tau) \subseteq 2^{2^{\mathcal{C}}} \times 2^{2^{\mathcal{C}}}$ is defined as follows. For any $K_1, K_2 \in 2^{2^{\mathcal{C}}}$, we have*

$$(K_1, K_2) \in \Psi(\tau', \tau) \Leftrightarrow \exists \text{ distinct } C_1, \dots, C_{\tau'} \in K_1 \text{ s.t.}$$
$$\forall i \in \{1, \dots, \tau'\}\ \exists C \in K_2 \text{ for which } |C_i \cap C| \geq \tau.$$

In a solution to problem P_2, every node v outputs a set $K_v \subseteq 2^{\binom{L_v}{\beta\tau}}$ such that $|K_v| = \beta\tau'$ (for some integer parameter $\tau' > 0$) and such that for every outneighbor u of v, $(K_v, K_u) \notin \Psi(\tau', \tau)$. Note that this implies that for every outneighbor u, K_v contains at most $\tau' - 1$ sets C for which there is a set $C' \in K_u$ for which $|C \cap C'| \geq \tau$. Because K_v has size $\beta\tau'$, there exists some $C_v \in K_v$ such that for every $C' \in K_u$ for every outneighbor u, we have $|C_v \cap C'| < \tau$. A solution of P_2 can be transformed into a solution of P_1 in a single round (each node v has to communicate it's set K_v to all its outneighbors u). Maus and Tonoyan [30] showed that for appropriate choices of the parameters τ and τ', P_2 can be solved in 0 rounds. To see this, consider the following technical lemma, which follows almost directly from Lemmas 3.3 and 3.4 in [30].

▶ **Lemma 7** (adapted from [30]). *Let $\gamma, \tau, \tau' \geq 1$ be three integer parameters such that $\tau \geq 8\log\gamma + 2\log\log|\mathcal{C}| + 2\log\log m + 16$ and $\tau' = 2^{\tau - \lceil\log(2e\gamma^2)\rceil}$. For every color list $L \in \binom{\mathcal{C}}{\ell}$ of size ℓ for some $\ell \geq 2e\gamma^2\tau$, we further define $S(L) := \binom{\binom{L}{\gamma\tau}}{\gamma\tau'}$. Then, there exists $\bar{S}(L) \subseteq S(L)$ such that $|\bar{S}(L)| \geq |S(L)|/2$ and such that for every $K \in \bar{S}(L)$ and every $L' \in \binom{\mathcal{C}}{\ell}$, there are at most $d_2 < \frac{1}{4m|\mathcal{C}|^\ell} \cdot |S(L)|$ different $K' \in S(L')$ such that $(K, K') \in \Psi(\tau', \tau)$ or $(K', K) \in \Psi(\tau', \tau)$. Further, for every $K \in S(L)$ and every $L' \in \binom{\mathcal{C}}{\ell}$, there are at most d_2 different $K' \in S(L')$ for which $(K, K') \in \Psi(\tau', \tau)$.*

Let us sketch how Lemma 7 implies that for appropriate choices of the parameters, P_2 can be solved without communication. In the following, we set the parameters of Lemma 7 as $\gamma := \beta$ and τ and τ' as given by the lemma statement. Assume that initially, every node v has a list L_v of size ℓ, where $\ell \geq 2e\beta^2\tau$. We define the type T_v of a node as the tuple (c, L_v), where c is the color of v in the initial proper m-coloring of G and $L_v \in \binom{\mathcal{C}}{\ell}$ is the color list of v. Let $T_1, \ldots, T_t$ be a fixed ordering of the $t = m\binom{|\mathcal{C}|}{\ell} \leq m|\mathcal{C}|^\ell$ types and let L_i be the color list of nodes of type T_i. If we assign a set $K_i \in S(L)$ to each type T_i so that for any two sets K_i and K_j, $(K_i, K_j) \notin \Psi(\tau', \tau)$ and $(K_j, K_i) \notin \Psi(\tau', \tau)$, then if each node v of type T_i (for $i \in \{1, \ldots, t\}$) outputs K_i, this assignment solves problem P_2. We assign the sets K_i greedily, where for every type T_i, we choose some $K_i \in \bar{S}(L_i)$ such that $\bar{S}(L_i)$ is the subset of $S(L_i)$ that is guaranteed to exist by Lemma 7. Assume for any $i \geq 1$ each type T_j for $j \in \{1, \ldots, i-1\}$ already picked K_j. Then type T_i will pick some list $K_i \in \bar{S}(L)$ that does not conflict with choices of the $i-1$ previous types. By the lemma, for any type T_j, $j \neq i$, there are at most $d_2 < \frac{1}{4m|\mathcal{C}|^\ell} \cdot |S(L_i)| \leq \frac{1}{2t} \cdot |\bar{S}(L_i)|$ sets in $\bar{S}(L_i)$ that conflict. Because there are only t types, we can always choose an appropriate K_i that does not conflict with already assigned sets K_j for $j < i$. Consequently, P_2 can be solved in 0 rounds, and thus the original list coloring problem can be solved in 2 rounds.

3.2 Basic Oriented List Defective Coloring Algorithm

In the following, we first give a basic algorithm that solves a slightly generalized version of the OLDC problem. Concretely, the algorithm assigns a color $x_v \in L_v$ with defect $d_v(x_v)$ to each node v s.t. at most $d_v(x_v)$ outneighbors w of v choose a color x_w with $|x_v - x_w| \leq g$, where $g \geq 0$ is some given parameter. Recall that we assume that all colors are integers and therefore, the value $|x_v - x_w|$ is defined. Note that for $g = 0$, this is the OLDC problem as defined in Definition 1. We give a basic algorithm for this more general problem because we will need it as a subroutine in the algorithm for proving our main technical result, Theorem 2. The steps for solving the generalized OLDC problem are similar to the approach described in Section 3.1. We however in particular have to adapt the algorithm to handle the case where each node comes with an individual list size.

A single defect per node. At the core of our basic (generalized) OLDC algorithm is an algorithm that solves the following weaker variant of the problem. Instead of having color-specific defects, every node v has a fixed defect value $d_v \geq 0$, i.e., we have $d_v(x) = d_v$ for all $x \in L_v$. Based on an algorithm for this *single-defect* case, one can solve the general OLDC problem by using a reduction explained in the proof of Lemma 12. For that reason, assume during the following section that each node v has three given inputs, a color list L_v, a defect value $d_v \geq 0$ and the number of outneighbors β_v. For each node v, the algorithm then requires lists of size $|L_v| \geq \alpha(\beta_v/(d_v+1))^2 \cdot \tau$ for some constant $\alpha > 0$ and some parameter $\tau > 0$. Note that the required list size only depends on the ratio between β_v and $d_v + 1$ and not on their actual values. In the following section we therefore do not work with the defect value d_v or the outdegree β_v of some node, but with a value γ_v that is essentially equal to the ratio $\beta_v/(d_v+1)$ such that the list size of v is proportional to γ_v^2. More formally, we partition the nodes into so-called γ-classes such that nodes in the same class have the same value γ_v and hence a similar $\beta_v/(d_v+1)$ ratio. The details appear in the subsequent section.

3.2.1 γ-Classes and Parameters

Each node v comes with some parameter $\gamma_v = 2^i$ for some $i \in [h]$, where h is a parameter. We call i the γ-class of v. Since these γ-classes have a natural order, we call a node u to be in a *lower* (respectively *higher*) γ-class than v if $i_u < i_v$ (respectively $i_u > i_v$). We define the following two parameters, which both depend on the maximum γ-class index h, the color space $\mathcal{C}$, and the initial (proper) m-coloring of G.

$$\tau(h, \mathcal{C}, m) := \lceil 8h + 2 \log\log |\mathcal{C}| + 2 \log\log m + 16 \rceil, \tag{4}$$
$$\tau'(h, \mathcal{C}, m) := 2^{\tau(h,\mathcal{C},m) - \lceil 2h + \log(2e) \rceil}. \tag{5}$$

Note that these choices are consistent with the parameter setting in Lemma 7. If clear from the context, we omit the parameters $h, \mathcal{C}$ and m for simplicity and denote $\tau(h, \mathcal{C}, m)$ by τ and $\tau'(h, \mathcal{C}, m)$ by τ'. A node v of γ-class i_v is equipped with a color list L_v of size $|L_v| = \ell_{i_v} := \alpha \cdot 4^{i_v} \tau(2g + 1) = \alpha\gamma_v^2 \tau(2g + 1)$ for some sufficiently large $\alpha > 0$ and $g > 0$. Because we solve a generalized version of the OLDC problem, we have to use a more general form for our conflict relation Ψ. Let $x \in \mathcal{C}$ be a color and $C \subseteq \mathcal{C}$ a set of colors, we denote the number of conflicts of x with colors in C regarding some given $g \geq 0$ by $\mu_g(x, C) := |\{c \in C \mid |x - c| \leq g \}|$.

▶ **Definition 8** (τ&g-conflict). *Two lists $C, C' \subseteq \mathcal{C}$ do τ&g-conflict if $\sum_{x \in C} \mu_g(x, C') \geq \tau$.*

Note that $\sum_{x \in C} \mu_g(x, C') = \sum_{x \in C'} \mu_g(x, C)$ is always true. The Ψ conflict relation from Definition 6 is adapted accordingly.

▶ **Definition 9** (Conflict relation $\Psi_g(\tau', \tau)$). *Let $\tau', \tau > 0$ be two parameters. The relation $\Psi_g(\tau', \tau) \subseteq 2^{2^{\mathcal{C}}} \times 2^{2^{\mathcal{C}}}$ is defined as follows. For any $K_1, K_2 \in 2^{2^{\mathcal{C}}}$, we have*

$$(K_1, K_2) \in \Psi_g(\tau', \tau) \Leftrightarrow \exists \text{ distinct } C_1, \ldots, C_{\tau'} \in K_1 \text{ s.t.}$$
$$\forall i \in \{1, \ldots, \tau'\} \; \exists C \in K_2 \text{ for which } C_i \text{ and } C \text{ do } \tau\&g\text{-conflict.}$$

We adapt the definitions of problem P_1 and P_2 to what we need for the generalized OLDC problem. We define $k_i := 2^i \cdot \tau$ and $k' := 2^h \cdot \tau'$. Subsequently, we denote the γ-class of a node v by i_v.

▶ **Definition 10** (Problems P_1 and P_2).

P_1: *Every node v has to output $C_v \subseteq L_v$ of size $|C_v| = k_{i_v}$ s.t. there are at most $d_v/2$ outneighbors u of v s.t. u is in γ-class $i_u \leq i_v$, C_u and C_v do τ&g-conflict.*

P_2: *Every node v has to output a list $K_v \in 2^{\binom{L_v}{k_{i_v}}}$ of size $|K_v| = k'$ s.t. for each outneighbor u in γ-class $i_u \leq i_v$, $(K_v, K_u) \notin \Psi_g(\tau', \tau)$.*

The next lemma states that P_2 can be solved in zero rounds. The high-level idea is to first adapt Lemma 7 such that we can apply it even if the size of the initial color lists differs. We start by using a simple trick to make sure that the color list L_v of each node v does not contain colors that are *close* to each other i.e., there are no distinct colors $x_1, x_2 \in L_v$ s.t. $|x_1 - x_2| \leq g$. After doing this, the conflict relation Ψ_g behaves almost the same as Ψ behaves in the fundamental problem. For details, we refer to the full version of the paper [15].

▶ **Lemma 11.** *P_2 can be solved without communication given an initial m-coloring.*

3.2.2 Algorithm

Assume each node is equipped with a color list of size $|L_v| \geq \alpha\left(\beta_v/(d_v+1)\right)^2 \tau(2g+1)$ and some defect value $d_v > 0$. The γ-class of a node v is defined as the smallest i_v s.t. $2^{i_v} \geq \frac{2\beta_v}{d_v+1}$. Based on the individual γ-class, each node solves P_2 (Lemma 11) and forwards the solution to the neighbors. The knowledge gained by that is used to solve P_1 without additional communication. In more detail, node v comes with list K_v s.t. $(K_v, K_u) \notin \Psi_g(\tau', \tau)$ for any outneighbor u with $i_u \leq i_v$. This implies that at most $\tau' - 1$ lists $C \in K_v$ do $\tau\&g$-conflict with some list in K_u. Hence, there are at most $\beta_v(\tau'-1)$ many $C \in K_v$ that $\tau\&g$-conflict. By the pigeonhole principle there is some $C_v \in K_v$ with at most[2] $\beta_v(\tau'-1)/k' < (d_v+1)/2$ many such conflicts within all the C_u of outneighbors u of smaller γ-classes. Hence, C_v is a valid solution for P_1 that is then forwarded to the neighbors.

To solve the list coloring problem itself we iterate through the γ-classes in descending order. Each node v has to decide on a color $x \in C_v$ s.t. in the end at most d_v outneighbors are colored with the same color x. Let us fix a node v. By design, in the iteration v decides on a color, all outneighbors of higher γ-classes already decided on a color and v knows the P_1 solution lists C_u of the outneighbors u with $i_u \leq i_v$. Let $f_v(x)$ be the frequency of color x within the outneighbors u of v i.e., the sum over all occurrences of x in C_u's of neighbors with the same or smaller γ-class plus the number of outneighbors of higher γ-classes that are already colored with x. The color x with the lowest frequency in C_v will be the final color of v for the following reason: There are at most $d_v/2$ outneighbors that have an unbounded number of $\tau\&g$-conflicts, while C_v shares at most $\tau - 1$ colors with the remaining C_u's (note that with outneighbors of higher γ-class at most one color in C_v is in conflict, hence, for worst-case observation we can ignore that case). By the pigeonhole principle there exist a color $x \in C_v$ with

$$f_v(x) \leq \frac{\sum_{c \in C_v} f_v(c)}{|C_v|} \leq \frac{d_v/2 \cdot |C_v| + \beta_v(\tau-1)}{|C_v|} < d_v + 1.$$

The round complexity of the whole algorithm is $O(h)$, since we iterate through all the γ-classes to assign colors. This completes the algorithm to handle *single* defects. We will now extend this result to solve the OLDC problem.

3.2.3 Multiple Defects

In the general case, each node can have lists that are composed of colors of different defects. In order to reduce the general case to the case, where each node only has a single defect, every node v first rounds all its defects to the next smaller power of 2. Every node v then can have $h = O(\log \beta)$ different defect values. The node v only keeps the colors in its list for one of those defect values. This value is chosen such that the sum $\sum_{x \in L'_v}(d_v(x)+1)^2$ is maximized, where L'_v is the reduced list of v. Note that when doing this, the sum $\sum_{x \in L_v}(d_v(x)+1)^2$ for the original list L_v is at most by an $O(h) = O(\log \beta)$ factor larger.

▶ **Lemma 12.** *Given a graph with an initial m-coloring. There is an $O(h)$-round* LOCAL *algorithm that assigns each node v a color $x_v \in L_v$ such that every node $v \in V$ has at most $d_v(x_v)$ outneighbors w with a color x_w for which $|x_w - x_v| \leq g$ if for each node $v \in V$*

$$\sum_{x \in L_v} (d_v(x)+1)^2 \geq \alpha\beta_v^2 \cdot \tau(h, \mathcal{C}, m) \cdot h \cdot (2g+1)$$

for some sufficiently large constant α, some integer $h \geq \max_v \lceil \log(\frac{\beta_v}{\min_{x \in L_v} d_v(x)+1}) \rceil$ and color space $\mathcal{C}$. Messages are of size at most $O(\min\{\Lambda \cdot \log|\mathcal{C}|, |\mathcal{C}|\} + \log\log\beta + \log m)$-bits.

[2] By definition of the γ-class, we have $(d_v+1)/2 \geq \beta_v/2^h$

To apply this lemma one can use that $h = O(\log \beta)$ and for some color space of size poly Δ and initial $O(\Delta^2)$-coloring (e.g., by [26]) we have $\tau(h, \mathcal{C}, m) = O(\log \Delta)$. Hence, Lemma 12 solves the OLDC problem (where $g = 0$) in $O(\log \Delta)$ communication rounds if the initial color list L_v for each node v fulfills the condition $\sum_{x \in L_v} (d_v(x) + 1)^2 \geq \alpha \beta_v^2 \cdot \log^2 \Delta$ for some sufficient large constant α. Note that the OLDC algorithm mentioned in Theorem 2 requires a stronger condition on the color lists L_v than Lemma 12. However, to reach this better result, we apply Lemma 12 as subroutine. The details of this more involved analysis are stated in Appendix A.

4 Recursive Color Space Reduction

Distributed list defective coloring algorithms first implicitly appeared in [24] as a tool to recursively reduce the color space of a distributed coloring problem. In this section, we show that the idea of using list defective colorings to recursively reduce the color space can also directly be applied to the (oriented) list defective coloring problem. In this way, at the cost of requiring slightly larger lists, we can turn a given distributed (oriented) list defective coloring algorithm into another distributed (oriented) list defective coloring algorithm that is faster and/or needs smaller messages. The high-level idea is the following. Assume that we are given an (oriented) list defective coloring problem with colors from a color space $\mathcal{C}$. We can arbitrarily partition $\mathcal{C} = \mathcal{C}_1 \cup \dots \cup \mathcal{C}_p$ into p approximately equal parts. Instead of directly choosing a color, each node v now first just selects the color subspace $\mathcal{C}_i$ from which v chooses its color. If v starts with color list L_v, then after choosing the color subspace $\mathcal{C}_i$, v's color list reduces to $L_{v,i} = L_v \cap \mathcal{C}_i$ (with the original defects on those colors). However, v now only has to compete with neighbors that also pick the same color subspace $\mathcal{C}_i$. The choices of color subspaces by the nodes can itself be phrased as an (oriented) list defective coloring instance for a color space of size p and thus also with lists of size at most p. Theorem 3 in Section 1.1 formalizes this idea.

It has been well-known since Linial's seminal work in [26] that in directed graphs of outdegree at most β, one can compute a proper $O(\beta^2)$-coloring in $O(\log^* n)$ rounds (or in $O(\log^* m)$ rounds if an initial proper m-coloring is provided). In [23], it was shown that in the same way, one can also compute an oriented d-defective coloring with $O((\beta/d)^2)$ colors. In [30], the coloring result of [26] was extended to the list coloring problem and in Section 3 of this paper (and to a limited extent also in [30]), the defective coloring result of [23] is extended to the oriented list defective coloring problem. While there has been progress on solving the natural list and defective coloring variants of $O(\beta^2)$ coloring, it is still unknown if a coloring with $O(\beta^{2-\varepsilon})$ colors (for some constant $\varepsilon > 0$) can be computed in time $f(\beta) + O(\log^* n)$.[3] Even if a moderately fast distributed algorithm for better oriented list defective colorings exists, we directly also get much faster algorithms for computing proper colorings with $o(\beta^2)$ colors. In the following, we assume that there exists an oriented defective coloring algorithm with a round complexity that is polynomial in the number of colors per node plus $O(\log^* n)$. Such algorithm for example exist for (list) defective colorings in graphs of neighborhood independence at most Δ^ε [9, 24].

[3] Note that oriented graphs with maximum outdegree β have colorings with $O(\beta)$ colors. However, the best distributed algorithm to compute an $O(\beta)$-coloring requires time $O(\log^3 \beta \cdot \log n)$ [18]. It is not known if colorings with $O(\beta^{2-\varepsilon})$ colors can be computed with an n-dependency $o(\log n)$.

▶ **Corollary 13.** *Let $\nu \geq 0$ be a parameter and let $\kappa(\Lambda)$ be a non-decreasing functions of the maximum list size Λ. Assume that we are given a deterministic distributed algorithm $\mathcal{A}$ that solves oriented list defective coloring instances for which*

$$\forall v \in V \;:\; \sum_{x \in L_v} \big(d_v(x)+1\big)^{1+\nu} \geq \beta_v^{1+\nu} \cdot \kappa(\Lambda).$$

Assume further that if an initial proper m-coloring is given, $\mathcal{A}$ has a round complexity of $\mathrm{poly}(\Lambda)+O(\log^* m)$*. Then, there exists a* $\big(2^{O(\sqrt{\log\beta\log\kappa(\Lambda)})}+O(\log^* m)\big)$*-round deterministic distributed list coloring algorithm $\mathcal{A}'$ to solve list coloring instances with colors from a color space of size* $\mathrm{poly}(\beta)$ *for which*

$$\forall v \in V \;:\; \sum_{x \in L_v} (d_v(x)+1)^{1+\nu} \geq \beta_v^{1+\nu} \cdot 2^{O(\sqrt{\log\beta\cdot\log\kappa(\Lambda)})}.$$

Note that the $2^{O(\sqrt{\log\Delta})}+O(\log^* n)$-round algorithm for computing a $(\Delta+1)$-coloring in graphs of bounded neighborhood independence and thus in particular in line graphs of bounded rank hypergraphs is based on the same idea as Corollary 13. The corollary shows how in some cases, recursive color space reduction can be used to significantly speed up a given (oriented) list defective coloring problem. The following corollary shows that recursive color space reduction can sometimes also be used to significantly reduce the required message size of an (oriented) list defective coloring algorithm. In the oriented list defective coloring algorithm of Section 3, all nodes need to learn the lists and defect vectors of their neighbors and this dominates the required communication. A list L_v of length $|L_v| \leq \Lambda$ consisting of colors from a color space of size $|\mathcal{C}|$ can be represented by $\min\{|\mathcal{C}|, \Lambda\log|\mathcal{C}|\}$ bits and a corresponding defect vector can be represented by $\Lambda\log\beta$ bits, or even by $\Lambda\log\log\beta$ bits if we assume that all defects are integer powers of 2 (which can usually be assumed at the cost of a factor 2 in the required list size). In the following, we assume that we have an algorithm that requires $O(|\mathcal{C}|\cdot B+\log n)$ bits for some parameter $B \geq 1$ (the $\log n$ is included to cover things like exchanging initial colors, unique IDs, etc.).

▶ **Corollary 14.** *Let $\nu \geq 0$ be a parameter and let $\kappa(\Lambda)$ be a non-decreasing functions of the maximum list size Λ. Assume that we are given a deterministic distributed algorithm $\mathcal{A}$ that solves oriented list defective coloring instances for which*

$$\forall v \in V \;:\; \sum_{x \in L_v} \big(d_v(x)+1\big)^{1+\nu} \geq \beta_v^{1+\nu} \cdot \kappa(\Lambda).$$

Assume further that $\mathcal{A}$ has a round complexity of $T(\Lambda)$ and requires messages of $O(|\mathcal{C}|\cdot B+\log n)$ bits, where $\mathcal{C}$ the color space and $B \geq 1$ is some parameter. Then, for every integer $r \geq 1$, there exists an $O(T(\Lambda)\cdot r)$-round deterministic distributed list coloring algorithm $\mathcal{A}'$ to solve list coloring instances with colors from the same color space and for which

$$\forall v \in V \;:\; \sum_{x \in L_v} (d_v(x)+1)^{1+\nu} = \beta_v^{1+\nu} \cdot \kappa(\Lambda)^r.$$

The algorithm $\mathcal{A}'$ requires messages of size $O(|\mathcal{C}|^{1/r}\cdot B+\log n)$.

As for Theorem 3, when replacing β_v by $\deg(v)$, Corollary 13 and Corollary 14 both also hold for the list defective coloring problem in undirected graphs.

5 Applying List Defective Colorings

In [5] and [14], Barenboim, and Fraigniaud, Heinrich, and Kosowski developed a technique to transform fast, but relaxed (oriented) list coloring into efficient algorithms for the (*degree*$+1$)-list coloring problem. The same technique has later also been used by the algorithms in [24, 4, 3]. The high-level idea of this transformation is as follows. Assume that for some $\alpha > 1$, we have a T-round algorithm $\mathcal{A}$ that solves list coloring instances with lists of size $> \alpha\Delta$ in graphs of maximum degree Δ. We can then first use a defective k-coloring to decompose the graph into k subgraphs of maximum degree $\leq \Delta/(2\alpha)$. One then iterates over those color classes and extends a given partial (*degree* $+ 1$)-list coloring. When working on the nodes of some color class, all nodes that still have at least $\Delta/2$ uncolored neighbors also still have a list of size $> \Delta/2$. This is more than α times the maximum degree $\Delta/(2\alpha)$ in the current color class, and we can therefore color such nodes by using algorithm $\mathcal{A}$. In $k \cdot T$ rounds, we can therefore reduce the maximum degree of our (*degree* $+ 1$)-list coloring problem from Δ to $\Delta/2$ and by repeating $O(\log \Delta)$ times, we can solve the (*degree* $+ 1$)-list coloring problem. If the algorithm $\mathcal{A}$ works on directed graphs of maximum outdegree β and requires lists of size $> \alpha\beta$, the same idea also works if we decompose the graph by using an arbdefective coloring instead of a defective coloring.

The contribution of this section is two-fold. Firstly, we show that if we assume the existence of (oriented) list coloring algorithms that are significantly better than the current state of the art, we would directly obtain significantly faster algorithms for the standard $(\Delta + 1)$-coloring problem. Moreover, we show that by replacing the algorithm $\mathcal{A}$ in the description above by an (oriented) list defective coloring algorithm, the technique cannot only be used for the (*degree*$+1$)-list coloring problem, but it also works for computing arbdefective colorings and more generally list arbdefective colorings. In fact, it works for list arbdefective colorings with lists L_v and defects d_v such that for all $v \in V$, $\sum_{v\in V}(d_v(x) + 1) > \deg(v)$. In the following, we refer to such instances as (*degree*$+1$)-list arbdefective coloring instances. We subsequently assume that $\mathcal{A}^D_{\nu,\kappa}$ is a deterministic distributed list defective coloring algorithm that operates on undirected graphs and $\mathcal{A}^O_{\nu,\kappa}$ is a deterministic distributed oriented list defective coloring algorithm that operates on directed graphs. For real values $\nu \geq 0$ and $\kappa > 0$ we assume that $\mathcal{A}^D_{\nu,\kappa}$ and $\mathcal{A}^O_{\nu,\kappa}$ solve all (oriented) list defective coloring problems for which for all $v \in V$,

$$\sum_{x\in L_v} \big(d_v(x) + 1\big)^{1+\nu} \geq \deg(v)^{1+\nu} \cdot \kappa \text{ and} \tag{6}$$

$$\sum_{x\in L_v} \big(d_v(x) + 1\big)^{1+\nu} \geq \beta_v^{1+\nu} \cdot \kappa, \tag{7}$$

respectively. We assume that the round complexity of algorithm $\mathcal{A}^D_{\nu,\kappa}$ is $T^D_{\nu,\kappa}$ and that the round complexity of algorithm $\mathcal{A}^O_{\nu,\kappa}$ is $T^O_{\nu,\kappa}$. Theorem 4 in Section 1.1 shows that by using $\mathcal{A}^O_{\nu,\kappa}$, one can solve (*degree* $+ 1$)-list arbdefective coloring instances in time $O\big(\Lambda^{\frac{\nu}{1+\nu}} \cdot \kappa^{\frac{1}{1+\nu}} \cdot \log(\Delta) \cdot T^O_{\nu,\kappa} + \log^* n\big)$ and by using $\mathcal{A}^D_{\nu,\kappa}$ one can solve such list arbdefective colorings in time $O\big(\Lambda^{\nu} \cdot \kappa^2 \cdot \log(\Delta) \cdot T^O_{\nu,\kappa} + \log^* n\big)$.

Implications of Theorem 4

We first discuss two immediate implications of Theorem 4, and we afterwards show how the theorem can be used to improve the best current deterministic complexity of the $(\Delta + 1)$-coloring problem in the CONGEST model.

Complexity of Computing (List) Arbdefective Colorings. For a first immediate implication of Theorem 4, we can use the algorithm of Theorem 2 as the oriented list defective coloring algorithm $\mathcal{A}^{O}_{\nu,\kappa}$. If we assume that the color space that we have is of size $|\mathcal{C}| = \text{poly}(\beta)$, in this case, $\nu = 1$ and $\kappa = O(\log\beta \cdot \log^3 \log\beta)$. This results in an arbdefective coloring algorithm that solves instances with lists L_v for which $\forall v \in V : \sum_{x\in L_v}(d_v(x)+1) > \deg(v)$ in time $O\big(\sqrt{\Lambda} \cdot \log^{5/2}\Delta \cdot \log^{3/2}\log\Delta + \log^* n\big)$. In particular, this implies that for any $d \geq 0$ and any $q > \frac{\Delta}{d+1}$, a d-arbdefective q-coloring can be computed in time $O\Big(\sqrt{\frac{\Delta}{d+1}} \cdot \log^{5/2}\Delta \cdot \log^{3/2}\log\Delta + \log^* n\Big)$, which significantly improves the previously best algorithms that achieves the same arbdefective coloring in time $O(\Delta + \log^* n)$ [2] or a more relaxed d-arbdefective $O\big(\frac{\Delta}{d+1}\big)$-coloring in time $O\big(\frac{\Delta}{d+1} + \log^* n\big)$. Note also that the condition $\forall v \in V : \sum_{x\in L_v}(d_v(x)+1) > \deg(v)$ is necessary in order to compute a (list) arbdefective coloring in time $f(\Delta) + O(\log^* n)$. If the condition does not hold, any deterministic algorithm for the problem requires at least $\Omega(\log_\Delta n)$ rounds [2].

Better List Defective Coloring Implies Better $(\Delta+1)$-Coloring. The theorem in particular also implies that certain progress on (oriented) list defective coloring algorithms would directly lead to faster algorithms for the standard $(\Delta+1)$-coloring problem. Assume that for an initial m-coloring of the graph, we have an oriented list defective coloring algorithm with a round complexity that is $\text{poly}(\Lambda) + O(\log^* m)$ and that satisfies equation (6) for any constant $\nu < 1$. In combination with Corollary 13, Theorem 4 then implies that we then obtain a $(\mathit{degree}+1)$-list coloring (and thus $(\Delta+1)$-coloring) algorithm with a time complexity of $O\big(\Delta^{\frac{\nu}{1+\nu}+o(1)} + \log^* n\big)$, which would be polynomial improvement over the $O(\sqrt{\Delta\log\Delta} + \log^* n)$-round algorithm of [14, 10, 30]. The same would be true if we had a list defective coloring algorithm with a round complexity of $\text{poly}(\Lambda) + O(\log^* m)$ and that satisfies equation (7) for any constant $\nu < 1/2$. We believe that if it is possible to significantly improve the current best $O(\sqrt{\Delta\log\Delta} + \log^* n)$-round of $(\Delta+1)$-coloring, the key will be to better understand the distributed complexity of (oriented) defective colorings and probably also of the more general (oriented) list defective colorings.

Complexity of $(\Delta+1)$-Coloring in the CONGEST Model. Apart from the standard $(\Delta+1)$-coloring problem, in the following, we also consider the general $(\mathit{degree}+1)$-list coloring problem. In order to keep the results simple and because this is the most interesting case, we will assume that we have $(\mathit{degree}+1)$-list coloring instances with a color space of size at most $\text{poly}(\Delta)$. Note that in the case of the standard $(\Delta+1)$-coloring problem, the color space is of size $\Delta+1$. For small Δ, the best $(\Delta+1)$-coloring algorithm in the LOCAL model has a round complexity of $O(\sqrt{\Delta\log\Delta} + \log^* n)$ [14, 10, 30] and Theorem 5 in Section 1.1 states that this round complexity can almost be matched in the CONGEST model.

References

1 Baruch Awerbuch, Andrew V. Goldberg, Michael Luby, and Serge A. Plotkin. Network decomposition and locality in distributed computation. In *Proc. 30th Symp. on Foundations of Computer Science (FOCS)*, pages 364–369, 1989. doi:10.1109/SFCS.1989.63504.

2 Alkida Balliu, Sebastian Brandt, Fabian Kuhn, and Dennis Olivetti. Distributed Δ-coloring plays hide-and-seek. *arXiv preprint arXiv:2110.00643*, 2021.

3 Alkida Balliu, Sebastian Brandt, Fabian Kuhn, and Dennis Olivetti. Distributed edge coloring in time polylogarithmic in Δ. In *Proc. 41st ACM Symp. on Principles of Distributed Computing (PODC)*, pages 15–25, 2022.

4 Alkida Balliu, Fabian Kuhn, and Dennis Olivetti. Distributed edge coloring in time quasi-polylogarithmic in delta. In *Proc. 39th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 289–298, 2020.

5 Leonid Barenboim. Deterministic (δ+1)-coloring in sublinear (in δ) time in static, dynamic, and faulty networks. *Journal of ACM*, 63(5):1–22, 2016. doi:10.1145/2979675.

6 Leonid Barenboim and Michael Elkin. Distributed (delta+1)-coloring in linear (in delta) time. In *Proc. 41st Annual ACM Symp. on Theory of Computing (STOC)*, pages 111–120, 2009.

7 Leonid Barenboim and Michael Elkin. Sublogarithmic distributed MIS algorithm for sparse graphs using Nash-Williams decomposition. *Distributed Comput.*, 22:363–379, 2010. doi:10.1007/s00446-009-0088-2.

8 Leonid Barenboim and Michael Elkin. Deterministic distributed vertex coloring in polylogarithmic time. *Journal of ACM*, 58:23:1–23:25, 2011. doi:10.1145/2027216.2027221.

9 Leonid Barenboim and Michael Elkin. Distributed deterministic edge coloring using bounded neighborhood independence. In *Proc. 30th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 129–138, 2011.

10 Leonid Barenboim, Michael Elkin, and Uri Goldenberg. Locally-iterative distributed $(\Delta + 1)$-coloring below Szegedy-Vishwanathan barrier, and applications to self-stabilization and to restricted-bandwidth models. In *Proc. 37th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 437–446, 2018.

11 Leonid Barenboim, Michael Elkin, and Fabian Kuhn. Distributed (Δ+1)-Coloring in Linear (in Δ) Time. *SIAM Journal on Computing*, 43(1):72–95, 2014. doi:10.1137/12088848X.

12 Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The Locality of Distributed Symmetry Breaking. *Journal of the ACM*, 63(3):1–45, 2016. doi:10.1145/2903137.

13 Yi-Jun Chang, Wenzheng Li, and Seth Pettie. An optimal distributed (Δ+1)-coloring algorithm? In *Proc. 50th ACM Symp. on Theory of Computing, (STOC)*, pages 445–456, 2018. doi:10.1145/3188745.3188964.

14 Pierre Fraigniaud, Marc Heinrich, and Adrian Kosowski. Local conflict coloring. In *Proc. 57th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 625–634, 2016.

15 Marc Fuchs and Fabian Kuhn. List defective colorings: Distributed algorithms and applications. *CoRR*, abs/2304.09666, 2023. doi:10.48550/arXiv.2304.09666.

16 Mohsen Ghaffari, Christoph Grunau, Bernhard Haeupler, Saeed Ilchi, and Václav Rozhon. Improved distributed network decomposition, hitting sets, and spanners, via derandomization. In *Proc. 34th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2023.

17 Mohsen Ghaffari, Christoph Grunau, and Václav Rozhon. Improved deterministic network decomposition. In *Proc. 32nd ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2904–2923, 2021. doi:10.1137/1.9781611976465.173.

18 Mohsen Ghaffari and Fabian Kuhn. Deterministic distributed vertex coloring: Simpler, faster, and without network decomposition. In *Proc. 62nd IEEE Symp. on Foundations of Computing (FOCS)*, 2021.

19 A.V. Goldberg, S.A. Plotkin, and G.E. Shannon. Parallel symmetry-breaking in sparse graphs. *SIAM Journal on Discrete Mathematics*, 1(4):434–446, 1988.

20 Magnús M. Halldórsson, Fabian Kuhn, Yannic Maus, and Tigran Tonoyan. Efficient randomized distributed coloring in CONGEST. In *Proc. 53rd ACM Symp. on Theory of Computing (STOC)*, pages 1180–1193, 2021.

21 Magnús M. Halldórsson, Fabian Kuhn, Alexandre Nolin, and Tigran Tonoyan. Near-optimal distributed degree+1 coloring. *CoRR*, abs/2112.00604, 2021.

22 David G. Harris, Johannes Schneider, and Hsin-Hao Su. Distributed (Δ +1)-coloring in sublogarithmic rounds. *J. ACM*, 65(4):19:1–19:21, 2018.

23 Fabian Kuhn. Local weak coloring algorithms and implications on deterministic symmetry breaking. In *Proc. 21st ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, 2009.

24 Fabian Kuhn. Faster deterministic distributed coloring through recursive list coloring. In *Proc. 32st ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 1244–1259, 2020.

25 Fabian Kuhn and Roger Wattenhofer. On the complexity of distributed graph coloring. In *Proc. 25th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 7–15, 2006.

26 Nathan Linial. Distributive graph algorithms – Global solutions from local data. In *Proc. 28th Symp. on Foundations of Computer Science (FOCS)*, pages 331–335. IEEE, 1987. doi:10.1109/SFCS.1987.20.

27 L. Lovász. On decompositions of graphs. *Studia Sci. Math. Hungar.*, 1:237–238, 1966.

28 Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 15(4):1036–1053, 1986. doi:10.1137/0215074.

29 Yannic Maus. Distributed graph coloring made easy. In *Proc. 33rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2021.

30 Yannic Maus and Tigran Tonoyan. Local conflict coloring revisited: Linial for lists. In *Proc. 34th Symp. on Distributed Computing (DISC)*, volume 179 of *LIPIcs*, pages 16:1–16:18, 2020.

31 Moni Naor. A lower bound on probabilistic algorithms for distributive ring coloring. *SIAM Journal on Discrete Mathematics*, 4(3):409–412, 1991. doi:10.1137/0404036.

32 Alessandro Panconesi and Aravind Srinivasan. On the complexity of distributed network decomposition. *Journal of Algorithms*, 20(2):356–374, 1996.

33 David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. Society for Industrial and Applied Mathematics, 2000. doi:10.1137/1.9780898719772.

34 Václav Rozhoň and Mohsen Ghaffari. Polylogarithmic-time deterministic network decomposition and distributed derandomization. In *Proc. 52nd ACM Symp. on Theory of Computing (STOC)*, pages 350–363, 2020.

35 Johannes Schneider and Roger Wattenhofer. A new technique for distributed symmetry breaking. In *Proc. 29th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 257–266, 2010. doi:10.1145/1835698.1835760.

36 Mario Szegedy and Sundar Vishwanathan. Locality based graph coloring. In *Proc. 25th ACM Symp. on Theory of Computing (STOC)*, pages 201–207, 1993.

A Main Oriented List Defective Coloring Algorithm

If we have an OLDC instance in which some nodes have colors with constant defect requirements, the number of h of γ-classes can be $\Theta(\beta)$. Because also the value of $\tau(h, |\mathcal{C}|, m)$ is linear in h, this means that even if $g = 0$ and even if $|\mathcal{C}|$ and m are both polynomial in β, the condition in Lemma 12 is of the form $\sum_{x \in L_v}(d_v(x)+1)^2 \geq \alpha\beta_v^2 \log^2\beta$. One of the $\log\beta$ factors comes from the fact that at the very beginning of the algorithm, every node v reduces its color list to a list in which all colors have approximately the same defect value. In the following, we show that at the cost of a more complicated algorithm, we can improve this $\log\beta$ factor to a factor of the form $\operatorname{poly}\log\log\beta$. In the following discussion, we assume that $g = 0$, but we note that along the way, we will have to use Lemma 12 with positive g as a subroutine.

In order to obtain the improvement, we first want an algorithm where for computing the 0-round problem P_2, a node v of some γ-class only needs to compete with outneighbors of the same γ-class. For this, we use an iterative approach to solve P_2 and P_1. For each $i \in [h]$, let $V_i \subseteq V$ be the set of nodes in γ-class i. For each node $v \in V_i$ that is colored with some color x, we will make sure that v has at most $d_v(x)/4$ outneighbors of color x in γ-classes j for $j < i$, at most $d_v(x)/4$ outneighbors of color x in the same γ-class and that v has at most $d_v/2$ outneighbors in γ-classes j for $j > i$. Thus, we assume that each node $v \in V$ only uses the part $L_{v,i}$ of its L_v consisting of colors with a defect d_v such that $\gamma_v = 2^i \geq 4\beta_v/(d_v+1)$. We then iterate over the γ classes $i \in [h]$ in increasing order. In iteration i, we solve problems P_2 and P_1 for the nodes in V_i. When dealing with nodes in V_i, we can therefore assume that for all outneighbors in $u \in V_j$ for $j < i$, the list C_u (i.e., the output of problem P_1) is already computed. We then remove each color x from the list $L_{v,i}$ for which there are more than $d_v(x)/4$ outneighbors $u \in V_1 \cup \cdots \cup V_{i-1}$ for which $x \in C_u$. In this way, we guarantee that v cannot choose a color with defect more than $d_v/4$ to outneighbors in lower γ-classes even before solving P_2 for node v. We can then solve P_2 and P_1 by only considering outneighbors in V_i. Of course, we have to make sure that even after removing colors from $L_{v,i}$, the list of v is still sufficiently large to solve problem P_2. The advantage of only having to consider neighbors in V_i when solving P_2 and P_1 is that in the condition on $\sum_{x \in L_{v,i}} (d_v(x)+1)^2$, we can replace the outdegree β_v of v by the number of outneighbors $\beta_{v,i}$ that v has in V_i. This gives us more flexibility in the choice of v's γ-class i. If $\sum_{x \in L_{v,i}} (d_v(x)+1)^2$ is large, v can choose γ-class i and tolerate many outneighbors in the same γ-class and if $\sum_{x \in L_{v,i}} (d_v(x)+1)^2$ is small, v can only choose γ-class i if a small number of outneighbors choose γ-class i. We will see that the problem of choosing a good γ-class can be phrased as an OLDC problem that can be solved by using Lemma 12 with appropriate parameters. The following technical Lemma 15 assumes that the γ-classes are already assigned and it formally proves under which conditions the above algorithmic idea allows to solve a given OLDC instance.

▶ **Lemma 15.** *Let $G = (V, E)$ be a directed graph that is equipped with an initial proper m-coloring. Let $h \geq 1$ be an integer parameter and assume that every node $v \in V$ is in some γ-class $i_v \in [h]$. For every $i \in [h]$, let V_i be the nodes in γ-class i and let $\beta_{v,i}$ be the number of outneighbors of v in V_i. Each node v has a color list $L_v \subseteq \mathcal{C}$ and one fixed defect value d_v, i.e., $d_v(x) = d_v$ for all $x \in L_v$. We further define $\tau := \tau(h, \mathcal{C}, m)$ and some integer parameter $q \in [\tau]$. We assume that for all $v \in V$*

$$\forall v \in V \;:\; \frac{4 \cdot \max\left\{\beta_{v,i_v}, \frac{\beta_v}{q}\right\}}{d_v + 1} \leq 2^{i_v} \quad \textit{and} \quad |L_v| \geq \left[\alpha \cdot 4^{i_v} + \frac{4}{d_v+1} \cdot \sum_{j=i_v - \lfloor \log q \rfloor}^{i_v - 1} \beta_{v,j} \cdot 2^j\right] \cdot \tau$$

for a sufficiently large constant $\alpha > 0$. Then there is an $O(h)$-round algorithm that assigns each node v a color $x \in L_v$ such that every node $v \in V$ has at most d_v outneighbors of color x. The algorithm requires messages consisting of $O(\min\{\Lambda \log |\mathcal{C}|, |\mathcal{C}|\} + \log\log \beta + \log m)$ bits.

We are now ready to prove Theorem 2, our main contribution. In the following $\hat{\beta}_v$ is the outdegree of v rounded up to the next integer power of 2. Further $\hat{\beta} := \max_{v \in V} \hat{\beta}_v$. Note that for all v, we have $\hat{\beta}_v \leq 2\beta_v$. The following lemma is a rephrasing of the theorem. By adjusting the constant α, Theorem 2 follows from Lemma 16 because for $h = \lceil \log \hat{\beta} \rceil$ and $h' = \lceil \log 4h \rceil$, $\tau(h, \mathcal{C}, m) = O(\log \beta + \log\log |\mathcal{C}| + \log\log m)$ and $\tau(h', [h], m) = O(\log\log \beta + \log\log m)$.

▶ **Lemma 16.** *Let $G = (V, E)$ be a properly m-colored directed graph and let $h := \lceil \log \hat{\beta} \rceil$ and $h' := 4^{\lceil \log_4 \log 8h \rceil}$. Assume that we are given an OLDC instance on G for which*

$$\forall v \in V \;:\; \sum_{x \in L_v} \left(d_v(x) + 1\right)^2 \geq \alpha^2 \cdot \hat{\beta}_v^2 \cdot \tau \cdot \bar{\tau} \cdot h'^2, \tag{8}$$

where $\tau = 4^{\lceil \log_4 \tau(h, \mathcal{C}, m) \rceil}$ and $\bar{\tau} = 4^{\lceil \log_4 \tau(h', [h], m) \rceil}$. Then, there is a deterministic distributed algorithm that solves this OLDC instance in $O(\log \beta)$ rounds using $O\big(\min\{|\mathcal{C}|, \Lambda \cdot \log |\mathcal{C}|\} + \log \beta + \log m\big)$-bit messages.

Proof. First note that w.l.o.g., we can assume that for all v and all $x \in L_v$, $(d_v(x)+1)^2$ and α are both integer powers of 4. We can just round up α and round down $d_v(x)$ to the next value for which this is true. We then just need to choose the constant α slightly larger. With those assumptions, the right-hand side of (8) is then a integer power of 4. For each node v, we define $R_v := \alpha \cdot \hat{\beta}_v^2 \cdot \bar{\tau} \cdot h'^2$. For every $v \in V$ and every $x \in L_v$, we then have $\frac{R_v}{(d_v(x)+1)^2} = 4^\mu$ for some $\mu \in [h]$. We can therefore partition each list L_v in to lists $L_v = L_{v,1} \cup \dots \cup L_{v,h}$ such that for all $\mu \in [h]$, $L_{v,\mu}$ consists of the colors $x \in L_v$ for which $\frac{R_v}{(d_v(x)+1)^2} = 4^\mu$. The algorithm to solve the given OLDC instance consists of two phases. In the first phase, every node $v \in V$ chooses its γ-class, which is an integer $i_v \in [h]$. In the second phase, we then use Lemma 15 to solve the OLDC instance.

We first discuss the objective of the first phase and we consider some node v. For every $\mu \in [h]$, we define $D_{v,\mu} := \sum_{x \in L_{v,\mu}} (d_v(x)+1)^2$ and $D_v := \sum_{\mu=1}^h D_{v,\mu} = \sum_{x \in L_v} (d_v(x)+1)^2$. For each $\mu \in [h]$, we further define $\lambda_{v,\mu} \in (0,1]$ as follows

$$\lambda_{v,\mu} := \begin{cases} 0 & \text{if } D_{v,\mu}/D_v < 1/(2h) \\ 4^{\lfloor \log_4 (D_{v,\mu}/D_v) \rfloor} & \text{otherwise.} \end{cases}$$

In the next part, we make a case distinction and first assume that $\lambda_{v,\mu} < 1/4$ for all μ. The case when there is some μ with $\lambda_{v,\mu} \geq 1/4$ is a simple case that we discuss later.

Case I: $\forall \mu \in [h] : \lambda_{v,\mu} < 1/4$. Note that the definition of $\lambda_{v,\mu}$ implies that for all μ where $\lambda_{v,\mu} \neq 0$, $\lambda_{v,\mu} > 1/(8h)$ and $\lambda_{v,\mu} = 4^{-r_{v,\mu}}$ for some integer $r_{v,\mu} \in \{0, \dots, \lceil \log_4 4h \rceil\}$. Note also that the values of $D_{v,\mu}$ for which $\lambda_{v,\mu} = 0$ sum up to at most $D_v/2$ and therefore

$$\sum_{\mu=1}^{h} \lambda_{v,\mu} \geq \frac{1}{8}.$$

For every $v \in V$, we next define a function $f_v : [h] \to [h] \cup \{\perp\}$ such that for all $\mu \in [h]$, $f_v(\mu) = \mu - r_{v,\mu} + 2$ if $\lambda_{v,\mu} > 0$ and $f_v(\mu) = \perp$ otherwise. Note that $0 < \lambda_{v,\mu} < 1/4$ implies that $f_v(\mu) \leq h$. For every $\mu \in [h]$, we next also define a second function $i_v : [h] \to [h] \cup \{\perp\}$ as follows. For every μ, we set $i_v(\mu) = f_v(\mu)$ if $f_v(\mu) = \perp$ or if $f_v(\mu) \geq 1$ and there is no $\mu' < \mu$ for which $f_v(\mu') = f_v(\mu)$. Otherwise, we set $i_v(\mu) = \perp$. Note that for any two $\mu, \mu' \in [h]$ with $\mu \neq \mu'$, we either have $i_v(\mu) = i_v(\mu') = \perp$ or we have $i_v(\mu) \neq i_v(\mu')$. We next show that

$$\sum_{\mu \in [h] : i_v(\mu) \neq \perp} \lambda_{v,\mu} \geq \frac{2}{3} \cdot \sum_{\mu \in [h] : f_v(\mu) \neq \perp \wedge f_v(\mu) \geq 1} \lambda_{v,\mu} \geq \frac{2}{3} \cdot \left(\frac{1}{8} - \frac{1}{48} \right) \geq \frac{1}{20}. \tag{9}$$

To see this, consider some μ for which $\lambda_{v,\mu} > 0$. Note that for $f_v(\mu) < 1$, we need $r_{v,\mu} \geq \mu + 2$ and therefore $\lambda_{v,\mu} \leq 4^{-\mu-2}$. Thus, the sum over those $\lambda_{v,\mu}$ is at most $\frac{4}{3} \cdot 4^{-1-2} = 1/48$. It therefore remains to show that the sum over the $\lambda_{v,\mu}$ for which $f_v(\mu) \geq 1$, but $i_v(\mu) = \perp$ is at most a third the sum over the $\lambda_{v,\mu}$ for which $f_v(\mu) \geq 1$. We have $i_v(\mu) = \perp$ and $f_v(\mu) \geq 1$ iff there is a $\mu' < \mu$ for which $f_v(\mu') = f_v(\mu)$. For $f_v(\mu') = f_v(\mu)$, we need to have $\lambda_{v,\mu} = \lambda_{v,\mu'} \cdot 4^{\mu' - \mu}$. Consider some value $z \geq 1$ for which there is a value μ_z with $f_v(\mu_z) = z$ and assume that μ_z is the smallest such value. The sum over all λ_μ for $\mu > \mu_z$ and $f_v(\mu) = f_v(\mu_z)$ is at most $\sum_{\mu=\mu_z+1}^{\infty} \lambda_{\mu_z} \cdot 4^{\mu_z - \mu} = \lambda_{\mu_z}/3$. This concludes the proof of Inequality (9).

In order to assign a γ-class i_v to every node $v \in V$, we define another (generalized) OLDC instance. For this instance, the "color" list of node v is $\mathcal{L}_v = \{i_v(\mu) : i_v(\mu) \neq \perp\}$. For every $i \in \mathcal{L}_v$, we define the inverse function $\mu_v(i)$ to be the value μ for which $i_v(\mu) = i$. For each color $i \in \mathcal{L}_v$, we then define a defect $\delta_{v,i}$ as

$$\delta_{v,i} := \left\lfloor \sqrt{\lambda_{v,\mu_v(i)} \cdot R_v} \right\rfloor.$$

We further define $q := h$ and $g := \lfloor \log h \rfloor$. We then want to find an assignment of values $i_v \in \mathcal{L}_v$ to each node such that for every $v \in V$, the number of outneighbors u for which $i_u \in [i_v - g, i_v]$ is at most δ_{v,i_v}. We next show that such an assignment of γ-classes i_v satisfies the requirement needed by Lemma 15 and we can therefore use it to efficiently solve the original OLDC instance. We afterwards show that the generalized OLDC instance to find the values i_v satisfies the requirement of Lemma 12.

Let us therefore assume that we have an assignment of γ-class i_v to the nodes that solve the above generalized OLDC problem. For each $i \in [h]$, we again use V_i to denote the set of nodes v with $i_v = i$ and we assume $\beta_{v,i}$ is the number of outneighbors of v in V_i. The fact that v has at most δ_{v,i_v} outneighbors u with $i_u \in [i_v - g, i_v]$ implies that $\delta_{v,i_v} \geq \beta_{v,j}$ for all $j \in [i_v - g, i_v]$. For all $v \in V$, we have

$$\delta_{v,i_v} = \left\lfloor \sqrt{\lambda_{v,\mu_v(i)} \cdot R_v} \right\rfloor \overset{(\lambda_{v,\mu_v(i)} \geq 1/(8h))}{\geq} \left\lfloor \sqrt{\frac{R_v}{8h}} \right\rfloor = \left\lfloor \sqrt{\frac{\alpha \cdot \hat{\beta}_v^2 \cdot \bar{\tau} \cdot h'^2}{8h}} \right\rfloor \geq \frac{\hat{\beta}_v}{h}.$$

The last inequality follows because $h, \tau, \bar{\tau}, h' \geq 1$ and if we choose $\alpha \geq 8$. For the following calculations, we define $\mu_v := \mu_v(i_v) = i_v + r_{v,\mu} - 2$. Note that if v chooses γ-class i_v, it uses the colors in L_{v,μ_v}. All those colors have a defect d_v such that $(d_v + 1)^2 = R_v/4^{\mu_v}$. Using $q = h$, we therefore have

$$\begin{aligned}
\frac{4 \cdot \max\left\{\beta_{v,i_v}, \frac{\beta_v}{q}\right\}}{d_v + 1} &\leq \frac{4 \cdot \delta_{v,i_v}}{d_v + 1} \\
&\leq \sqrt{\frac{16\lambda_{v,\mu_v} \cdot R_v}{(d_v + 1)^2}} \\
&= \sqrt{16\lambda_{v,\mu_v} \cdot 4^{\mu_v}} \\
&= \sqrt{4^2 \cdot 4^{-r_{v,\mu_v}} \cdot 4^{i_v + r_{v,\mu_v} - 2}} = 2^{i_v}.
\end{aligned}$$

The first part of the requirement of Lemma 15 is therefore satisfied. For the second part, recall that v uses the colors in L_{v,μ_v} and that $D_{v,\mu_v} = \sum_{x \in L_{v,\mu_v}} (d_v(x) + 1)^2 = |L_{v,\mu_v}| \cdot (d_v + 1)^2$, where d_v is defined as before. We have

$$|L_{v,\mu_v}| \geq \frac{\lambda_{v,\mu_v} D_v}{(d_v + 1)^2} \geq \frac{\lambda_{v,\mu_v} \cdot \alpha\tau \cdot R_v}{(d_v + 1)^2} = 4^{-r_{v,\mu_v}} \cdot \alpha\tau \cdot 4^{\mu_v} = \frac{\alpha}{16} \cdot 4^{i_v} \cdot \tau. \tag{10}$$

Before we continue, we switch to Case II.

Case II: $\exists \mu \in [h] : \lambda_{v,\mu} \geq 1/4$. Before looking of the problem of assigning the γ-classes, we have a look at the requirements for Lemma 15 in Case II, i.e., if there is a $\mu \in [h]$ for which $\lambda_{v,\mu} \geq 1/4$. Let μ_v be one such value μ. In this case, we set $i_v = \mu_v$, $\mathcal{L}_v = \{i_v\}$, and $\delta_{v,i_v} := \lfloor \sqrt{R_v}/4 \rfloor$. We then have $\delta_{v,i_v} = \left\lfloor \sqrt{\alpha \hat{\beta}_v^2 \bar{\tau} h'^2 / 16} \right\rfloor \geq \beta_v$. The last inequality holds if $\alpha \geq 16$ because $\bar{\tau}$ and h' are positive integers. We therefore clearly have $\delta_{v,i_v} \geq \max\{\beta_{v,i_v}, \beta_v/q\}$ and therefore

$$\frac{4 \cdot \max\left\{\beta_{v,i_v}, \frac{\beta_v}{q}\right\}}{d_v + 1} \leq \frac{4\delta_{v,i_v}}{d_v + 1} \leq \sqrt{\frac{16R_v}{16(d_v + 1)^2}} = 2^{\mu_v} = 2^{i_v}.$$

Hence, the first part of the requirement of Lemma 15 also holds in Case II. Similarly to Case I, we can lower bound the size of the color list L_{v,μ_v} that is used by v:

$$|L_{v,\mu_v}| \geq \frac{\lambda_{v,\mu_v} D_v}{(d_v+1)^2} \geq \frac{\alpha\tau \cdot R_v}{4(d_v+1)^2} = \frac{\alpha}{4} \cdot 4^{i_v} \cdot \tau. \tag{11}$$

The bound given by (10) therefore also holds in Case II.

We now continue considering both cases together. It remains to show (to apply Lemma 15) that

$$|L_{v,\mu_v}| \geq \alpha' \cdot 4^{i_v} \cdot \tau + \frac{4}{d_v+1} \cdot \sum_{j=i_v-\lfloor \log q \rfloor}^{i_v-1} \beta_{v,j} \cdot 2^j \cdot \tau \tag{12}$$

for some constant α' that can be chosen as large as needed by choosing the constant α sufficiently large. Note that we have $g = \lfloor \log q \rfloor$ and thus for $j \in [i_v - \lfloor \log q \rfloor, i_v]$, $\beta_{v,j} \leq \delta_{v,i_v}$. We therefore have

$$\frac{4}{d_v+1} \cdot \sum_{j=i_v-\lfloor \log q \rfloor}^{i_v-1} \beta_{v,j} \cdot 2^j \cdot \tau \leq \frac{4\delta_{v,i_v}\tau}{d_v+1} \cdot \sum_{\ell=1}^{g} 2^{i_v-\ell} < \frac{4\delta_{v,i_v}}{d_v+1} \cdot 2^{i_v} \cdot \tau.$$

We have already seen that $4\delta_{v,i_v}/(d_v+1) \leq 2^{i_v}$ and the bound in the above inequality can therefore be upper bounded by $4^{i_v}\tau$. For every constant $\alpha' > 0$, we can therefore choose a constant $\alpha > 0$ such that the bound in (12) is upper bounded by the bound in (10). This shows that if v is in Case I, node v satisfies the requirements to apply Lemma 15.

We next also show that the assignment of γ-classes i_v can be done by using the algorithm of Lemma 12. Recall that every node v needs to pick an $i_v \in \mathcal{L}_v$ such that the total number of outneighbors u that pick $i_u \in [i_v - g, i_v]$ is at most δ_{v,i_v}. To apply Lemma 12, we have to lower bound $\sum_{i\in\mathcal{L}_v}(\delta_{v,i}+1)^2$. We have

$$\begin{aligned}\sum_{i\in\mathcal{L}_v}(\delta_{v,i}+1)^2 &\geq \min\left\{\frac{1}{16}, \sum_{i\in\mathcal{L}_v}\lambda_{v,\mu_v(i)}\right\} \cdot R_v\\ &\overset{(9)}{\geq} \frac{1}{20} \cdot R_v\\ &= \frac{1}{20} \cdot \alpha \cdot \hat{\beta}_v^2 \cdot \bar{\tau} \cdot h'^2.\end{aligned}$$

Note that we have $g = \lfloor \log h \rfloor$ and $h' \geq \log(8h)$. We therefore have $2h' \geq 2g+1$. By choosing a sufficiently large constant α, the above inequality therefore implies that the requirements of Lemma 12 are satisfied as long as the value of $\bar{\tau}$ is sufficiently large. We have $\bar{\tau} = \tau(h', [h], m)$. Note that the color space of the OLDC problem that we use to assign the values i_v is $[h]$. Recall that for all v and μ, $\lambda_{v,\mu} = 0$ or $\lambda_{v,\mu} \geq 1/(8h)$. For each node $v \in V$, we therefore have

$$\min_{i\in\mathcal{L}_v} \delta_{v,i} \geq \min\left\{\frac{\sqrt{R_v}}{4}, \sqrt{\lambda_{v,\mu_v(i)}R_v}\right\} \geq \min\left\{\frac{1}{4}, \frac{1}{\sqrt{8h}}\right\} \cdot \sqrt{R_v} \geq \frac{\sqrt{R_v}}{8h} \geq \frac{\beta_v}{8h}.$$

We therefore have

$$\max_{v\in V} \frac{\beta_v}{\min_{i\in\mathcal{L}_v}\delta_{v,i}+1} \leq 8h.$$

Because we have $h' \geq \log(8h)$, the choice $\bar{\tau} = \tau(h', [h], m)$ satisfies the requirements of Lemma 12 and we can therefore compute the γ-classes i_v for all nodes v by using the algorithm of Lemma 12.

We will now analyze the required message size and round complexity of the algorithm. In the first phase the OLDC problem on color lists $\mathcal{L}_v$ and defects δ_{v,i_v} has to be solved. As $|\mathcal{L}_v| \leq h$ and transmitting such a single defect does not need more than $\log h$ bits, by Lemma 12, solving this OLDC instance the maximum message size is $O(h + \log h + \log m) = O(h + \log m)$ bits. The number of rounds needed for this first phase are $O(h')$. In the second phase we have messages of size $O(\min\{|\mathcal{C}| + \Lambda \log |\mathcal{C}|\} + \log \log \beta)$ (the initial color is already known in the second phase) due to Lemma 12. The round complexity of the second phase is $O(h)$ due to Lemma 15. Combining both phases and using that $h' = O(h) = O(\log \beta)$, the maximum message size and the runtime are as stated. ◀

Conditionally Optimal Parallel Coloring of Forests

Christoph Grunau
ETH Zürich, Switzerland

Rustam Latypov
Aalto University, Finland

Yannic Maus
TU Graz, Austria

Shreyas Pai
Aalto University, Finland

Jara Uitto
Aalto University, Finland

Abstract

We show the first conditionally optimal deterministic algorithm for 3-coloring forests in the low-space massively parallel computation (MPC) model. Our algorithm runs in $O(\log \log n)$ rounds and uses optimal global space. The best previous algorithm requires 4 colors [Ghaffari, Grunau, Jin, DISC'20] and is randomized, while our algorithm are inherently deterministic.

Our main technical contribution is an $O(\log \log n)$-round algorithm to compute a partition of the forest into $O(\log n)$ ordered layers such that every node has at most two neighbors in the same or higher layers. Similar decompositions are often used in the area and we believe that this result is of independent interest. Our results also immediately yield conditionally optimal deterministic algorithms for maximal independent set and maximal matching for forests, matching the state of the art [Giliberti, Fischer, Grunau, SPAA'23]. In contrast to their solution, our algorithms are not based on derandomization, and are arguably simpler.

2012 ACM Subject Classification Theory of computation → Massively parallel algorithms

Keywords and phrases massively parallel computation, coloring, forests, optimal

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.23

Related Version *Full Version*: `https://arxiv.org/abs/2308.00355`

Funding *Rustam Latypov*: Academy of Finland, Grant 334238.
Yannic Maus: Austrian Science Fund (FWF), Grant P36280-N.
Shreyas Pai: Academy of Finland, Grant 334238.

1 Introduction

A recent sequence of papers investigates fundamental symmetry-breaking problems such as coloring, maximal independent set and maximal matching on trees [9, 7, 32, 22, 26]. We conclude, simplify and unify this line of work by giving a conceptually simple algorithm for 3-coloring, maximal independent set and maximal matching. We solve the three problems in a unified way by computing a so-called H-decomposition (we discuss these in more detail in Section 1.2). Even though such decompositions are the natural tool for solving the aforementioned problems on trees, computing them efficiently in the MPC model remained outside the reach of previous techniques.

▶ **Theorem 1.** *There are deterministic $O(\log \log n)$-round low-space MPC algorithms for 3-coloring, maximal matching and maximal independent set (MIS) on forests. These algorithms use $O(n)$ global space.*

The runtimes of our algorithms are conditionally optimal, conditioned on the 1 vs 2 cycle conjecture, at least if one restricts to so-called component stable algorithms [23, 15, 35, 41] (see Section 1.4 for a brief discussion about component-stability).

We note that algorithms for maximal matching and maximal independent set matching our guarantees are known from a very recent work [26]. However, their algorithms are quite complicated and technical, and use sophisticated derandomization techniques. Moreover, their techniques inherently cannot be used to color a tree with a small number of colors. Indeed, the 3-coloring problem is considered to be the hardest of the three problems, e.g., once such a coloring is known one can compute an MIS in $O(1)$ rounds. Additionally, a crucial property used in previous MPC algorithms for MIS and maximal matching is that any partial solution can be extended to a solution of the whole graph; a property that does not hold for 3-coloring. The best previous algorithm for coloring trees uses 4 colors and is randomized [22]. If one allows for randomization the single additional color makes the problem significantly easier by the following divide and conquer approach: if one partitions the tree into two parts by letting each node join one of the parts uniformly at random, the connected components induced by each part have logarithmic diameter. Once the diameter is small, one can use $O(\log\log n)$ MPC rounds to color each component independently with two colors in a brute force manner. In the next section we zoom out and present the bigger picture of our work.

1.1 MPC Model and Exponential Speed-Up Over LOCAL Algorithms

The Massively Parallel Computation (MPC) model [30] is a mathematical abstraction of modern frameworks of parallel computing such as Hadoop [43], Spark [44], MapReduce [18], and Dryad [29]. In the MPC model, we have M machines that communicate in all-to-all fashion, in synchronous rounds. In each round, every machine receives the messages sent in the previous round, performs (arbitrary) local computations, and is allowed to send messages to any other machine. Initially, an input graph of n nodes and m edges is arbitrarily distributed among the machines. At the end of the computation, each machine needs to know the output of each node it holds, e.g., their color in the vertex-coloring problem.

The MPC model is typically divided into 3 regimes according to the local space S. The *superlinear* and the *linear* regimes allow for $S = n^{1+\Omega(1)}$ and $S = \widetilde{O}(n)$ words[1] of space (memory) per machine. A word is $O(\log n)$ bits and is enough to store a node or a machine identifier from a polynomial (in n) domain. The local space restricts the amount of data a machine initially holds and is allowed to send and receive per round. Both linear and superlinear regimes allow for very efficient algorithms because machines can get a "global view" of the graph in the sense that it can store information for each node of the graph [31, 20, 24]. However, the growing size of most real-world graphs makes it impossible to get such a global view on a single machine and hence research in recent years has focused on the most challenging *low-space* (or *sublinear*) regime with $S = n^{\delta}$, for some constant $\delta < 1$, where we cannot even store the whole neighborhood of a single node in a single machine. As each machine can only get a local view there are close connections to the LOCAL model of distributed computing that we further elaborate on below.

Furthermore, we focus on the most restricted case of *linear global space*, i.e., $S \cdot M = \Theta(n + m)$. Notice that $\Omega(n + m)$ words are *required* to store the input graph.

[1] The $\widetilde{O}$ notation hides polylogarithmic factors.

The LOCAL Model and Graph Exponentiation. The LOCAL model is a classic model of distributed message passing. Each node of an input graph hosts a processor and the nodes communicate along the edges of the graph in synchronous rounds. The local computation, local space, and message sizes are unbounded in this model. Most research in the LOCAL model has focused on symmetry breaking problems like graph colorings, MIS, and maximal matchings. For most of these classic problems $O(\log n)$-round randomized algorithms are known [36, 1, 38] which can directly be translated to the MPC model. A major focus on recent and current research is to develop *sublogarithmic* MPC algorithms that beat the logarithmic baseline.

In fact, the strong connection between the models also shows up in faster algorithms, as almost all recent MPC algorithms for such problems are MPC-optimized implementations of algorithms that were originally developed for the LOCAL model. The main technique to obtain this speedup is the graph *exponentiation technique* [33]. It allows to gather the T-hop radius neighborhood of a node in $O(\log T)$ MPC rounds. So, as long as these neighborhoods fit the space constraints, after gathering them one can simulate T-round LOCAL algorithms locally to compute the output for each node. Furthermore, for component-stable algorithms, the connection also goes the other way around, that is, an $\Omega(T)$ lower bound on the round complexity in the LOCAL model implies an exponentially lower $\Omega(\log T)$ conditional lower bound in the MPC model. Thus, the holy grail is to obtain this exponential speedup over the LOCAL model. A central open problem in the area is to find an $O(\log \log n)$ round MPC algorithm for the classic MIS problem on general graphs, which enjoys a matching conditional $\Omega(\log \log n)$-round lower bound.

Unfortunately, we are very far from answering this question. The current state of the art is an $\widetilde{O}(\sqrt{\log \Delta} + \log \log \log n)$-round randomized MPC algorithm [25]. We note that we take into account the new results on network decomposition, which reduce the dependency on n [42, 21]. This result is obtained by combining the graph exponentiation technique with *sparsification* methods [33, 25]. The exponentiation technique is used to simulate Ghaffari's $O(\log \Delta + \text{poly} \log \log n)$-round MIS algorithm for the LOCAL model [19].

From a high-level perspective they break the LOCAL algorithm into $O(\sqrt{\log \Delta})$ phases each of length $T = O(\sqrt{\log \Delta})$. In the beginning of each phase, the graph is subsampled so that the maximum degree of any node is at most $2^{\sqrt{\log \Delta}}$. Then, we can gather the T-hop neighborhood of each node in $O(\log \log \Delta)$ MPC rounds and simulate T rounds of the LOCAL algorithm in a single MPC round. The main benefit of simulating (shorter) phases and subsampling to smaller degree graphs is to reduce the memory resources needed during the exponentiation technique. Unfortunately, this phase-based approach seems to hit a fundamental barrier at $\sqrt{\log \Delta}$ rounds, and it is unclear how to reduce memory usage without it. Due to little progress in improving on this result, recent research has focused on special graph classes such as trees and bounded arboricity graphs.

Symmetry Breaking on Trees and Bounded Arboricity Graphs. Studying low-space MPC algorithms for MIS on trees and forests has been fruitful. This line of work started with a randomized $O(\log^3 \log n)$ round low-space MPC algorithm for MIS and maximal matching on trees [9]. Later, the round complexity was first improved to $O(\log^2 \log n)$ [7] and finally to $O(\log \log n)$ [22], where both algorithms extend to low-arboricity graphs. The $O(\log \log n)$ algorithm is conditionally optimal, at least if one restricts oneself to component-stable algorithms. Finally, a recent work derandomized the $O(\log \log n)$ round algorithm using MPC specific derandomization techniques and thus obtained a deterministic $O(\log \log n)$ round MIS and Maximal Matching algorithm for trees and more generally low-arboricity graphs [26].

While especially the $O(\log\log n)$ round algorithms are quite technical and involved, all of the aforementioned previous algorithms rely on the same fundamental idea. Namely, to interleave graph exponentiation with the computation of partial solutions to rapidly decrease the maximum degree of the remaining graph. Unfortunately, it seems unlikely that such a rapid degree reduction is possible in general graphs; thus it seems that new approaches are necessary in order to get an $O(\log\log n)$ round algorithm for general graphs.

Also, their approach does not work for coloring a forest with a constant number of colors. The main reason is that they critically rely on the fact that any partial solution can be extended to a full solution, which is not the case for coloring a forest with a fixed number of colors.

1.2 Our Technical Contribution

We present a unified solution for 3-coloring, MIS, maximal matching that takes $O(\log\log n)$ rounds. The core technical contribution that unifies these is an efficient algorithm to compute *H-decompositions*.

▶ **Theorem 2.** *There is a deterministic $O(\log\log n)$-rounds low-space MPC algorithm that computes a strict H-decomposition with $O(\log n)$ layers on forests in $O(n)$ global space.*

H-decompositions were introduced to the area of distributed computing by Barenboim and Elkin [5]. An H-decomposition (of a forest) partitions the vertices of the graph into layers such that every node has at most two neighbors[2] in higher or equal layers. For forests an H-partition with $O(\log n)$ layers always exist. In the LOCAL model, such an H-decomposition immediately implies an algorithm for 3-coloring in $O(\log n)$ rounds. Essentially, one can iterate through the layers in a reverse order and color all nodes in a layer while avoiding conflicts with the already colored neighbors in higher layers.

The novelty of our approach is not that we use such decompositions to compute a 3-coloring of a forest, in fact, this straightforward approach has made it into the classrooms of many graduate programs of universities, but in the way how we compute it. We detail on our solution in more detail in the nutshell, but the main take-away is as follows. We steer every machine to learn some parts of the graph (to large extent in an uncoordinated fashion) such that every machine can compute a partial H-decomposition locally, which we can later unify to a global decomposition. We are not aware of any other MPC algorithm for H-decompositions with a similar approach.

Balanced Exponentiation. In order to achieve exponential speedup, our algorithms rely on graph exponentiation. However, there are no known sparsification techniques that can cope with the memory resources that are needed for the classic graph exponentiation technique. Instead, we provide a self-contained exponentiation procedure whose memory overhead is very mild on forests. To explain our procedure, we first need to define a *subtree*. A subtree is a subgraph of a tree, such that if it is removed, the rest of the tree stays connected. A node is *important* if it is contained in a subtree of size $n^{\delta/8}$. We present the following result (the formal statement appears in the full version).

[2] There are generalizations to higher number of neighbors that are important when dealing with bounded arboricity graphs [37, 6].

Let $0 < k \leq n^{\delta/8}$ be a parameter. There is a deterministic low-space MPC *algorithm that, given an n-node forest F, uses $O(\log k)$ rounds in which every important node $v \in F$ discovers its k-hop neighborhood in every direction of the graph, except for at most one.*

Given a node $v \in F$, we refer to each of its neighbors $x \in N(v)$ as a *direction* with regard to v. Informally, what node v can discover in direction x is simply the subgraph of F that is connected to v via x, which is uniquely defined, since F is a forest.

This result above may be of independent interest and may be useful to design algorithms for other graph problems. We obtain it by extending the exponentiation technique of a recent work by [3]. Their work designs an exponentiation technique which (almost) equals ours in the special case when k equals the maximum diameter of a component of the forest. In their work it is used in an $O(\log \mathrm{diam})$-round algorithm to compute the connected components of a forest. Later it has also been used to solve certain dynamic programming tasks on tree-structured data [28], also in time that is logarithmic in the diameter. In the full version we present a more detailed discussion on the similarities and the difference between the exponentiation result in this work and the one in their work, and why our result requires a different analysis. The main benefit of our result is that a flexible choice of k allows the runtime and space to be small, if the required “view” for the nodes is small, which we heavily utilize in our algorithm to compute H-decompositions.

1.3 Our Method in a Nutshell

As mentioned in the previous section, our key technical contribution is to compute a so-called H-decomposition of the input forest F. In particular, the goal is to compute a partition $V(F) = V_1 \sqcup V_2 \sqcup \ldots \sqcup V_L$ of the vertices into $L = O(\log n)$ layers such that each node in V_i has at most two neighbors in $\bigcup_{j \geq i} V_j$. There exists a simple peeling algorithm which computes such a partition; iteratively peel off all nodes of degree at most 2 and define V_i as the set of nodes that got peeled off in the i-th iteration. A simple calculation shows that at least half of all the remaining nodes get peeled off in each iteration, and hence we get a decomposition into $O(\log n)$ layers. Moreover, one can determine the iteration in which a node gets peeled off by only looking at its $O(\log n)$-hop neighborhood. Thus, if we could compute for a given node its entire $O(\log n)$-neighborhood and store it in a single machine, then we could locally determine the layer of that node with no further communication.

One way to compute the $O(\log n)$-neighborhood of each node in the MPC model is the well-known graph exponentiation technique. Generally speaking, graph exponentiation allows to learn the 2^i-hop neighborhood of each node in $O(i)$ MPC rounds. Thus, we could in principle hope to learn the $O(\log n)$-hop neighborhood of each node in just $O(\log \log n)$ rounds. However, one obviously necessary precondition of the graph exponentiation technique is that the $O(\log n)$-hop neighborhood of each node has size n^δ, as otherwise we cannot possibly store the neighborhood in one machine. This is quite a limiting condition. If the input is for example a star, even the two-hop neighborhood of each node contains $\Omega(n)$ vertices. Moreover, even if each local neighborhood would fit into one machine, the global space required to store all the neighborhoods might still be prohibitively large, especially if one aims for near-linear global space.

Thus, we cannot use the vanilla graph exponentiation technique. Instead, we use the balanced graph exponentiation technique for forests mentioned in the previous section. The output guarantee of the balanced exponentiation algorithm, running in $O(\log \log n)$ rounds, weakens the guarantee that each node sees its $O(\log n)$-hop in two ways. First, it only gives a guarantee for nodes that are contained in a sufficiently small subtree, namely of size at most n^δ. Second, for each node v in a small subtree, it computes all nodes of distance $O(\log n)$, except for nodes in one direction.

We start by briefly discussing how one can deal with the first shortcoming. If one iteratively removes all nodes that are contained in a subtree of size at most x from F and all nodes of degree at most 2, then all nodes are removed within $O(\log_x(n))$ iterations. This fact was used in similar forms in previous results and for completeness we give a standalone proof (see Lemma 7). Thus, if we repeatedly assign nodes in subtrees of size at most n^δ and nodes of degree at most 2 to one of $O(\log n)$ layers, then after $O(1/\delta)$ iterations, we assigned each node to one of $O((1/\delta)\log n)$ layers. Thus, it intuitively suffices to focus on nodes in subtrees of size at most n^δ. Section 4 gives a formal treatment of this argument.

The more severe difficulty stems from the fact that there might not be a single node in the forest for which we have stored its entire $O(\log n)$-neighborhood in one machine. This makes it impossible to locally determine the layer of each node, or even a single one, assigned by the simple peeling process described in the beginning. Instead, each node v locally simulates a conservative variant of the peeling algorithm described above; in each iteration not all the nodes of degree at most 2 are removed, but only those that v has stored in its machine. Note that if v has strictly more than 2 neighbors not stored in its machine, then the conservative peeling algorithm would never peel off v. Moreover, even if v would eventually be peeled off, then there is no guarantee that it happens within the first $O(\log n)$ iterations. However, the fact that v has stored all the nodes in its $O(\log n)$-hop neighborhood except for nodes in one direction in its machine suffices to show that v gets peeled off within the first $O(\log n)$ iterations (see Lemma 21). Thus, each node v locally computes a layering $V^v = V_1^v \sqcup \ldots V_L^v$ for some $L = O(\log n)$ such that $v \in V^v$ and each node in V_i^v has at most two neighbors contained in $\left(\bigcup_{j \geq i} V_j^v\right) \cup (V(F) \setminus V^v)$. As some nodes might not be assigned to any layer, we refer to such a decomposition as a partial H-decomposition. Note that a node might get assigned to different layers from different nodes. Fortunately, this is not a problem because of the following nice structural property about (partial) H-decompositions: if we are given multiple (partial) H-decompositions, then we can get another (partial) H-decomposition by assigning each node to the smallest layer assigned by any of the H-decompositions. This structural observation allows us to combine the different locally computed (partial) H-decompositions into a single partial H-decomposition where each node in a small subtree is assigned to one of the $O(\log n)$ layers.

Rooted vs. Unrooted Forests. Our results are for unrooted forests, which are indeed more difficult than rooted forests. In fact, the fastest known MPC algorithm to root a forest takes $O(\log \operatorname{diam})$ rounds (and at least on general forests this runtime is conditionally tight) [3]; so rooting the forest does not fit our time budget of $O(\log\log n)$ rounds. Many steps of our algorithm would simplify (or maybe even allow for alternative solutions) if the forest was rooted. For example, in a directed forests, we would not need our balanced exponentiation procedure. One can show that nodes can just exponentiate towards their children until the local memory is full without breaking any global memory bounds. However we would still need our combinatorial algorithm for creating a (global) H-partition. Observe that [3] also contains a $O(\log \operatorname{diam})$-round 2-coloring algorithm for rooted constant-degree forests, which can be generalized to rooted unbounded-degree forests.

1.4 Further Related Work

Component Stability. Roughly speaking, an MPC algorithm is component-stable, if the outputs of nodes in different components are independent of each other. Low-space component-stable MPC algorithms are closely connected to algorithms in the LOCAL model and this connection was used to lift (unconditional) lower bounds from the LOCAL model into conditional lower bounds in the MPC model [23]. Under the 1 vs 2 cycle conjecture, this technique turns an $\Omega(T)$-round lower bound in LOCAL into an $\Omega(\log T)$ lower bound in low-space MPC. This approach was used to establish, among others, $\Omega(\log\log n)$ randomized lower bounds for MIS and maximal matching. Later, the technique was extended to deterministic component-stable algorithms as well [15]. While the assumption of component-stability might seem very natural to MPC algorithms, it is known that component-instability can help. For example, any component-stable algorithm for finding an independent set of size $\Omega(n/\Delta)$ requires $\Omega(\log\log^* n)$ rounds, while there is an $O(1)$-round algorithm that is not component-stable [15].

log(diam) Algorithms on Forests. There are surprisingly few works with a strict log(diam) runtime for any graph families in any MPC regimes, where diam refers to the diameter. To our knowledge, the only existing ones are low-space algorithms for forests [3, 28]. The authors of [3] show that connectivity, rooting, and all LCL (locally checkable labeling) problems can be solved on forests in $O(\log \text{diam})$ using optimal global space $O(n)$. The authors of [28] build on top of the works of [3] by introducing a framework to solve dynamic programming tasks and optimization problems, all in time log(diam) and global space $O(n)$. We note that given a double-logarithmic dependency on n, the connectivity problem can be solved on general graphs (even deterministically) in $O(\log \text{diam} + \log\log n)$ time using linear total space [8, 13]. Also, the 1 vs 2 cycle conjecture directly rules out an $o(\log \text{diam})$ for connectivity.

Symmetry-Breaking on General Graphs. In general graphs, $(\Delta + 1)$-vertex coloring is an intensively studied symmetry breaking problem, where Δ is the maximum degree of the graph. A series of works [39, 4, 40, 11] for Congested Clique model which is similar to the MPC model with linear local memory has culminated in a deterministic $O(1)$-round algorithm [17].

In the low-space MPC model, the first algorithm for the problem was randomized and used $O(\log\log\log n)$ rounds with almost linear $\widetilde{O}(m)$ global space [11][3]. By derandomizing the classic logarithmic-time algorithms, one can obtain an $O(\log\Delta + \log\log n)$-round algorithm for $(\Delta + 1)$-coloring, MIS, and maximal matching [14, 17]. For coloring, this was improved to $O(\log\log\log n)$ through derandomizing a tailor-made algorithm [16]. The deterministic algorithms require $n^{1+\Omega(1)}$ global space.

1.5 Outline

We define strict H-decompositions in Section 3, and then we show how to compute them in $O(\log\log n)$ low-space MPC rounds using $O(n \cdot \text{poly}(\log n))$ global space in Section 4. The key subroutine for the algorithm in Section 4 is discussed in Appendix A. In Section 5 we show how to use these decompositions to compute a coloring, MIS, and matching. In Appendix B we show how to reduce the global memory usage of our algorithms from $O(n \cdot \text{poly}(\log n))$ to $O(n)$. Due to space constraints, the balanced exponentiation procedure and the missing proofs of Section 4 appear in the full version. Many of the proofs are omitted due to the page limit and deferred to the full version.

[3] The $O(\sqrt{\log\log n})$ runtime stated in the paper is automatically improved to $O(\log\log\log n)$ through developments in network decomposition [42].

2 Preliminaries and Notation

The input graph is an undirected, finite, simple forest $F = (V, E)$ with $n = |V|$ nodes and $m = |E|$ edges such that $E \subseteq [V]^2$ and $V \cap E = \emptyset$. For a subset $S \subseteq V$, we use $G[S]$ to denote the subgraph of G induced by nodes in S.

Let $\deg_F(v)$ denote the degree of a node v in F and let Δ denote the maximum degree of F. For node set $S \subseteq V(F)$ and a node $v \in S$ we write $\deg_S(v)$ for the degree of v in $F[S]$. The distance $d_F(v, u)$ between two vertices v, u in F is the number of edges in the shortest $v - u$ path in F; if no such path exists, we set $d_F(v, u) := \infty$. Sometimes we simply write $\deg(v)$ and $d(v, u)$ if it is clear from context that we refer to the degree and distance in graph F. The greatest distance between any two vertices in F is the diameter of F, denoted by $\mathrm{diam}(F)$.

For each node v and for every $k \in \mathbb{N}$, we denote the k-hop (or k-radius) neighborhood of v as $N^k(v) = \{u \in V : d(v, u) \leq k\}$. Set $N^1(v)$ is simply the set of neighbors of v, to which often refer to as $N(v)$. We often consider sets of nodes S from which we need to remove a single node u. Hence, we use the notation $S \setminus u$ as a shorthand for $S \setminus \{u\}$.

3 Strict H-decompositions

We begin with the formal definition of an H-decomposition, that is, a partition of the graph into layers such that every node has at most two neighbors in higher or equal layers. We also extend the definition to the setting where some nodes remain without a layer.

▶ **Definition 3** ((Partial) H-Decomposition). *Let F be a forest and* layer: $V(F) \mapsto \mathbb{N} \cup \{\infty\}$. *For $i \in \mathbb{N} \cup \{\infty\}$ define $V_i = \{v \in V(F) \mid \mathrm{layer}(v) = i\}$, $V_{\geq i} = \bigcup_{j \geq i} V_j$.*

We say that layer *is a* partial H-decomposition *if $deg_{V_{\geq i}}(v) \leq 2$ holds for every v with* $\mathrm{layer}(v) = i$. *We speak of an* H-decomposition *if $V_\infty = \emptyset$, and $L = \max\{\mathrm{layer}(v) \mid v \in V(F), \mathrm{layer}(v) \neq \infty\}$ is the* length *of the decomposition.*

We also refer to the V_i's as the *layers* of the (partial) H-decomposition.

Why 3-coloring and strict H-decompositions? H-decompositions were introduced to the area of distributed computing by Barenboim and Elkin [5]. Nowadays, they are a frequent tool in the area and by increasing the degree bound 2 to $(2 + \varepsilon)a$ the concept also extends to graphs with arboricity at most a (this is the original setting considered in [5]). More generally, it can be shown that H-decompositions with $O(\log n)$ layers exist. In the LOCAL model, an H-decomposition of a tree with $O(\log n)$ layers can be computed by iteratively removing nodes of degree 1 (rake) and nodes of degree 2 (compress).

In the LOCAL model, one can 3-color any graph with a given H-decomposition as in Definition 3 with L layers in $O(L + \log^* n)$ rounds. Each layer induces a graph with maximum degree 2. First, use Linial's algorithm to color each layer in parallel with $C = O(1)$ colors. This coloring may contain lots of monochromatic edges between different layers and is only used as a schedule to compute the final 3-coloring. In order to compute that final coloring, iterate through the layers in a decreasing order, and in each layer iterate through the C colors. When processing one of the C color classes, every node picks one color in $\{1, 2, 3\}$ not used by any of its already colored neighbors (at most two).

Optimally, we would like to use the above LOCAL model algorithm as the base for our exponentially faster MPC algorithm. However, even if we were given an H-decomposition for free it is non-trivial to actually use it for 3-coloring a graph if the runtime is restricted to $O(\log\log n)$ rounds and you only allow for a polylogarithmic memory overhead. Going through the layers in some sequential manner would be way too slow as it would require logarithmically rounds. Still, as the LOCAL algorithm has locality $T = \Theta(\log n)$ the output of a node may depend on the topology in logarithmic distance. In order to achieve a fast MPC algorithm, we want the nodes to use the graph exponentiation technique to learn the part G_v of their T-hop neighborhood that is relevant to determine their output in $O(\log T) = O(\log\log n)$ rounds. We refer to G_v as the predecessor graph of node v. The challenge with the standard H-decomposition as given by Definition 3 is that, even though $G_v \subseteq V_{\geq \text{layer}(v)}$, it may be of size $\Theta(n)$. Hence, even if every node could learn its predecessor graph and store it in its local memory S_v (formally, the memory of every node is stored on some machine), the global space bound can only be upper bounded by $\sum_{v\in V} |G_v| = O(n^2)$, drastically, violating the desired near-linear bound. In order to circumvent this issue we introduce the concept of a strict H-decomposition which is optimized for its usage in the MPC model. The bottom line of this decomposition is that besides the properties of a classic H-decomposition, we also have a set V^{pivot}. The set V^{pivot} induces a graph with maximum degree 2 and hence can be colored with 3 colors in $O(\log^* n)$ rounds with Linial's algorithm [35]. The main gain compared to the classic H-decomposition is, that once we have colored the nodes in V^{pivot}, we can show that the predecessor graph of every node $v \in V \setminus V^{pivot}$ is of logarithmic size (when considering the same LOCAL model algorithm that colors these nodes layer by layer). Hence, each node can learn its predecessor graph in $O(\log\log n)$ rounds without violating global space constraints. We next present the definition of a strict H-decomposition.

▶ **Definition 4** ((Partial) Strict H-Decomposition). *Let F be a forest and* layer: $V(F) \mapsto \mathbb{N} \cup \{\infty\}$ *be a function. We define $V_{<\infty} = \{v \in V(F) \mid \text{layer}(v) < \infty\}$ and*

$$V^{pivot} := \{v \in V_{<\infty} \mid \text{layer}(v) \geq \text{layer}(w) \textit{ for every } w \in N_F(v)\}.$$

We refer to a (partial) H-decomposition layer *as* strict *if for every $v \in V_{<\infty} \setminus V^{pivot}$, it holds that*

$$|\{w \in N_F(v) \setminus V^{pivot} \mid \text{layer}(w) = \text{layer}(v)\} \cup \{w \in N_F(v) \mid \text{layer}(w) > \text{layer}(v)\}| \leq 1. \tag{1}$$

That is, the total number of non-pivot neighbors with the same layer and of neighbors with a strictly higher layer is at most 1.

There is some similarity between the definition of a strict H-decomposition and the H-decompositions used in the theory of so called locally checkable labelings [12, 10, 2]. These decompositions iteratively layer degree 1 nodes and paths of length at least ℓ. Our strict H-decomposition is similar to the case when we remove paths of length at least $\ell = 3$ (see Lemma 10). The following lemma is one of the most crucial structural properties of partial H-decompositions that we exploit in the core of our algorithm (see Appendix A).

▶ **Lemma 5** (Partial Strict H-Decomposition, Closure under taking minimums). *Let F be a forest and* $\text{layer}_1, \text{layer}_2 \colon V(F) \to \mathbb{N} \cup \{\infty\}$ *be two partial strict H-decompositions. Let* $\text{layer} \colon V(F) \to \mathbb{N} \cup \{\infty\}$ *with*

$$\text{layer}(v) = \min(\text{layer}_1(v), \text{layer}_2(v))$$

for every $v \in V(F)$. Then, layer *is also a partial strict H-decomposition.*

From a high level point of view, Lemma 5 says that we can independently compute two partial strict H-decompositions, and even though they might contain conflicting layer assignments for certain nodes, we can obtain a unified decomposition, by assigning each node to the smaller layer of the two choices. In fact, this insight also generalizes to more than two (possibly conflicting) decompositions. At the core of our procedure in Appendix A, many nodes (independently) learn large parts of the graph. Then, every node computes a partial decomposition on the parts that it has learned, and in a second step all these partial decompositions are combined, where each node takes the minimum layer that it got assigned in any of the decompositions. Taking the minimum is a very efficient procedure in the MPC model and only requires constant time. The remaining difficulty in Appendix A is to show that nodes learn large enough parts in the graph in order to make very fast global progress, that is, we show that the unified decomposition assigns a layer to a large fraction of the nodes.

4 Strict H-decomposition in MPC

In this section, we present our $O(\log\log n)$-round MPC algorithm for computing a strict H-decomposition. However, the hardest part of that algorithm, that is, assigning each node that is contained in a small subtree (see definition below) to a layer is deferred to Appendix A. The algorithm in this section uses $n \cdot \operatorname{poly}\log n$ global space. In Appendix B we explain how to extend the algorithm to optimal space.

High Level Overview. For the sake of this high level overview let us first assume that we compute an H-decomposition with $O(\log n)$ layers that may not be strict. Similar to the classic rake & compress algorithm, our algorithm iteratively assigns nodes to layers. After assigning a node to some layer we *remove* it from the graph and continue on the remaining graph, which may actually become disconnected and turn into a forest. In order to present the details of the high level intuition we require the definition of a subtree which is central to our whole approach.

▶ **Definition 6** (Subtree). *Let T be a tree. A* subtree *$T' \subseteq T$ is a connected induced subgraph of T such that $T \setminus T'$ contains at most one component.*

A subtree $T' \subseteq F$ of a forest F is a connected induced subgraph of F such that the number of components of $F \setminus T'$ is not larger than the number of connected components of F.

The definition of a subtree is best understood in a rooted tree, where the subtree rooted at a node v is formed by all its descendants.

In order to assign a layer to all nodes of the graph, we iterate the following two steps until all nodes have received a layer:

1. Assigns a layer to each node contained in a *small subtree* of size $\leq n^{\delta/10}$ (SubTreeRC(F)),
2. Assign a layer to each node of degree ≤ 2 in the remaining graph.

This process can be seen as a generalization of the classic rake and compress procedure, in which one iteratively removes leaves, i.e., subtrees of size 1, and nodes of degree 2. The rake and compress procedure requires $O(\log n)$ iterations to *remove* all nodes of the graph. Our generalized process requires $O(1/\delta) = O(1)$ in order to assign a layer to every node of the graph (see Lemma 7 for $x = n^{\delta/10}$). Note that the lemma statement considers a slightly different process than the one presented in this overview; the difference lies in the fact that we actually want to compute a strict H-decomposition. However, a similar lemma holds for the process of this overview. The main contribution and the main difficulty of our work lies in the procedure SubTreeRC(F) as nodes do not know whether they are contained in a small subtree, but still these subtrees can have diameter up to $n^{\delta/10}$, so conditioned on the 1 vs 2 cycle conjecture it is impossible that a single node can learn the whole subtree in $O(\log\log n)$ rounds (we don't prove this formally, but it's very unlikely that such a result holds without breaking the conjecture). We explain the details of the procedure SubTreeRC(F) in Appendix A.

We continue with our generalized rake and compress statement that shows that a constant number of iterations of the aforementioned process suffice. As we want to compute a strict H-decomposition (see Definition 4), we need to slightly modify Step 2 of the above outline, for which we require further definitions; the details of why SubTreeRC(F) returns layers that induce a strict H-decomposition are presented in Appendix A.

A path in a graph is a *degree-2 path* if all of its nodes, including its endpoints have degree 2. The *length of a path* is the number of nodes in the path, e.g., a single node is a path of length 1.

The following lemma is easiest to be understood when setting $x = \ell = 1$ where the process (almost) equals the classic rake & compress process – in fact it consists of a rake step, a compress step, and another rake step – and the theorem shows that it removes 1/3 of the nodes (2/3 of the nodes remain in the graph).

▶ **Lemma 7** (Generalized rake and compress). *Let $x, \ell \in \mathbb{Z}$. Consider a process on a tree T that consists of the following steps:*

1. *Remove (at least) all subtrees of size $\leq x$ from T, resulting in T_1,*
2. *Remove (at least) all nodes contained in a degree-2 path of length at least ℓ from T_1, resulting in T_2,*
3. *Remove (at least) all nodes with degree ≤ 1 from T_2.*

The number of nodes remaining is at most a $1/(1 + (x + 1)/2\ell) = O(\ell/x)$ fraction of the nodes from T. The degrees of nodes in Step 2) and 3) of the process are with respect to the graph induced by remaining nodes at the respective step.

We now state a lemma for a key subroutine that we will use as black box in this section and dedicate Appendix A to designing an algorithm that proves the lemma.

▶ **Lemma 8** (SubTreeRC). *Let F be a forest on n vertices. There exists a deterministic MPC algorithm SubTreeRC with $O(n^{\delta})$ local space, $0 < \delta < 1$, and $\widetilde{O}(n)$ global space which takes F as input and computes in $O(\log\log n)$ rounds a partial strict H-decomposition* layer: $V(F) \mapsto [\lceil \log(|V(F)| + 1) \rceil] \cup \{\infty\}$ *such that* layer$(v) < \infty$ *for every node $v \in V(F)$ contained in a subtree of size $n^{\delta/10}$.*

Our MPC algorithm for computing strict H-decomposition appears in Algorithm 1. We will now prove the correctness and progress guarantees of our algorithm.

▶ **Lemma 9.** *At the end of each iteration i, we have that* layer *is a partial strict H-decomposition with at most $(i + 1) \cdot$* offset *layers.*

▶ **Lemma 10.** *In iteration i, Algorithm 1 correspond to a generalized rake and compress step with $x = n^{\delta/10}$ and $\ell = 3$.*

▶ **Corollary 11.** *In iteration i, Algorithm 1 correspond to a generalized rake and compress step with $x = 1$ and $\ell = 3$.*

Algorithm 1 Strict H-decomposition.

```
1: Throughout $V_\infty = \{v \in F \mid \text{layer}(v) = \infty\}$ denotes the set of nodes whose layer equals $\infty$.
2: function StrictHDecomp(Forest $F$)
3:     Initialize: $\text{layer}(v) = \infty$ for all $v \in V(F)$; $\mathsf{offset} \leftarrow \lceil \log(|V(F)|+1) \rceil + 1$
4:     for $i = 1, 2, \ldots, \lceil 10/\delta \rceil$ do
5:         $F_i \leftarrow F[V_\infty]$
6:         $\text{layer} \leftarrow i \cdot \mathsf{offset} + \text{SubTreeRC}(F_i, x = n^{\delta/10})$
7:         Let $V_i^{pivot} \leftarrow \{v \in V_\infty \mid d_{V_\infty}(v) \le 2, d_{V_\infty}(w) \le 2$ for all $w \in N(v)\}$
8:         $\text{layer}(v) \leftarrow (i+1) \cdot \mathsf{offset}$ for every node $v \in V_i^{pivot}$
9:         $\text{layer}(v) \leftarrow (i+1) \cdot \mathsf{offset}$ for every node $v \in V_\infty$ with $\le 1$ in $V_\infty$
10:    return layer
```

▶ **Theorem 12.** *Algorithm* StrictHDecomp(F) *(Algorithm 1) applied to some forest F computes a strict H-decomposition of F with $O(\log n)$ layers, uses $O(\log \log n)$ low-space MPC rounds and $\widetilde{O}(n)$ global space.*

Proof. By Lemmas 7 and 10, in each iteration, the number of nodes in the forest shrinks by a factor of $O(n^{\delta/10})$. Therefore, after $O(1/\delta)$ iterations of the for loop, the number of nodes with layer ∞ will be zero.

By Lemma 9, in an iteration i, we produce a partial strict H-decomposition with at most $(i+1) \cdot \mathsf{offset}$ layers and in the next iterations $j > i$, we compute a partial strict H-decomposition of the nodes that received layer ∞ (V_∞) in iteration i. The offset value ensures that the nodes in V_∞ get a higher layer than the nodes in $V \setminus V_\infty$. After $i = O(1/\delta)$ iterations, each node has a layer at most $O(\log n)$ since $(i+1) \cdot \mathsf{offset} = O(\log n)$, and hence we produce a valid strict H-decomposition.

Lemma 8 ensures that implementing each iteration takes $O(\log \log n)$ low-space MPC rounds and $\widetilde{O}(n)$ global space. The theorem follows because there are just $O(1/\delta)$ iterations. ◀

5 Coloring, MIS, and Matching

The following theorem is proven at the end of the section.

▶ **Theorem 13.** *There is a deterministic $O(\log \log n)$ round algorithm for 3-coloring trees in the low-space MPC model using $\widetilde{O}(n)$ words of global space.*

For an input tree F, consider having a strict H-decomposition layer : $V(F) \to \mathbb{N}$ described in Definition 4, which we get from Algorithm 1 in $O(\log \log n)$ rounds and $\widetilde{O}(n)$ words of global space. We first color the subgraph induced by the nodes in V^{pivot}. Recall that V^{pivot} is the set of nodes that have no neighbor with a higher layer.

Coloring the Pivot Nodes. The subgraph $F[V^{pivot}]$ has maximum degree 2 each node $v \in V^{pivot}$ has at most two neighbors in V^{pivot} with the same layer, and no neighbors with higher layer. In order to color $F[V^{pivot}]$, we first run Linial's $O(\Delta^2)$-coloring algorithm [34], which requires $O(\log^* n)$ rounds. Since $\Delta(F[V^{pivot}]) \le 2$, Linial's algorithm results in an $O(1)$-coloring which we can convert to a 3-coloring by performing the following: In each round, all nodes with the highest color among their neighbors in V^{pivot} recolor themselves with the smallest color such that a proper coloring is preserved. Clearly, one color is eliminated in each round and since each node v has at most 2 neighbors in $F[V^{pivot}]$, we achieve a 3-coloring of $F[V^{pivot}]$ in a constant number of rounds.

Coloring the Remaining Nodes. We will now compute a 3-coloring of the nodes in $V \setminus V^{pivot}$. We first orient all edges $e = \{u, v\}$ with $u, v \in V \setminus V^{pivot}$ from u to v if $\text{layer}(u) < \text{layer}(v)$ and arbitrarily if $\text{layer}(u) = \text{layer}(v)$. The following lemma will help us to ensure that we do not create conflicts with the 3-coloring computed on V^{pivot}.

▶ **Lemma 14.** *Each node in $v \in V \setminus V^{pivot}$ has at most two forbidden colors. If v has an outgoing edge, then it can have at most one forbidden color.*

Proof. Each node in $v \in V \setminus V^{pivot}$ has at most two neighbors in V^{pivot}. This is because nodes in V^{pivot} do not have neighbors in higher layer, so v can only have neighbors in V^{pivot} at the same or higher layer. By Definition 4, v can have at most two such neighbors.

Nodes v with one outgoing edge can have at most one neighbor in V^{pivot}, as otherwise v has three neighbors with same or higher layer, and Definition 4 is violated. So if v has an outgoing edge, it can have at most one forbidden color. ◀

In what follows, each node $v \in V \setminus V^{pivot}$ will remember its at most two forbidden colors due to neighbors in V^{pivot}. Definition 4 also guarantees that all nodes in $V \setminus V^{pivot}$ will have at most one outgoing edge. So the nodes $w \in V \setminus V^{pivot}$ with no outgoing edge pick an arbitrary color that is not forbidden as their final color.

In order to properly color the nodes with exactly one outgoing edge, consider the following centralized procedure: Color the nodes one by one in a greedy manner starting from the highest layer and with an arbitrary order within one layer. Here, greedy means, that a node picks the smallest color that is not forbidden and not used by any of its already colored neighbors. This process computes a proper 3-coloring as each node will have one color used by the neighbor along its outgoing edge, and at most one forbidden color. The output of a node $v \in V \setminus V^{pivot}$ in this centralized procedure only depends on the *directed path of v* obtained by following outgoing edges starting at v. In the following lemma we show that this directed path cannot be too long.

▶ **Lemma 15.** *The directed path of a node $v \in V \setminus V^{pivot}$ obtained by following outgoing edges starting at v has length at most $O(\log n)$.*

Proof. Consider a directed edge (u, w) in the directed path of v. If $\text{layer}(u) = \text{layer}(w)$, then w cannot have an outgoing edge as it will have two neighbors in $V \setminus V^{pivot}$ with same or higher layer, violating Definition 4. In other words, if $\text{layer}(u) = \text{layer}(w)$, then the directed path of v ends at w.

Therefore, if we go along the directed path, the layer of the nodes either strictly increases or the path does not continue. Since there are $O(\log n)$ layers in the H-decomposition, the length of a directed path is at most $O(\log n)$. ◀

In our MPC algorithm, the idea is for each node to learn its $O(\log n)$ length directed path by performing graph exponentiation only along the directed edges. Since all nodes with no outgoing edges are already colored with their final color, consider performing the following MPC algorithm only for nodes with one outgoing edge: Each node computes its final color after gathering its directed path by performing $O(\log \log n)$ graph exponentiation steps along directed edges.

Proof of Theorem 13. Correctness follows from the fact that each node can recolor itself with its final color when seeing its whole directed path. The runtime follows from the fact that we only perform $O(\log\log n)$ graph exponentiation steps and color the directed paths.

Let us analyze the space usage of our algorithm. Since the length of a directed path stored by each node during the algorithm is at most $O(\log n)$, we do not violate global space. Note that the sequential coloring of frozen layers does not require additional space. Notice that even though each node v is the source of at most one request, multiple nodes may send a request to v. Hence, during graph exponentiation, node v may have to communicate with a large number of nodes in lower layers. To mitigate this issue, we perform a load balancing process by sorting all the at most n requests by the ID of their destination. This can be done deterministically in $O(1)$ rounds. Now, all the requests with destination v lie in consecutive machines, and therefore, we can broadcast the response of v to all these machines in $O(1)$ rounds by creating a constant depth broadcast tree on these machines. Therefore, each step of graph exponentiation can be done in $O(1)$ rounds, which leads to an overall running time of $O(\log\log n)$ rounds. ◀

MIS and Maximal Matching

The maximal independent set and maximal matching algorithms follow from Theorem 13.

▶ **Theorem 16.** *There is a deterministic $O(\log\log n)$ round MIS algorithm for trees in the low-space MPC model using $\widetilde{O}(n)$ words of global space.*

Proof. By Theorem 13, we can color the tree with 3 colors. For all colors i, perform the following. Nodes colored i add themselves to the independent set, and all nodes adjacent to nodes colored i remove themselves from the graph. Clearly this results in a maximal independent set in $O(1)$ rounds and the space requirements are satisfied. ◀

▶ **Theorem 17.** *There is a deterministic $O(\log\log n)$ round maximal matching algorithm for trees in the low-space MPC model using $\widetilde{O}(n)$ words of global space.*

Proof. By Theorem 13, we can color the tree with 3 colors using a H-decomposition. Recall that in the decomposition, each node v with $\text{layer}(v) = i$ has at most two neighbors with layer at least i. Let us define the parent nodes of v. We orient an edge $\{u, v\}$ from v to u if (i) u belongs to a strictly higher layer than v or (ii) u belongs to the same layer and has a higher ID. For all colors i, perform the following. Node v colored i proposes to its highest ID outgoing neighbor u, and u accepts the proposal of the highest ID proposer. If u accepts v's proposal in which case the edge $\{u, v\}$ joins the matching. If u rejects v's proposal, it means that u is matched with some other node and then we repeat the same procedure with v's other possible out-neighbor. Note that when a node joins the matching, it prevents all other incident edges from joining the matching. As a result, all nodes colored i have either joined the matching or they have no out-going edges. After iterating through all color classes, all nodes have either joined the matching or they have no incident edges, implying that all their original neighbors belong to the matching. This results in a maximal matching in $O(1)$ rounds and the space requirements are satisfied. ◀

References

1 Noga Alon, Lásló Babai, and Alon Itai. A Fast and Simple Randomized Parallel Algorithm for the Maximal Independent Set Problem. *Journal of Algorithms*, 7(4):567–583, 1986. doi:10.1016/0196-6774(86)90019-2.

2 Alkida Balliu, Keren Censor-Hillel, Yannic Maus, Dennis Olivetti, and Jukka Suomela. Locally Checkable Labelings with Small Messages. In *the Proceedings of the International Symposium on Distributed Computing (DISC)*, pages 8:1–8:18, 2021. doi:10.4230/LIPIcs.DISC.2021.8.

3 Alkida Balliu, Rustam Latypov, Yannic Maus, Dennis Olivetti, and Jara Uitto. Optimal Deterministic Massively Parallel Connectivity on Forests. In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2589–2631, 2023. doi:10.1137/1.9781611977554.ch99.

4 Philipp Bamberger, Fabian Kuhn, and Yannic Maus. Efficient Deterministic Distributed Coloring with Small Bandwidth. In *PODC '20: ACM Symposium on Principles of Distributed Computing (PODC)*, pages 243–252, 2020. doi:10.1145/3382734.3404504.

5 Leonid Barenboim and Michael Elkin. Sublogarithmic distributed MIS algorithm for sparse graphs using nash-williams decomposition. *Distributed Comput.*, 22(5-6):363–379, 2010. doi:10.1007/s00446-009-0088-2.

6 Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The Locality of Distributed Symmetry Breaking. *Journal of the ACM*, 63(3):20:1–20:45, 2016.

7 Soheil Behnezhad, Sebastian Brandt, Mahsa Derakhshan, Manuela Fischer, MohammadTaghi Hajiaghayi, Richard M. Karp, and Jara Uitto. Massively parallel computation of matching and mis in sparse graphs. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, pages 481–490, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3293611.3331609.

8 Soheil Behnezhad, Laxman Dhulipala, Hossein Esfandiari, Jakub Łącki, and Vahab Mirrokni. Near-Optimal Massively Parallel Graph Connectivity. In *FOCS*, 2019. doi:10.1109/FOCS.2019.00095.

9 Sebastian Brandt, Manuela Fischer, and Jara Uitto. Breaking the Linear-memory Barrier in MPC: Fast MIS on Trees with Strongly Sublinear Memory. *Theoretical Computer Science*, 849:22–34, 2021. doi:10.1016/j.tcs.2020.10.007.

10 Yi-Jun Chang. The Complexity Landscape of Distributed Locally Checkable Problems on Trees. In *DISC*, pages 18:1–18:17, 2020. doi:10.4230/LIPIcs.DISC.2020.18.

11 Yi-Jun Chang, Manuela Fischer, Mohsen Ghaffari, Jara Uitto, and Yufan Zheng. The Complexity of $(\Delta + 1)$-Coloring in Congested Clique, Massively Parallel Computation, and Centralized Local Computation. In *PODC*, 2019. doi:10.1145/3293611.3331607.

12 Yi-Jun Chang and Seth Pettie. A Time Hierarchy Theorem for the LOCAL Model. *SIAM J. Comput.*, 48(1):33–69, 2019. doi:10.1137/17M1157957.

13 Sam Coy and Artur Czumaj. Deterministic Massively Parallel Connectivity. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, 2022. doi:10.1145/3519935.3520055.

14 Artur Czumaj, Peter Davies, and Merav Parter. Graph Sparsification for Derandomizing Massively Parallel Computation with Low Space. In *the Proceedings of the Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 175–185, 2020. doi:10.1145/3350755.3400282.

15 Artur Czumaj, Peter Davies, and Merav Parter. Component Stability in Low-Space Massively Parallel Computation. In *PODC*, 2021. doi:10.1145/3465084.3467903.

16 Artur Czumaj, Peter Davies, and Merav Parter. Improved Deterministic $(\Delta + 1)$-Coloring in Low-Space MPC. In *PODC*, pages 469–479, 2021. doi:10.1145/3465084.3467937.

17 Artur Czumaj, Peter Davies, and Merav Parter. Simple, deterministic, constant-round coloring in congested clique and MPC. *SIAM Journal on Computing*, 50(5):1603–1626, 2021. doi:10.1137/20M1366502.

18 Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, pages 107–113, 2008. doi:10.1145/1327452.1327492.

19 Mohsen Ghaffari. An Improved Distributed Algorithm for Maximal Independent Set. In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 270–277, 2016.

20 Mohsen Ghaffari, Themis Gouleakis, Christian Konrad, Slobodan Mitrovic, and Ronitt Rubinfeld. Improved Massively Parallel Computation Algorithms for MIS, Matching, and Vertex Cover. In *PODC*, pages 129–138, 2018. doi:10.1145/3212734.3212743.

21 Mohsen Ghaffari, Christoph Grunau, Bernhard Haeupler, Saeed Ilchi, and Václav Rozhoň. Improved Distributed Network Decomposition, Hitting Sets, and Spanners, via Derandomization. In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2532–2566, 2023. doi:10.1137/1.9781611977554.ch97.

22 Mohsen Ghaffari, Christoph Grunau, and Ce Jin. Improved MPC Algorithms for MIS, Matching, and Coloring on Trees and Beyond. In *DISC*, 2020. doi:10.4230/LIPIcs.DISC.2020.34.

23 Mohsen Ghaffari, Fabian Kuhn, and Jara Uitto. Conditional Hardness Results for Massively Parallel Computation from Distributed Lower Bounds. In *FOCS*, pages 1650–1663, 2019. doi:10.1109/FOCS.2019.00097.

24 Mohsen Ghaffari and Ali Sayyadi. Distributed Arboricity-Dependent Graph Coloring via All-to-All Communication. In *ICALP*, pages 142:1–142:14, 2019. doi:10.4230/LIPIcs.ICALP.2019.142.

25 Mohsen Ghaffari and Jara Uitto. Sparsifying Distributed Algorithms with Ramifications in Massively Parallel Computation and Centralized Local Computation. In *SODA*, 2019. doi:10.1137/1.9781611975482.99.

26 Jeff Giliberti, Manuela Fischer, and Christoph Grunau. Deterministic massively parallel symmetry breaking for sparse graphs. *CoRR*, abs/2301.11205, 2023. doi:10.48550/arXiv.2301.11205.

27 Michael T. Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, searching, and simulation in the mapreduce framework. In Takao Asano, Shin-ichi Nakano, Yoshio Okamoto, and Osamu Watanabe, editors, *Algorithms and Computation*, pages 374–383, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

28 Chetan Gupta, Rustam Latypov, Yannic Maus, Shreyas Pai, Simo Särkkä, Jan Studený, Jukka Suomela, Jara Uitto, and Hossein Vahidi. Fast dynamic programming in trees in the mpc model, 2023. arXiv:2305.03693.

29 Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. *ACM SIGOPS Operating Systems Review*, pages 59–72, 2007. doi:10.1145/1272996.1273005.

30 Howard J. Karloff, Siddharth Suri, and Sergei Vassilvitskii. A Model of Computation for MapReduce. In *SODA*, 2010. doi:10.1137/1.9781611973075.76.

31 Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. Filtering: A Method for Solving Graph Problems in MapReduce. In *SPAA*, pages 85–94, 2011. doi:10.1145/1989493.1989505.

32 Rustam Latypov and Jara Uitto. Deterministic 3-coloring of trees in the sublinear MPC model. *CoRR*, abs/2105.13980, 2021. arXiv:2105.13980.

33 Christoph Lenzen and Roger Wattenhofer. Brief Announcement: Exponential Speed-Up of Local Algorithms Using Non-Local Communication. In *PODC*, 2010. doi:10.1145/1835698.1835772.

34 Nathan Linial. Distributive Graph Algorithms – Global Solutions from Local Data. In *FOCS*, 1987. doi:10.1109/SFCS.1987.20.

35 Nathan Linial. Locality in Distributed Graph Algorithms. *SIAM J. Comput.*, 21(1):193–201, 1992. doi:10.1137/0221015.

36 Michael Luby. A Simple Parallel Algorithm for the Maximal Independent Set Problem. *SIAM Journal on Computing*, 15:1036–1053, 1986. doi:10.1137/0215074.

37 Crispin Nash-Williams. Decomposition of Finite Graphs Into Forests. *Journal of the London Mathematical Society*, s1-39:12, 1964. doi:10.1112/jlms/s1-39.1.12.

38 Öjvind Johansson. Simple Distributed $\Delta + 1$-coloring of Graphs. *Information Processing Letters*, pages 229–232, 1999. doi:10.1016/S0020-0190(99)00064-2.

39 Merav Parter. $(\delta + 1)$ coloring in the congested clique model. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *International Colloquium on Automata, Languages, and Programming, (ICALP)*, volume 107, pages 160:1–160:14, 2018. doi:10.4230/LIPIcs.ICALP.2018.160.

40 Merav Parter and Hsin-Hao Su. Randomized $(\delta + 1)$-coloring in $o(\log^* \delta)$ congested clique rounds. In *32nd International Symposium on Distributed Computing (DISC)*, volume 121, pages 39:1–39:18, 2018. doi:10.4230/LIPIcs.DISC.2018.39.

41 Tim Roughgarden, Sergei Vassilvitskii, and Joshua R. Wang. Shuffles and Circuits (On Lower Bounds for Modern Parallel Computation). *Journal of the ACM*, 2018. doi:10.1145/3232536.

42 Václav Rozhoň and Mohsen Ghaffari. Polylogarithmic-time deterministic network decomposition and distributed derandomization. In *STOC*, pages 350–363, 2020. doi:10.1145/3357713.3384298.

43 Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2009.

44 Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *the Proceedings of the SENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, 2010. doi:10.5555/1863103.1863113.

A Massively Parallel Subtree Rake and Compress

This section is dedicated to designing an algorithm that proves Lemma 8, which states that we can compute in $O(\log \log n)$ rounds a partial strict H-decomposition that assigns each node contained in a subtree of size $O(n^{\delta/10})$ to one of $O(\log n)$ layers. We restate the lemma.

▶ **Lemma 8** (SubTreeRC). *Let F be a forest on n vertices. There exists a deterministic MPC algorithm SubTreeRC with $O(n^{\delta})$ local space, $0 < \delta < 1$, and $\widetilde{O}(n)$ global space which takes F as input and computes in $O(\log \log n)$ rounds a partial strict H-decomposition* $\mathrm{layer}: V(F) \mapsto [\lceil \log(|V(F)| + 1) \rceil] \cup \{\infty\}$ *such that* $\mathrm{layer}(v) < \infty$ *for every node $v \in V(F)$ contained in a subtree of size $n^{\delta/10}$.*

Our algorithm critically relies on the balanced graph exponentiation technique mentioned in Section 1.2 and explained in detail in the full version. In the following definition, you should think about U as being the set of nodes that v has stored in its local memory after the balanced graph exponentiation. We refer to U as good if it contains all nodes within distance $O(\log n)$ of v, except for potentially one direction, for which no node is contained in U.

▶ **Definition 18** (U is a good subset for v). *Let F be a forest, $U \subseteq V(F)$ and $v \in V(F)$. We say that U is a good subset for v if*

1. $v \in U$,
2. $|N_F(v) \setminus U| \leq 1$, *i.e., v has at most one neighbor in F not in U,*
3. $N_F(w) \subseteq U$ *for every $w \in U \setminus \{v\}$ with $d_F(v, w) \leq 3L$ where $L := \lceil \log(|U| + 1) \rceil$.*

In Appendix A.1, we give a peeling algorithm that takes as input a set U and computes a partial strict H-decomposition with $O(\log n)$ layers by repeatedly peeling off low-degree vertices contained in U. Moreover, if U is good for v, then v gets assigned to one of the $O(\log n)$ layers. This peeling algorithm will later be simulated without any further communication on the machine that has stored the set U in its memory.

Using the balanced graph exponentiation technique, we can compute a collection of sets $U_1, U_2, \ldots, U_k$ such that for each node v contained in a subtree of size at most $n^{\delta/10}$ there exists some subset U_j that is good for v. In particular, in the full version, we prove the following statement.

▶ **Lemma 19** (Lemma from Balanced Exponentiation). *Let F be a forest on n vertices. There exists a deterministic low-space MPC algorithm with $O(n^\delta)$ local space, $0 < \delta < 1$, and $O(n \cdot \mathrm{poly}(\log n))$ global space which takes F as input and computes in $O(\log\log n)$ rounds a collection of non-empty sets $U_1, U_2, \ldots, U_k \subseteq V(F)$ such that*

1. *(Local Space) $|U_j| = O(n^\delta)$ for every $j \in [k]$,*
2. *(Global Space) $\sum_{j=1}^{k} |U_j| = O(n \cdot \mathrm{poly}(\log n))$ and*
3. *for every $v \in V(F)$ which is contained in a subtree of size at most $n^{\delta/10}$, there exists a $j \in [k]$ such that U_j is a good subset for v (see Definition 18).*

Moreover, the algorithm also computes for each U_j the forest $F[U_j]$ induced by vertices in U_j and stores it on a single machine.

Our final MPC algorithm for proving Lemma 8 first computes a collection of sets $U_1, U_2, \ldots, U_k$ using Lemma 19. Then, the machine storing U_j locally simulates the peeling algorithm of Appendix A.1 with input U_j. As a result, we obtain one partial strict H-decomposition for each set U_j. These partial strict H-decompositions are then combined into one partial strict H-decomposition by assigning each node to the smallest layer assigned by any of the partial strict H-decompositions. More details can be found in Appendix A.2.

A.1 The Conservative Peeling Algorithm

Algorithm 2 computes a partial strict H-decomposition by repeatedly removing low-degree vertices contained in U.

Algorithm 2 Conservative Peeling Algorithm.

```
1:  function ConservativePeeling(Forest $F$, Subset $U \subseteq V(F)$)
2:      $V_{\geq 1} \leftarrow V(F)$, layer: $V(F) \mapsto \mathbb{N} \cup \{\infty\}$, $L \leftarrow \lceil \log(|U|+1) \rceil$
3:      for $i = 1, 2, \ldots, L$ do
4:          We define $N_{\geq i}(v) := N_{F[V_{\geq i}]}(v)$ for every $v \in V_{\geq i}$.
5:          $V_i^{pivot} \leftarrow \{v \in V_{\geq i} \cap U \mid N_{\geq i}(v) \subseteq U$ and $\forall\, w \in N_{\geq i}(v) \cup \{v\}$: $|N_{\geq i}(w)| \leq 2\}$.
6:          $V_i \leftarrow V_i^{pivot} \cup \{v \in V_{\geq i} \cap U \mid |N_{\geq i}(v) \setminus V_i^{pivot}| \leq 1\}$
7:          $V_{\geq i+1} \leftarrow V_{\geq i} \setminus V_i$
8:          layer$(v) \leftarrow i$ for every $v \in V_i$
9:      layer$(v) \leftarrow \infty$ for every $v \in V_{\geq L+1}$
10:     return layer
```

If we would just be interested in computing a partial H-decomposition instead of a strict one, then we could replace Algorithm 2 with the single line $V_i \leftarrow \{v \in V_{\geq i} \cap U \mid |N_{\geq i}(v)| \leq 2\}$.

We first show that Algorithm 2 indeed computes a partial strict H-decomposition.

▶ **Lemma 20.** *Let F be a forest and $U \subseteq V(F)$. Let* layer: $V(F) \mapsto \mathbb{N} \cup \{\infty\}$ *be the mapping computed by Algorithm 2 when given F and U as input. Then,* layer *is a partial strict H-decomposition as defined in Definition 4.*

Next, we show that if U is a good subset for v, as defined in Definition 18, then v gets assigned to one of the $O(\log n)$ layers.

▶ **Lemma 21.** *Let F be a forest, $U \subseteq V(F)$ and $v \in V(F)$. Let* $\text{layer}\colon V(F) \mapsto \mathbb{N} \cup \{\infty\}$ *be the mapping computed by Algorithm 2 when given F and U as input. If U is a good subset for v (see Definition 18), then* $\text{layer}(v) \leq L := \lceil \log(|U|+1) \rceil$.

Finally, we show that we can locally simulate Algorithm 2 by only knowing the forest induced by vertices in U and the degree of each node U in the original forest.

▶ **Lemma 22** (Local Sequential Simulation)**.** *Let F be an arbitrary forest and $U \subseteq V(F)$ be a non-empty subset. Let* $\text{layer}\colon V(F) \mapsto \mathbb{N} \cup \{\infty\}$ *be the mapping computed by Algorithm 2 when given F and U as input. There exists a sequential algorithm running in $O(|U|)$ space with the following guarantee: The input of the algorithm is the forest $F[U]$ and the degree $\deg_F(u)$ of each node $u \in U$ in the forest F. The algorithm outputs for each node $v \in U$ its layer* $\text{layer}(v)$.

A.2 Subtree Rake and Compress

Algorithm 3 computes a partial strict H-decomposition with $O(\log n)$ layers where each node in a subtree of size at most x is assigned to one of the layers. We later set $x = n^{\delta/10}$. The correctness follows from the key structural property that partial strict H-decompositions are closed under taking minimums (Lemma 5).

Algorithm 3 SubTreeRC Algorithm.

```
1: function SubTreeRC(forest $F$, $x \in \mathbb{N}$)
2:     Let $U_1, U_2, U_3, \ldots, U_k \subseteq V(F)$ such that for every node $v \in V(F)$ contained in a
       subtree of size at most $x$ in $F$, there exists some $j \in [k]$ such that $U_j$ is a good subset for
       $v$ (see Definition 18)
3:     $\text{layer}_j \leftarrow \mathit{ConservativePeeling}(F, U_j)$ for every $j \in [k]$   ▷ $\text{layer}_j\colon V(F) \mapsto \mathbb{N} \cup \{\infty\}$
4:     $\text{layer}(v) = \min_{j \in [k]} \text{layer}_j(v)$                              ▷ $\text{layer}\colon V(F) \mapsto \mathbb{N} \cup \{\infty\}$
5:     return layer
```

▶ **Lemma 23.** *The algorithm above computes a partial H decomposition* $\text{layer}\colon V(F) \mapsto [\lceil \log(|V(F)|+1) \rceil] \cup \{\infty\}$ *such that* $\text{layer}(v) < \infty$ *for every node $v \in V(F)$ contained in a subtree of size at most x.*

Proof. Lemma 20 states that layer_j is a strict partial H-decomposition for every $j \in [k]$. Hence, Lemma 5 implies that layer is also a strict H-decomposition. Moreover, for every node $v \in V(F)$ contained in a subtree of size at most x in F, there exists some $j \in [k]$ such that U_j is a good subset for v. Thus, Lemma 21 gives that $\text{layer}_j(v) < \infty$ and therefore $\text{layer}(v) < \infty$. ◀

We are now ready to prove Lemma 8.

Proof of Lemma 8. We first run the balanced exponentiation algorithm of Lemma 19 which runs in $O(\log \log n)$ rounds and needs $\widetilde{O}(n)$ global space. As a result, we obtain a collection of non-empty subsets $U_1, U_2, \ldots, U_k \subseteq V(F)$ satisfying the three properties stated in Lemma 19. In particular, for each $j \in [k]$, there exists one machine which has stored $F[U_j]$. As $|U_j| = O(n^\delta)$ and F is a forest, $F[U_j]$ indeed fits into one machine. Moreover, one can compute in $O(1)$ rounds for each node $v \in V(F)$ its degree $\deg_F(v)$ and store $\deg_F(v)$ for every node $v \in U_j$ in the same machine as we store $F[U_j]$ using standard MPC primitives [27]. Let $\text{layer}_j \leftarrow \mathit{ConservativePeeling}(F, U_j)$. Lemma 22 implies that we can compute

$\text{layer}_j(u)$ for every node $u \in U_j$ locally on the machine that stores $F[U_j]$ without any further communication. Then, in $O(1)$ rounds we can compute $\text{layer}(v) = \min_{j\in[k]} \text{layer}_j(v)$ for every $v \in V$ using the fact that we can sort N items in $O(1)$ rounds in the low-space MPC model with $\widetilde{O}(N)$ global space [27]. In more detail, we create one tuple $(v, \text{layer}_j(v))$ for every $j \in [k]$ and $u \in U_j$ and one tuple (v, ∞) for every node $v \in V(F)$. Then, we sort the tuples according to the lexicographic order. Given the sorted tuples, it is straightforward to determine $\text{layer}(v)$ for every $v \in V$. As $\sum_{j=1}^{k} |U_j| = \widetilde{O}(n)$, it follows that the algorithm needs $\widetilde{O}(n)$ global space. It thus remains to argue about the correctness, which directly follows from the third property of Lemma 19 and Lemma 23. ◀

B Coloring, MIS, Matching, and H-decomposition with Optimal Space

In this section we show how to obtain optimal global space by equipping the algorithm from Theorems 13, 16, and 17 with suitable pre- and processing steps that free additional space.

▶ **Theorem 1.** *There are deterministic $O(\log\log n)$-round low-space MPC algorithms for 3-coloring, maximal matching and maximal independent set (MIS) on forests. These algorithms use $O(n)$ global space.*

Proof. We perform the standard procedure of iteratively putting in layer i nodes of degree at most 2 for $i = 1$ to $O(\log\log n)$. This removes $O(\text{poly}\log n)$ fraction of the nodes since each iteration layers a constant fraction of the nodes. Therefore, the new number of nodes is $n' = n/\text{poly}\log n$, and an MPC algorithm using $\widetilde{O}(n')$ global space uses $O(n)$ words of global space.

So we freeze these initial $O(\log\log n)$ layers obtain G' remaining graph with $n' = n/\text{poly}\log n$ nodes. Then we apply Theorem 13 to compute a 3-coloring in G' in $O(\log\log n)$ rounds and $O(n)$ global space. Finally we complete the solution on the nodes in the frozen layers one layer at a time taking an additional $O(\log\log n)$ rounds.

The claim for MIS and maximal matching follows by the proofs of Theorem 16 and Theorem 17 respectively after computing the H-decomposition and the 3-coloring. ◀

Using a similar preprocessing step, we can also show that a strict H-decomposition of Theorem 12 can be computed with optimal global space.

▶ **Theorem 2.** *There is a deterministic $O(\log\log n)$-rounds low-space MPC algorithm that computes a strict H-decomposition with $O(\log n)$ layers on forests in $O(n)$ global space.*

Proof. Same as above, iteratively putting in layer i the pivot nodes and nodes of degree 1 as in Algorithm 1 of Algorithm 1 for $i = 1$ to $O(\log\log n)$. By Corollary 11 and using Lemma 7 with $x = 1$ and $\ell = 3$, we get that each iteration layers a constant fraction of nodes, which implies that $O(\text{poly}\log n)$ fraction of the nodes are removed after $O(\log\log n)$ iterations.

Now we have a partial strict H-decomposition if we assign layer ∞ to the remaining nodes. These nodes form a graph G' with $n' = n/\text{poly}\log n$ nodes, and so we can compute a strict H-decomposition on G' using Algorithm 1 in $O(\log\log n)$ rounds and $\widetilde{O}(n') = O(n)$ global space. Therefore, we have computed a strict H-decomposition of G in $O(\log\log n)$ rounds and in $O(n)$ global space. ◀

On the Inherent Anonymity of Gossiping

Rachid Guerraoui
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Anne-Marie Kermarrec
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Anastasiia Kucherenko
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Rafael Pinot
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Sasha Voitovych[1]
University of Toronto, Canada

Abstract

Detecting the *source of a gossip* is a critical issue, related to identifying *patient zero* in an epidemic, or the *origin of a rumor* in a social network. Although it is widely acknowledged that random and local gossip communications make source identification difficult, there exists no general quantification of the level of anonymity provided to the source. This paper presents a principled method based on *ε-differential privacy* to analyze the inherent source anonymity of gossiping for a large class of graphs. First, we quantify the fundamental limit of source anonymity any gossip protocol can guarantee in an arbitrary communication graph. In particular, our result indicates that when the graph has poor connectivity, no gossip protocol can guarantee any meaningful level of differential privacy. This prompted us to further analyze graphs with controlled connectivity. We prove on these graphs that a large class of gossip protocols, namely *cobra walks*, offers tangible differential privacy guarantees to the source. In doing so, we introduce an original proof technique based on the reduction of a gossip protocol to what we call a *random walk with probabilistic die out*. This proof technique is of independent interest to the gossip community and readily extends to other protocols inherited from the security community, such as the *Dandelion* protocol. Interestingly, our tight analysis precisely captures the *trade-off* between dissemination time of a gossip protocol and its source anonymity.

2012 ACM Subject Classification Security and privacy → Privacy-preserving protocols

Keywords and phrases Gossip protocol, Source anonymity, Differential privacy

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.24

Related Version *Full Version*: `https://arxiv.org/abs/2308.02477` [28]

Acknowledgements The authors are thankful to Nirupam Gupta and Pierre-Louis Roman for fruitful discussions on the early version of the paper, and to the anonymous reviewers of DISC 2023 for their constructive comments.

1 Introduction

A gossip protocol (a.k.a., an epidemic protocol) is a distributed algorithm that disseminates information in a peer-to-peer system [47, 1, 34, 38, 19, 24]. Gossip protocols have been long used to model the propagation of infectious diseases [30, 37, 3], as well as rumors in social networks where users randomly exchange messages [17, 26]. It is commonly accepted that random and local communications between the users make source identification hard, and thus

[1] Part of the work was done when Sasha Voitovych was an intern at EPFL as part of the EPFL Excellence Research Internship Program.

provide *inherent* anonymity to the source of the gossip, i.e., anonymity that comes solely from the spreading dynamic without relying on any additional cryptographic primitives (as in [42]). Source anonymity in gossip protocols constitutes an active area of research. On the one hand, many works aim to establish *privacy guarantees* for the source of the gossip by concealing it against an adversary, e.g., hiding the whistleblower on social media [27, 25, 23, 26, 7, 22]. On the other hand, a large effort is put towards identifying *privacy limits* for the source of a gossip by designing adversarial strategies that accurately recover the source, e.g., "patient zero" identification in epidemics [33, 54, 46, 49, 9, 41].

Although a significant amount of research is dedicated to the investigation of source anonymity, existing approaches (as summarized in [33]) mainly focus on specific settings, such as locating the source of a gossip for a particular protocol, hiding it against a chosen adversarial strategy or examining the problem on a narrow family of graphs (trees, complete graphs, etc.). This prevents the results from being generalized, and it remains unclear how hard it is to recover the source of a gossip in general, naturally raising the following question.

What are the fundamental limits and guarantees on the inherent source anonymity of gossiping in a general setting?

We take an important step towards addressing this question by adapting the celebrated mathematical framework of ε-differential privacy (ε-DP) to our context [20, 21]. Although the concept is a gold standard to measure privacy leakage from queries on tabular databases, it can be also adapted to different privacy semantics and threat models [15]. In our context, we use ε-DP to measure the *inherent* source anonymity of gossiping in general graphs. We adopt a widely used threat model where the adversary aims to guess the source by monitoring the communications of a set of *curious* nodes in the graph [33, 46, 48, 54, 16, 23]. Using differential privacy enables us to overcome the limitations of previous work, as DP guarantees hold regardless of the exact strategy of the attacking adversary. Additionally, DP guarantees can be combined with any prior knowledge the adversary has on the location of the source, making our results generalizable. Our contributions can be summarized as follows.

1.1 Main results

We propose a mathematical framework that adapts the concept of differential privacy to quantify source anonymity in any graph (Section 3). In doing so, we highlight the importance of considering two types of adversaries: the *worst-case* and the *average-case*. For the worst-case adversary, we focus on privacy guarantees that hold *regardless* of the location of the curious nodes in the graph. In other words, these guarantees hold even if the adversary knows the communication graph in advance and chooses curious nodes strategically. For the average-case adversary, we focus on privacy guarantees that hold with high probability when curious nodes are chosen uniformly at random. Here, the adversary does not know the structure of the underlying communication graph in advance. Within our mathematical framework, we establish the following results for both adversarial cases.

Privacy limits. We first quantify a fundamental limit on the level of ε-DP any gossip protocol can provide on any graph topology (Section 4). This result indicates that no gossip protocol can ensure any level of differential privacy on poorly connected graphs. This motivates us to consider graphs with controlled connectivity, namely expander graphs. Expanders are an important family of strongly connected graphs that are commonly considered in the gossip protocols literature [8, 29, 11]. On this class, we get the following results.

Privacy guarantees. We prove that a large class of gossip protocols provides tangible differential privacy guarantees to the source (Section 5). We first consider the parameterized family of gossip protocols known as $(1+\rho)$-cobra walks [18, 11, 45, 6], which constitutes a natural generalization of a simple random walk. A cobra walk can be seen as an SIS (Susceptible-Infected-Susceptible) epidemic, a well-established model for analyzing the spread of epidemics and viruses in computer networks [30, 37]. In particular, a $(1+\rho)$-cobra walk is an instance of an SIS epidemic scheme where active nodes constitute the infectious set, the duration of the infectious phase is equal to one and every infected node can only infect one or two of its neighbors at a time. In order to establish differential privacy guarantees on this class of gossip protocols, we rely on the critical observation that the cobra walk has a quantifiable probability of mixing before hitting a curious node (see Section 1.2 for more details on this observation). This characteristic is not unique to cobra walks, as it is shared by several other types of gossip protocols. Accordingly, we also show how to generalize our privacy guarantees to the ρ-Dandelion protocol [7], first introduced as an anonymous communication scheme for Blockchains.

Dissemination time vs. privacy trade-off. As an important by-product of our analysis, we precisely capture the trade-off between dissemination time and privacy of a large class of gossip protocols operating on sufficiently dense graphs we call near-Ramanujan graphs (Section 7). The privacy-latency tension has been suggested several times in the literature [7, 5, 32]. However, our work presents the first formal proof of this long-standing empirical observation. Specifically, we show that our privacy guarantees are tight for both $(1+\rho)$-cobra walks [11] and ρ-Dandelion protocol [7]. Additionally, we give a tight analysis of the dissemination time as a function of parameter ρ. This analysis leads us to conclude that increasing parameter ρ results in a faster dissemination, but decreases privacy guarantees of the protocol, formally establishing the existence of a trade-off between privacy and dissemination time. As cobra walks are strongly related to SIS-epidemics, and Dandelion to anonymous protocols in peer-to-peer networks, our results are relevant for both epidemic and source anonymity communities.

1.2 Technical challenges & proof techniques

A major technical contribution of our paper is the privacy guarantee of $(1+\rho)$-cobra walks in non-complete graphs. The derivation of this result has been challenging to achieve for two reasons. Firstly, our objective is to establish differential privacy guarantees in general graphs, which is a more complex scenario than that of complete graphs (as seen in [5]), where any communication between pairs of nodes is equiprobable, and symmetry arguments can be utilized. Yet, this technique is no longer applicable to our work. The fact that no symmetry assumptions about graph structure can be made calls for new more sophisticated proof techniques. Second, cobra walks are challenging to analyze directly. State-of-the-art approaches analyzing the dissemination time of cobra walks circumvent this issue by analyzing a dual process instead, called BIPS [11, 12, 6]. There, the main idea is to leverage the duality of BIPS and cobra walks with respect to hitting times [11]. While hitting times provide sufficient information for analyzing the dissemination time of a cobra walk, they cannot be used to evaluate differential privacy, as they do not provide sufficient information about the probability distribution of the dissemination process. We overcome this difficulty through a two-step proof technique, described below.

Step I: Reduction to a random walk with probabilistic die out. To establish ε-differential privacy, we essentially show that two executions of the same $(1+\rho)$-cobra walk that started from different sources are statistically indistinguishable to an adversary monitoring a set of curious nodes. In doing so, we design a novel proof technique that involves reducing the analysis of gossip dissemination in the presence of curious nodes, to a *random walk with probabilistic die out*. Such a protocol behaves as a simple random walk on the communication graph G, but it is killed at each step (i) if it hits a curious node, or otherwise (ii) with probability ρ. We show that disclosing the death site of such a random walk to the adversary results in a bigger privacy loss than all the observations reported by the curious nodes during the gossip dissemination. Then, we can reduce the privacy analysis of cobra walks to the study of such a random walk with probabilistic die out.

Step II: Analysis of a random walk with probabilistic die out. To study a random walk with probabilistic die out, we characterize the spectral properties of the (scaled) adjacency matrix $\boldsymbol{Q}$ corresponding to the subgraph of G induced by the non-curious nodes. In particular, we show that if curious nodes occupy a small part of every neighborhood in G, then the subgraph induced by non-curious nodes (i) is also an expander graph and (ii) has an almost-uniform first eigenvector. While (i) is a direct consequence of the Cauchy Interlacing Theorem, (ii) is more challenging to obtain. We need to bound $\boldsymbol{Q}$ from above and below by carefully designed matrices with an explicit first eigenvector. Combining (i) and (ii) allows us to precisely estimate the behavior of the random walk with probabilistic die out, which yields the desired differential privacy guarantees.

Generality of the proof. The reduction to a random walk with probabilistic die out is the most critical step of our proof. It is general and allows us to analyze several other protocols without having to modify the most technical part of the proof (Step II above). We demonstrate the generality of this technique by applying this reduction to the Dandelion protocol and obtain similar privacy guarantees to cobra walks.

1.3 Related work

Inherent anonymity of gossiping. To the best of our knowledge, only two previous works have attempted to quantify the inherent source anonymity of gossiping through differential privacy [5, 32]. The former work [5] is the first to analyze source anonymity using differential privacy. It measures the guarantees of a class of gossip protocols with a muting parameter (which we call "muting push" protocols) and contrasts these guarantees with the dissemination time of these protocols on a complete graph. Both the threat model and the nature of the technical results in [5] heavily depend on the completeness of the graph. In such a context, the analysis is considerably simplified for two reasons. Firstly, the presence of symmetry allows for the curious node locations to be ignored, rendering the average-case and the worst-case adversaries equivalent. Secondly, in contrast to what would happen in non-complete graphs, since any node can communicate with any other node in each round, a single round of communication is sufficient to hide the identity of the source. However, when considering the spread of epidemics or the propagation of information in social networks, communication graphs are seldom complete [43]. Our work highlights that non-completeness of the graph potentially challenges the differential privacy guarantees that gossip protocols can achieve and also makes it important to distinguish between average and worst-case threat models. Therefore, our results constitute a step toward a finer-grained analysis of the anonymity of gossiping in general graphs. Note that our work can be seen as a strict generalization of the results of [5], since, in addition to cobra walks and Dandelion, we also show that our proof techniques described in Section 1.2 apply to "muting push" protocols (see Appendix E of the full version of the paper [28]).

The second approach [32] addresses a problem that appears to be similar to ours at first glance, as it aims to quantify source anonymity in non-complete graphs. However, the authors consider a different threat model, where an adversary can witness any communication with some probability instead of only those passing through the curious nodes. Furthermore, the paper only gives negative results and does not provide any differential privacy guarantees, which is the most technically challenging part of our paper.

Dissemination time vs. privacy trade-off. Several previous works [53, 4, 14, 51] have suggested the existence of a tension between source anonymity (i.e., privacy) and latency of message propagation. Under the threat model we consider in this work (with curious nodes), [7] conjectured that the Dandelion protocol would exhibit a trade-off between (their definition of) source anonymity and dissemination time. Later, works [5] and [32] provided more tangible evidence for the existence of a dissemination time vs. privacy trade-off when analyzing source anonymity through differential privacy. However, these works do not provide a tight analysis of the tension between dissemination time and privacy, hence making their observation incomplete. To the best of our knowledge, our work is the first to rigorously demonstrate the existence of a trade-off between the dissemination time of a gossip protocol and the privacy of its source thanks to the *tightness* of our analysis.

2 Preliminaries

For a vector $\boldsymbol{x} \in \mathbb{R}^m$, we denote by x_i its ith coordinate, i.e., $\boldsymbol{x} = (x_1, x_2, \dots, x_m)^\top$. Similarly, for a matrix $\boldsymbol{M} \in \mathbb{R}^{m \times m'}$, we denote by M_{ij} its entry for the ith row and jth column. Furthermore, for any symmetric matrix $\boldsymbol{M} \in \mathbb{R}^{m \times m}$, we denote by $\lambda_1(\boldsymbol{M}) \geq \lambda_2(\boldsymbol{M}) \geq \ldots \geq \lambda_m(\boldsymbol{M})$ its eigenvalues. We use $\mathbf{1}_m \in \mathbb{R}^m$ to denote an all-one vector, $\boldsymbol{I}_m \in \mathbb{R}^{m \times m}$ to denote the identity matrix, $\boldsymbol{J}_m \in \mathbb{R}^{m \times m}$ to denote an all-one square matrix, and $\boldsymbol{O}_{m \times m'} \in \mathbb{R}^{m \times m'}$ to denote an all-zero matrix. Finally, for any $\boldsymbol{x} \in \mathbb{R}^m$, we denote by $||\boldsymbol{x}||_p \triangleq \left(\sum_{i=1}^m |x_i|^p\right)^{1/p}$ the ℓ_p norm of $\boldsymbol{x}$ for $p \in [1, \infty)$ and by $||\boldsymbol{x}||_\infty \triangleq \max_{i \in m} |x_i|$ the ℓ_∞ norm of $\boldsymbol{x}$.

Throughout the paper, we use the *maximum divergence* to measure similarities between probability distributions. We consider below a common measurable space (Ω, Σ) on which the probability measures are defined. Let μ, ν be two probability measures over Σ. The *max divergence* between μ and ν is defined as[2]

$$D_\infty\left(\mu \parallel \nu\right) \triangleq \sup_{\sigma \in \Sigma,\ \mu(\sigma) > 0} \ln \frac{\mu(\sigma)}{\nu(\sigma)}.$$

Furthermore, for two random variables X, Y with laws μ and ν respectively, we use the notation $D_\infty\left(X \parallel Y\right)$ to denote $D_\infty\left(\mu \parallel \nu\right)$.

[2] Note that we allow $\nu(\sigma) = 0$ in the definition. If $\nu(\sigma) = 0$ but $\mu(\sigma) > 0$ for some $\sigma \in \Sigma$, the max divergence is set to ∞ by convention.

2.1 Graph theoretical terminology

Consider an undirected connected graph $G = (V, E)$, where V is the set of nodes and E is the set of edges. G cannot have self-loops or multiple edges. For any $v \in V$, we denote by $N(v)$ the set containing the neighbours of v in G and by $\deg(v)$ the number of edges incident to v. Furthermore, G is said to be a *regular graph*, if there exists $d(G)$ such that $\deg(v) = d(G)$ for every $v \in V$; $d(G)$ is called the degree of the graph. Additionally, for a set $U \subseteq V$ and $v \in V$, we denote by $\deg_U(v)$ the number of neighbours of v contained in U, i.e., $\deg_U(v) = |N(v) \cap U|$. Below, we introduce some additional graph terminology.

▶ **Definition 1** (Vertex cut & connectivity). *A* vertex cut *of G is a subset of vertices $K \subseteq V$ whose removal disconnects G or leaves just one vertex. A minimum vertex cut of G is a vertex cut of the smallest size. The size of a minimum vertex cut for G, denoted $\kappa(G)$, is called the vertex connectivity of G.*

Consider an undirected connected graph $G = (V, E)$ of size n where V is an ordered set of nodes. We denote by $\boldsymbol{A}$ the adjacency matrix of G, i.e., $A_{vu} = 1$ if $\{v, u\} \in E$ and $A_{vu} = 0$ otherwise. We also denote by $\hat{\boldsymbol{A}} = \boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2}$ the normalized adjacency matrix of G, where $\boldsymbol{D}$ is the diagonal degree matrix, i.e., $D_{vu} = \deg(v)$ if $v = u$ and 0 otherwise. Since $\hat{\boldsymbol{A}}$ is a symmetric and normalized matrix, the eigenvalues of $\hat{\boldsymbol{A}}$ are real valued and $\lambda_1(\hat{\boldsymbol{A}}) = 1$. Using this terminology, the *spectral expansion* of G is defined as

$$\lambda(G) \triangleq \max\{|\lambda_2(\hat{\boldsymbol{A}})|, |\lambda_n(\hat{\boldsymbol{A}})|\}. \tag{1}$$

▶ **Definition 2** (Expander graph). *Consider an undirected regular graph G. If $d(G) = d$ and $\lambda(G) \leq \lambda$, then G is said to be a (d, λ)-expander graph.*

2.2 Gossip protocols

Consider an undirected connected communication graph $G = (V, E)$ where two nodes $u, v \in V$ can directly communicate if and only if $\{u, v\} \in E$. One node $s \in V$, called the *source*, holds a unique gossip g to be propagated throughout the graph. In this context, *a gossip protocol* is a predefined set of rules that orchestrates the behavior of the nodes with regard to the propagation of g. Essentially, the goal of a protocol is that with probability 1 every node in G eventually receives g. We assume discrete time steps and synchronous communication, i.e., the executions proceed in rounds of one time step.[3] While every node in G has access to the global clock, we assume that the execution of the protocol starts at a time $t_\star \in \mathbb{Z}$, which is *only* known to the source s.

Execution of a gossip protocol. At any point of the execution of the protocol, a node $u \in V$ can either be active or non-active. Only active nodes are allowed to send messages during the round. A gossip protocol always starts with the source s being the only active node, and at every given round $t + 1$ active nodes are the nodes that received the gossip at round t. We will use $X_t \subseteq V$ to denote the set of active nodes at the beginning of round $t \geq t_\star$ and set $X_{t_\star} = \{s\}$ by convention. Denoting by $(u \to v)$ a communication between nodes u and v, we define $\mathcal{C}$ to be the set of all possible communications in G, i.e., $\mathcal{C} = \{(u \to v) : \{u, v\} \in E\} \cup \{(u \to u) : u \in V\}$. Note that we allow an active node

[3] Although, for clarity, we focus on a synchronous communication, our analysis of privacy guarantees in Section 5 readily extends to an asynchronous setting.

u to send a fictitious message to itself to stay active in the next communication round. Then, the t^{th} round of an execution for a given protocol $\mathcal{P}$ can be described by a pair (X_t, C_t), where $X_t \subseteq V$ is a set of active nodes, and C_t is the (multi)set of communications of $\mathcal{C}$ which happened at round t. We denote by S the random variable characterizing the *execution* of the protocol. Naturally, an *execution* is described by a sequence of rounds, i.e., $S = \{(X_t, C_t)\}_{t \geq t_\star}$. We define *expected dissemination time* of the protocol as the expected number of rounds for all nodes to receive the gossip during an execution. Finally, we denote $\mathcal{E}$ the set of all possible executions.

Cobra and random walk. Coalescing-branching random walk protocol (a.k.a., cobra walk) [18, 11, 45, 6] is a natural generalization of a simple random walk that is notably useful to model and understand Susceptible-Infected-Susceptible (SIS) epidemic scheme [30, 37]. We consider a $(1+\rho)$-cobra walk as studied in [11] with $\rho \in [0,1]$[4]. This is a gossip protocol where at every round $t \geq t_\star$, each node $u \in X_t$ samples a token from a Bernoulli distribution with parameter ρ. If the token equals zero, u samples uniformly at random a node v from its neighbors $N(u)$ and communicates the gossip to it, i.e., $(u \to v)$ is added to C_t. If the token equals one, the protocol *branches*. Specifically, u independently samples two nodes v_1 and v_2 at random (with replacement) from its neighbors and communicates the gossip to both of them, i.e., $(u \to v_1)$, and $(u \to v_2)$ are added to C_t. At the end of the round, each node $u \in X_t$ deactivates. Note that, when $\rho = 0$, this protocol degenerates into a simple random walk on the graph; hence it has a natural connection with this random process.

Dandelion protocol. Dandelion is a gossip protocol designed to enhance source anonymity in the Bitcoin peer-to-peer network. Since it was introduced in [7], it has received a lot of attention from the cryptocurrency community. Dandelion consists of two phases: (i) the anonymity phase, and (ii) the spreading phase. The protocol is parameterized by $\rho \in [0,1)$, the probability of transitioning from the anonymity phase to the spreading phase. Specifically, the phase of the protocol is characterized by a token $anonPhase \in \{0,1\}$ held by a global oracle and initially equal to 0. At the beginning of each round of the Dandelion execution, if $anonPhase = 1$ the global oracle sets $anonPhase = 0$ with probability ρ and keeps $anonPhase = 1$ with probability $1 - \rho$. Once $anonPhase = 0$, the global oracle stops updating the token. Based on this global token, at each round, active nodes behave as follows. If the $anonPhase = 1$, the execution is in the anonymity phase and an active node u samples a node v uniformly at random from its neighborhood $N(u)$ and communicates the gossip to it, i.e., $(u \to v)$ is added to C_t. Afterwards, node u deactivates, i.e., in the anonymity phase only one node is active in each round. If the $anonPhase = 0$, the execution is in the spreading phase. Then the gossip is broadcast, i.e., each node $u \in X_t$ communicates the gossip to all of its neighbors and for $\forall v \in N(u)$, $(u \to v)$ is added to C_t.

[4] Some prior works also study k-cobra walks with branching parameter $k \geq 3$ [18]. We do not consider this class, since our negative result for a 2-cobra walk (Theorem 34 in the full version of the paper [28]) implies that a k-cobra walk for any $k \geq 3$ does not satisfy a reasonable level of differential privacy.

3 Mathematical framework for source anonymity in general graphs

Given a source and a gossip protocol, we fix the probability space $(\mathcal{E}, \Sigma, \mathbb{P})$, where Σ is the standard cylindrical σ-algebra on $\mathcal{E}$ (as defined in Appendix A.1 of [55]) and $\mathbb{P}$ is a probability measure characterizing the executions of the protocol. In the remaining, to avoid measurability issues, we only refer to subsets of $\mathcal{E}$ from Σ.

3.1 Measuring source anonymity with differential privacy

We now describe the mathematical framework we use to quantify source anonymity of gossiping. We consider a threat model where an external adversary has access to a subset $F \subset V$ of size $f < n - 1$ of *curious* nodes. Curious nodes in F execute the protocol correctly, but report their communications to the adversary. The adversary aims to identify the source of the gossip using this information. We distinguish two types of adversaries, namely worst-case and average-case, depending on the auxiliary information they have on the graph.

Threat models: worst-case and average-case adversaries. On the one hand, a *worst-case* adversary is aware of the structure of the graph G and may choose the set of curious nodes to its benefit. On the other hand, the *average-case* adversary is not aware of the topology of G before the start of the dissemination, hence the set of curious nodes is chosen uniformly at random among all subsets of V of size f. We assume that the messages shared in the network are unsigned and are passed unencrypted. Also, the contents of transmitted messages (containing the gossip) do not help to identify the source of the gossip. In other words, adversaries can only use the information they have on the dissemination of the gossip through the graph to locate the source. We also assume that the adversary does not know the exact starting time $t_\star \in \mathbb{Z}$ of the dissemination. To formalize the observation received by the external adversary given a set of curious nodes F, we introduce a function $\Psi^{(F)}$ that takes as input communications C from a single round and outputs only the communications of C visible to the adversary. Note that a communication $(v \to u)$ is visible to the adversary if and only if either v or u belongs to F. Consider an execution $S = \{(X_t, C_t)\}_{t \geq t_\star}$ of a gossip protocol, and denote by t_{ADV} the first round in which one of the curious nodes received the gossip. Then we denote by $S_{\text{ADV}} = \{\Psi^{(F)}(C_t)\}_{t \geq t_{\text{ADV}}}$ the random variable characterizing the observation of the adversary for the whole execution. Note that the adversary does not know $t_\star$, hence it cannot estimate how much time passed between $t_\star$ and t_{ADV}.

▶ Remark 3. For Dandelion, the adversary actually also has access to the value of *anonPhase* in round t, i.e., we have $S_{\text{ADV}} = \{\Psi^{(F)}(C_t), anonPhase_t\}_{t \geq t_{\text{ADV}}}$. We omit this detail from the main part of the paper for simplicity of presentation, but it does not challenge our results on privacy guarantees. See Appendix C.4 in the full version of the paper [28] for more details.

Measuring source anonymity. We formalize source anonymity below by adapting the well-established definition of differential privacy. In the remaining of the paper, for a random variable A, we will write $A^{(s)}$ to denote this random variable conditioned on the node $s \in V \setminus F$ being the source. In our setting, we say that a gossip protocol satisfies differential privacy if for any $u, v \in V$ the random sequences $S^{(v)}_{\text{ADV}}$ and $S^{(u)}_{\text{ADV}}$ are statistically indistinguishable. More formally, we define differential privacy as follows.

▶ **Definition 4** (Differential privacy). *Consider an undirected graph $G = (V, E)$ and a set of curious nodes $F \subset V$. Then, a gossip protocol satisfies ε-*differential privacy *(ε-DP) for the set F if, for any two nodes $v, u \in V \setminus F$, the following holds true*

$$D_\infty\left(S^{(v)}_{\text{ADV}} \parallel S^{(u)}_{\text{ADV}}\right) \leq \varepsilon.$$

When establishing differential privacy guarantees against a *worst-case adversary*, we aim to find a value ε which only depends on the number of curious nodes f, and is *independent* of the identity of the nodes in F. Accordingly, we say that a gossip protocol satisfies *ε-DP against a worst-case adversary* if it satisfies ε-DP for any set $F \subset V$ such that $|F| = f$.

When establishing differential privacy against an *average-case adversary*, we aim to find a value of ε for which the protocol satisfies ε-DP *with high probability*[5] when choosing the f curious nodes uniformly at random from V. Formally, let $\mathcal{U}_f(V)$ be the uniform distribution over all subsets of V of size f, a gossip protocol satisfies *ε-DP against an average-case adversary* if

$$\mathbb{P}_{F \sim \mathcal{U}_f(V)} \left[\max_{v,u \in V \setminus F} D_\infty \left(S_{\text{ADV}}^{(v)} \parallel S_{\text{ADV}}^{(u)} \right) \leq \varepsilon \right] \geq 1 - \frac{1}{n}. \tag{2}$$

3.2 Semantic of source anonymity

Differential privacy is considered the gold standard definition of privacy, since ε-DP guarantees hold *regardless* of the strategy of the adversary and any prior knowledge it may have on the location of the source. Yet, the values of ε are notoriously hard to interpret [39, 31]. To better understand the semantic of our definition of differential privacy, we consider below two simple examples of adversarial strategies: maximum a posteriori and maximum likelihood estimations. For these strategies, we derive bounds on the probability of an adversary successfully guessing the source in an effort to give a reader an intuition on the meaning of the parameter ε. The proofs are given in Appendix F of the full version of the paper [28].

Maximum a posteriori strategy. Maximum a posteriori (MAP) strategy can be described as follows. Suppose an adversary has an a priori distribution p that assigns to every node in $V \setminus F$ a probability of being the source of the gossip. Intuitively, p corresponds to the set of beliefs the adversary has on the origin of the gossip before observing the dissemination. This prior might reflect information acquired from any auxiliary authority or some expert knowledge on the nature of the protocol. Suppose the adversary observes an event σ. Then, a MAP-based adversary "guesses" which node is the most likely to be the source, assuming event σ occurred and assuming the source has been sampled from the prior distribution p. Such guess is given by

$$\hat{s}_{MAP} = \operatorname*{argmax}_{v \in V \setminus F} \mathbb{P}_{s \sim p} \left[v = s \mid S_{\text{ADV}}^{(s)} \in \sigma \right] = \operatorname*{argmax}_{v \in V \setminus F} \mathbb{P} \left[S_{\text{ADV}}^{(v)} \in \sigma \right] p(v). \tag{3}$$

Using ε-DP, we can upper bound the success probability of such a guess. Suppose the protocol satisfies ε-DP, then the probability of correctly identifying a source $s \sim p$ conditioned on σ happening is upper bounded as follows

$$\mathbb{P}_{s \sim p} \left[\hat{s}_{MAP} = s \mid S_{\text{ADV}}^{(s)} \in \sigma \right] \leq \exp(\varepsilon) p \left(\hat{s}_{MAP} \right). \tag{4}$$

Such an upper bound has a simple interpretation. Note that $p(\hat{s}_{MAP})$ characterizes the maximum probability of a successfully guessing $\hat{s}_{MAP}$ based solely on adversary's prior knowledge. Then, the upper bound above states that the probability of a successful guess after observing the dissemination is amplified by a factor of at most $\exp(\varepsilon)$ compared to success probability of a guess based on a priori knowledge only.

[5] An event is said to hold with high probability on graph G of size n, if it holds with probability $\geq 1 - 1/n$.

Maximum likelihood strategy. Maximum likelihood estimation (MLE) occupies a prominent place [23, 49, 50, 46] in the literature, both for designing source location attacks, and for defending against adversaries that follow an MLE strategy. This method is a special instance of MAP estimator in (3) with a uniform prior distribution $p = \mathcal{U}(V \setminus F)$ on the source. We can show that, if the protocol satisfies ε-DP, such guess has a bounded success probability.

$$\mathbb{P}_{s \sim \mathcal{U}(V \setminus F)} \left[\hat{s}_{MLE} = s \mid S_{\text{ADV}}^{(s)} \in \sigma \right] \leq \frac{\exp(\varepsilon)}{n - f}. \tag{5}$$

4 Fundamental limits of source anonymity: lower bound on ε

We start by studying the fundamental limits of differential privacy in general graphs. Specifically, we aim to show that vertex connectivity constitutes a hard threshold on the level of source anonymity gossiping can provide. First, we present a warm-up example indicating that in a poorly connected graph, no gossip protocol can achieve any meaningful level of differential privacy against a worst-case adversary. We then validate this intuition by devising a universal lower bound on ε that applies for any gossip protocol and any undirected connected graph. Complete proofs related to this section can be found in Appendix B of the full version of the paper [28].

4.1 Warm-up

Consider a non-complete graph $G = (V, E)$ and $K \subset V$, a vertex cut of G. Then, by definition, deleting K from G partitions the graph into two disconnected subgraphs. When $f \geq |K|$, a worst-case adversary can take F such that $K \subseteq F$. Then, the curious nodes can witness all the communications that pass from one subgraph to the other. Intuitively, this means that any two nodes that are not in the same subgraph are easily distinguishable by the adversary. Hence, differential privacy cannot be satisfied. This indicates that the level of differential privacy any gossip protocol can provide in a general graph fundamentally depends on the connectivity of this graph. To validate this first observation and determine the fundamental limits of gossiping in terms of source anonymity, we now determine a lower bound on ε.

4.2 Universal lower bound on ε

We present, in Theorem 5, a universal lower bound on ε which holds for any gossip protocol, on any connected graph and for both the worst-case and the average-case adversaries.

▶ **Theorem 5.** *Consider an undirected connected graph $G = (V, E)$ of size n, a number of curious nodes $f > 1$, and an arbitrary gossip protocol $\mathcal{P}$. If $\mathcal{P}$ satisfies ε-DP against an average-case or a worst-case adversary, then*

$$\varepsilon \geq \ln(f - 1).$$

Moreover, if $\kappa(G) \leq f$, then $\mathcal{P}$ cannot satisfy ε-DP with $\varepsilon < \infty$ against a worst-case adversary.

Proof sketch. To establish the above lower bound, we assume that the adversary simply predicts that the first non-curious node to contact the curious set is the source of the gossip. As the definition of differential privacy does not assume a priori knowledge of the adversarial strategy, computing the probability of success for this attack provides a lower bound on ε.

We first demonstrate the result for the average-case adversary. Assume that F is sampled uniformly at random from V. We can show that there exists $v \in V$ such that the attack implemented by the adversary succeeds with large enough probability when v is the source of the gossip. This fact essentially means that this v is easily distinguishable from any other node in the graph, which yields the lower bound $\varepsilon \geq \ln(f-1)$ in the average case. We now consider the worst-case adversary. Assume that F can be chosen by the adversary. As the lower bound $\varepsilon \geq \ln(f-1)$ holds with positive probability when F is chosen at random, there exists at least one set F for which it holds. Choosing this set of curious nodes establishes the claim for the worst-case adversary. Furthermore, when $\kappa(G) \leq f$, we follow the intuition from Section 4.1 to build a set F that disconnects the graph. Using this set, we prove that ε cannot be finite. ◀

Theorem 5 shows that the connectivity of the graph is an essential bottleneck for differential privacy in a non-complete graph. This stipulates us to study graphs with controlled connectivity, namely (d, λ)-expander graphs. Note that in a (d, λ)-expander, the vertex connectivity does not exceed d. Hence, Theorem 5 implies that no gossip protocol can satisfy any meaningful level of differential privacy against a worst-case adversary on a (d, λ)-expander if $f \geq d$. Considering this constraint, while studying a gossip against a worst-case adversary, we only focus on cases where the communication graph G has a large enough degree d.

5 Privacy guarantees: upper bound on ε

We now present a general upper bound on ε that both holds for $(1+\rho)$-cobra walks and ρ-Dandelion on d-regular graphs with fixed expansion, i.e., (d, λ)-expander graphs. Complete proofs related to this section can be found in Appendix C of the full version of the paper [28]. Our privacy guarantees are quite technical, which is justified by the intricacies of the non-completeness of the graph. Recall that, in the case of complete topologies analyzed in [5], after one round of dissemination all information on the source is lost unless a curious node has been contacted. However, in a general expander graph, this property does not hold anymore. Indeed, even after multiple rounds of propagation, the active set of the protocol can include nodes that are close to the location of the source s. Thus, differential privacy may be compromised.

5.1 Adversarial density

The attainable level of source anonymity for a given protocol is largely influenced by the location of curious nodes. However, accounting for all possible placements of curious nodes is a very challenging and intricate task. To overcome this issue and state our main result, we first introduce the notion of *adversarial density* that measures the maximal fraction of curious nodes that any non-curious node may have in its neighborhood. Upper bounding the adversarial density of a graph is a key element to quantifying the differential privacy guarantees of a gossip protocol. Formally, this notion is defined as follows.

▶ **Definition 6.** *Consider an undirected connected d-regular graph $G = (V, E)$, and an arbitrary set of curious nodes $F \subseteq V$. The* adversarial density *of F in G, denoted α_F, is the maximal fraction of curious nodes that any node $v \in V \setminus F$ has in its neighborhood. Specifically,*

$$\alpha_F \triangleq \max_{v \in V \setminus F} \frac{\deg_F(v)}{d}.$$

For any set of curious nodes F, we have $\alpha_F \leq f/d$. Hence, even when F is chosen by a worst-case adversary, the adversarial density is always upper bounded by f/d. However, for the average-case adversary we can obtain a much tighter bound, stated in Lemma 7 below.

▶ **Lemma 7.** *Consider an undirected connected d-regular graph $G = (V, E)$ of size n and a set of curious nodes $F \sim \mathcal{U}_f(V)$, with adversarial density α_F. We denote $\beta = f/n$ and $\gamma = \ln(n)/(ed)$, where e is Euler's constant. Then, with probability at least $1 - 1/n$, $\alpha_F \leq \alpha$ with*

$$\alpha \leq 4e \frac{\max\{\gamma, \beta\}}{1 + \max\{\ln(\gamma) - \ln(\beta), 0\}}.$$

Furthermore, if there exist $\delta > 0, c > 0$ such that $f/n > c$ and $d > \ln(n)/(c^2\delta^2)$ then a similar statement holds with $\alpha \leq (1 + \delta)\beta$.

We deliberately state this first lemma in a very general form. This allows us to precisely quantify how the upper bound on the adversarial density improves as f decreases. To make this dependency clearer, we provide special cases in which the bound on α_F is easily interpreted. First, assume that $d \in \omega_n(\log(n))$ and $f/n \in \Omega_n(1)$. Then, α_F is highly concentrated around f/n, up to a negligible multiplicative constant, when n is large enough. On the other hand, when the ratio f/n becomes subconstant, the concentration becomes looser. In particular, if $d \in \omega_n(\log(n))$ and $f/n \in o_n(1)$, then $\alpha_F \in o_n(1)$ with high probability. Finally, if f/n drops even lower (e.g., when $f/n \in n^{-\Omega_n(1)}$), we get $\alpha_F \in O_n(1/d)$ or $\alpha_F \in n^{-\Omega_n(1)}$ with high probability for any d.

5.2 General upper bound on ε

Thanks to Lemma 7 bounding adversarial density, we can now state our main theorem providing a general upper bound on ε for $(1 + \rho)$-cobra walks and ρ-Dandelion.

▶ **Theorem 8.** *Consider an undirected connected (d, λ)-expander graph $G = (V, E)$ of size n, let f be the number of curious nodes, and let $\mathcal{P}$ be a $(1 + \rho)$-cobra walk with $\rho < 1$. Set $\alpha = f/d$ (resp. set α as in Lemma 7). If $\lambda < 1 - \alpha$, then $\mathcal{P}$ satisfies ε-DP against a worst-case adversary (resp. an average-case adversary) with*

$$\varepsilon = \ln(\rho(n - f) + f) - 2\tilde{T}\ln(1 - \alpha) - \tilde{T}\ln(1 - \rho) - \ln(1 - \lambda) + \ln(24),$$

and $\tilde{T} = \left\lceil \log_{\frac{\lambda}{1-\alpha}} \left(\frac{1-\alpha}{4(n-f)}\right) \right\rceil \left(\log_{\frac{\lambda}{1-\alpha}}(1 - \alpha) + 2\right) + 2.$

The above statement also holds if $\mathcal{P}$ is a ρ-Dandelion protocol with $\rho < 1$.

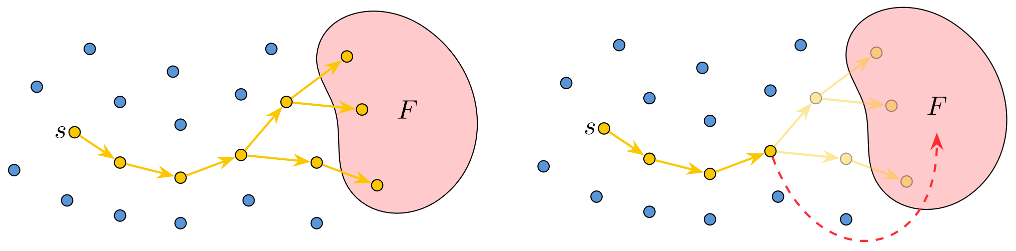
Note that the upper bound on ε in Theorem 8 improves as the number of curious nodes f decreases (since α decreases with f) or when the expansion improves (as λ decreases, $\tilde{T}$ also decreases). Yet, there is a complex interplay between the parameters n, f, d, and λ above. Additionally, we point out that for a worst-case adversary the privacy guarantees can be established only if $f/d < 1$. For the average-case, this assumption can be dropped, and we are able to establish positive results for f as high as $\Theta_n(n)$.

6 Proof sketch for Theorem 8

Although results for worst-case and average-case adversaries have their own technical specificity, they both share the same general idea. Specifically, we introduce a random process that helps bounding from above the value of ε. This random process resembles a random walk that at each step reveals its position to the adversary with some probability that depends on ρ and on the state of the process. We call this process a *random walk with probabilistic die out*. Then, we show that such random walk mixes sufficiently well before its position is revealed, which provides indistinguishability between any two possible sources.

The first half of our proof (step I) relies on the reduction of a gossip protocol to a random walk with probabilistic die out. This part is slightly different for different protocols, but for simplicity we only present step I for the cobra walk. In the second half (step II), we only analyze a random walk with probabilistic die out. It is hence universal and applies to both cobra walks and Dandelion protocols. Complete proofs for both protocols can be found in Appendix C of the full version of the paper [28].

6.1 Step I: reduction to a random walk with probabilistic die out



(a) Dissemination of a cobra walk. **(b)** Dissemination of a random walk with die out.

Figure 1 Illustration of the reduction from a cobra walk (Fig. 1a) to a random walk with probabilistic die out (Fig. 1b). In Fig. 1a, the dissemination continues after the walk branches and hits the curious set F in several places. In the random walk with die out, instead of letting the dissemination branch, we stop the dissemination as soon as the cobra walk branches and report the position of the branching node.

Consider a $(1+\rho)$-cobra walk started at s and denote $W^{(s)}$ the random variable indicating the last position of the cobra walk before it either branches or hits a curious node. More formally, if the round at which the cobra walk branches or contacts a curious node for the first time is τ, then the active set at this round would be $X_\tau^{(s)} = \{W^{(s)}\}$, with $W^{(s)} \in V \setminus F$. We first show that disclosing $W^{(s)}$ to the adversary reveals more information about the source than $S_{\text{ADV}}^{(s)}$. Intuitively, this follows from the Markov property of the active set $\left\{X_t^{(s)}\right\}_{t\geq t_\star}$ of the cobra walk. In fact, by definition of τ, we have $\tau \leq t_{\text{ADV}}$. Hence, the sequence of adversarial observations $S_{\text{ADV}}^{(v)}$ can be obtained from $X_\tau^{(s)} = \left\{W^{(s)}\right\}$ via a randomized mapping independent of the initial source s. Then, using the data processing inequality Theorem 14 of [40]) we show that for any two possible sources $u, v \in V\backslash F$, we have

$$D_\infty\left(S_{\text{ADV}}^{(v)} \parallel S_{\text{ADV}}^{(u)}\right) \leq D_\infty\left(W^{(v)} \parallel W^{(u)}\right). \tag{6}$$

This means that it suffices to obtain an upper bound on $D_\infty\left(W^{(v)} \parallel W^{(u)}\right)$ for any $u, v \in V\backslash F$ to obtain an appropriate value for ε. Then, we note that $W^{(s)}$ can be described as the death site of a process we refer to as *random walk with probabilistic die out*, which was started at s. Such a process constitutes a random walk which is killed at each step either (i) if it hits a curious node, or otherwise (ii) with probability ρ. We illustrate this process in Figure 1 and how it relates to the cobra walk.

6.2 Step II: upper bounding the max divergence between death sites

The rest of the proof is dedicated to analyzing the probability distribution of the death site of such a process. Let $\boldsymbol{Q} = \hat{\boldsymbol{A}}[V \setminus F]$ be the principled submatrix of $\hat{\boldsymbol{A}}$ induced by the rows and columns of $V \setminus F$ and let $\boldsymbol{R}$ be a diagonal matrix of size $(n-f) \times (n-f)$ such that $R_{ww} = \deg_F(w)/d$ for every $w \in V \setminus F$. Then, $W^{(s)}$ can be described as an absorbing Markov chain. More precisely, let nodes from $V \setminus F$ be transient states, and equip every node $w \in V \setminus F$ with an absorbing state sink(w) which corresponds to the event of dying at w. The transition matrix of our absorbing Markov chain can be written in a block form as

$$\boldsymbol{P} = \begin{bmatrix} (1-\rho)\boldsymbol{Q} & \boldsymbol{O}_{(n-f)\times(n-f)} \\ \rho \boldsymbol{I}_{n-f} + (1-\rho)\boldsymbol{R} & \boldsymbol{I}_{n-f} \end{bmatrix}. \tag{7}$$

In the above, $\boldsymbol{P}_{xy}$ denotes the transition probability from a state y to a state x. The first $n-f$ columns correspond to transition probabilities from transient states $w \in V \setminus F$ and the last $n-f$ ones correspond to transition probabilities from absorbing states sink(w) for $w \in V \setminus F$. The probability of transitioning between two transient states $v, u \in V \setminus F$ (top-left block of $\boldsymbol{P}$) is defined similarly to a simple random walk on G, multiplied by the probability of not branching $(1-\rho)$. The transition probability between w and sink(w) (bottom-left block of $\boldsymbol{P}$) is naturally defined as the probability of branching plus the probability of contacting a curious node at the current step without branching.

According to the above, being absorbed in sink(w) corresponds to the event $W^{(s)} = w$. Hence, using $\boldsymbol{Q}$ and $\boldsymbol{R}$ to compute a closed form expression for absorbing probabilities of the above Markov chain, we can rewrite $D_\infty\left(W^{(v)} \parallel W^{(u)}\right)$ as follows

$$D_\infty\left(W^{(v)} \parallel W^{(u)}\right) = \max_{w \in V \setminus F} \ln \frac{(\boldsymbol{I}_{n-f} - (1-\rho)\boldsymbol{Q})^{-1}_{vw}}{(\boldsymbol{I}_{n-f} - (1-\rho)\boldsymbol{Q})^{-1}_{uw}}. \tag{8}$$

To conclude the proof, we now need to upper bound the right-hand side (8). To do so, we first note that, as per Theorem 3.2.1 in [35], we can use the following series decomposition,

$$(\boldsymbol{I}_{n-f} - (1-\rho)\boldsymbol{Q})^{-1} = \sum_{t=0}^{\infty} (1-\rho)^t \boldsymbol{Q}^t. \tag{9}$$

This means that we can reduce the computation of $D_\infty\left(W^{(v)} \parallel W^{(u)}\right)$ to analyzing the powers of the matrix $\boldsymbol{Q}^t$. Furthermore, for large values of t, we can approximate $\boldsymbol{Q}^t$ by a one-rank matrix using the first eigenvalue and the first eigenvector of $\boldsymbol{Q}$. This motivates us to study the spectral properties of $\boldsymbol{Q}$. We begin by showing that $\boldsymbol{Q}$ is dominated by its first eigenvalue. To further estimate the coordinates of the first eigenvector of $\boldsymbol{Q}$, we need to introduce subsidiary matrices $\overline{\boldsymbol{Q}}$ and $\underline{\boldsymbol{Q}}$. We carefully design these matrices to have an explicit first eigenvector and so that their entries bound from above and below respectively those of $\boldsymbol{Q}$. Using these two properties, we obtain a measure of how far the first eigenvector of $\boldsymbol{Q}$ is from the uniform vector $\mathbf{1}_{n-f}/\sqrt{n-f}$. By controlling spectral properties of $\boldsymbol{Q}$, we establish efficient one-rank approximations of high powers of $\boldsymbol{Q}$. Applying this to (8), we obtain an upper bound on the max divergence between $W^{(v)}$ and $W^{(u)}$, for any $u, v \in V \setminus F$. Specifically, assuming that the adversarial density $\alpha_F < 1 - \lambda$, we get

$$D_\infty\left(W^{(v)} \parallel W^{(u)}\right) \leq \ln(\rho(n-f)+f) - 2\tilde{T}\ln(1-\alpha_F) - \tilde{T}\ln(1-\rho) - \ln(1-\lambda) + \ln(24),$$

where $\tilde{T} = \left\lceil \log_{\frac{\lambda}{1-\alpha_F}}\left(\frac{1-\alpha_F}{4(n-f)}\right)\right\rceil \left(\log_{\frac{\lambda}{1-\alpha_F}}(1-\alpha_F) + 2\right) + 2$. Finally, substituting (6) in the above, and upper bounding α_F as per Section 5.1 we get the expected result.

7 Trade-off: Dissemination time vs. privacy

Note that when the gossip protocol parameter ρ decreases, the privacy guarantees in Theorem 8 improve. Yet, this worsens the dissemination time, which suggests the existence of a *trade-off* between the dissemination time and the source anonymity of the protocol. In this section, we formalize this observation by showing the tightness of Theorem 8 on a family of strong expanders called *near-Ramanujan graphs*. Intuitively, for dense enough graph topologies, most terms in Theorem 8 vanish, hence considerably simplifying the analysis of the result. Near-Ramanujan graphs can be defined as follows.

▶ **Definition 9** (Near-Ramanujan family of graphs). *Let $\mathcal{G}$ be an infinite family of regular graphs. $\mathcal{G}$ is called near-Ramanujan if there exists a constant $c > 0$ such that $\lambda(G) \leq cd(G)^{-1/2}$ for any graph $G \in \mathcal{G}$ of large enough size.*

This choice of graph family is motivated by the fact that near-Ramanujan graphs naturally arise in the study of dense random regular graphs. In fact, for any large enough n and any $3 \leq d \leq n/2$ (with dn even) a random d-regular graph on n nodes is near-Ramanujan with high probability as shown in [10, 52]. That means that almost every d-regular graph is near-Ramanujan. Besides using near-Ramanujan graphs, we assume the topologies to be dense enough, i.e., $d \in n^{\Omega_n(1)}$. Refining the statement of Theorem 8 to this family of graphs, we obtain the following corollary.

▶ **Corollary 10.** *Let $\mathcal{P}$ be a $(1+\rho)$-cobra walk and let $\mathcal{G}$ be a family of d-regular near-Ramanujan graphs with n nodes and $d \in n^{\Omega_n(1)}$. Suppose $f/d \in 1 - \Omega_n(1)$ (resp. $f/n \in 1 - \Omega_n(1)$). Then, for any $G \in \mathcal{G}$ of large enough size n and any $\rho \in 1 - \Omega_n(1)$, $\mathcal{P}$ satisfies ε-DP against a worst-case adversary (resp. an average-case adversary) for some*

$$\varepsilon \in \ln\left(\rho(n-f) + f\right) + O_n(1).$$

The above statement also holds if $\mathcal{P}$ is a ρ-Dandelion protocol with $\rho < 1$.

From Corollary 10, when $\rho = 0$, we obtain a level of differential privacy that matches, up to an additive constant, the universal lower bound $\varepsilon \geq \ln(f-1)$. Accordingly, $\rho = 0$ leads to an *optimal* differential privacy guarantee. However, in this case, both the cobra walk and the Dandelion protocol degenerate into simple random walks with dissemination time in $\Omega_n(n\log(n))$ [2]. Increasing ρ parameter makes the dissemination faster, but potentially worsens the privacy guarantees.

Studying Dandelion and cobra walks, we show that the result in Corollary 10 is tight up to an additive constant. Then, we formally validate our intuition that decreasing ρ increases the dissemination time by providing corresponding tight guarantees on dissemination time. Finally, to put our results in perspective, we compare them to a random walk (optimal privacy but high dissemination time), and to a 2-cobra walk (optimal dissemination time with bad, completely vacuous, privacy guarantees). We summarize our findings for both worst-case and average-case adversaries in Table 1 and the detailed analysis can be found in Appendix D of the full version of the paper [28].

8 Summary & future directions

This paper presents an important step towards quantifying the inherent level of source anonymity that gossip protocols provide on general graphs. We formulate our results through the lens of differential privacy. First, we present a universal lower bound on the level of

Table 1 Summary of the tension between differential privacy of a $(1+\rho)$-cobra walk and Dandelion gossip and their dissemination time on dense near-Ramanujan graphs. Graphs have diameter D and consist of n nodes, f of which are curious. Note that the upper bounds on ε hold under assumptions in Corollary 10. Lower bounds on ε hold assuming $f/n \in 1 - \Omega_n(1)$, and for cobra walk we also assume $f \in n^{\Omega_n(1)}$. Dissemination time bounds for cobra walk and Dandelion hold for $\rho \in \omega_n\left(\sqrt{\log(n)/n}\right)$ and $\rho \in \Omega_n(1/n)$ respectively.

Protocol	Privacy (ε)	Dissemination time	References
Random walk	$\ln(f) + \Theta_n(1)$	$\Theta_n(n \log(n))$	Corollary 10, Theorem 5, [2]
ρ-Dandelion	$\ln(\rho(n-f)+f) + \Theta_n(1)$	$\Theta_n\left(\frac{1}{\rho} + D\right)$	Corollary 10, Theorem 45 [28], Theorem 49 [28]
$(1+\rho)$-Cobra walk	$\ln(\rho(n-f)+f) + \Theta_n(1)$	$O_n\left(\frac{\log(n)}{\rho^3}\right)$, $\Omega_n\left(\frac{\log(n)}{\rho}\right)$	Corollary 10, Theorem 32 [28], Theorem 44 [28]
2-Cobra walk	$\ln(n) + \Omega_n(1)$	$\Theta_n(\log(n))$	Theorem 32 [28], Theorem 44 [28]

differential privacy an arbitrary gossip protocol can satisfy. Then, we devise an in-depth analysis of the privacy guarantees of $(1+\rho)$-cobra walk and ρ-Dandelion protocols on expander graphs. When $\rho = 0$, the protocols spread the gossip via a random walk, which achieves optimal privacy, but has poor dissemination time. On the other hand, we show that increasing ρ improves the dissemination time while the privacy deteriorates. In short, our tight analysis allows to formally establish the trade-off between dissemination time and the level of source anonymity these protocols provide. An interesting open research question would be to establish whether this "privacy vs dissemination time" trade-off is fundamental or if there exists a class of gossip protocols that could circumvent this trade-off.

We consider differential privacy, because, unlike other weaker notions of privacy (e.g., MLE-based bounds), it can be applied against an *arbitrary* strategy of the adversary, factoring in *any* prior beliefs an adversary may have about the location of the source and the nature of the gossip protocol. This makes differential privacy strong and resilient. However, differential privacy is often criticized for being too stringent in some settings. Consequently, a number of possible interesting relaxations have been proposed in the literature such as Pufferfish [36] and Renyi differential privacy [44]. Adapting our analysis to these definitions constitutes an interesting open direction as it would enable consideration of less stringent graphs structures and probability metrics.

Finally, we believe that our results could be applied to solve privacy related problems in other settings. For example, it was recently observed in [13] that sharing sensitive information via a randomized gossip can amplify the privacy guarantees of some learning algorithms, in the context of privacy-preserving decentralized machine learning. However, this work only considers the cases when the communication topology is a clique or a ring. We believe that the techniques we develop in this paper can be useful to amplify privacy of decentralized machine learning on general topologies. This constitutes an interesting open problem.

References

1 Huseyin Acan, Andrea Collevecchio, Abbas Mehrabian, and Nick Wormald. On the push&pull protocol for rumor spreading. *SIAM Journal on Discrete Mathematics*, 31(2):647–668, 2017. doi:10.1145/2767386.2767416.

2 David J. Aldous. Lower bounds for covering times for reversible markov chains and random walks on graphs. *Journal of Theoretical Probability*, 2:91–100, 1989.

3 Yeganeh Alimohammadi, Christian Borgs, and Amin Saberi. Algorithms using local graph features to predict epidemics. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2022)*, 2022. doi:10.1137/1.9781611977073.136.

4 Amos Beimel and Shlomi Dolev. Buses for anonymous message delivery. *Journal of Cryptology*, 16(1), 2003. doi:10.1007/s00145-002-0128-6.

5 Aurélien Bellet, Rachid Guerraoui, and Hadrien Hendrikx. Who started this rumor? Quantifying the natural differential privacy of gossip protocols. In *International Symposium on Distributed Computing (DISC 2020)*, 2020. doi:10.4230/LIPIcs.DISC.2020.8.

6 Petra Berenbrin, George Giakkoupis, and Peter Kling. Tight bounds for coalescing-branching random walks on regular graphs. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2018)*, 2018.

7 Shaileshh Bojja Venkatakrishnan, Giulia Fanti, and Pramod Viswanath. Dandelion: Redesigning the bitcoin network for anonymity. In *Proceedings of the ACM on Measurement and Analysis of Computing Systems (SIGMETRICS 2017)*, 2017. doi:10.1145/3078505.3078528.

8 Stephen Boyd, Arpita Ghosh, Balaji Prabhakar, and Devavrat Shah. Randomized gossip algorithms. *IEEE Transactions on Information Theory*, 52(6):2508–2530, 2006. doi:10.1109/TIT.2006.874516.

9 Yun Chai, Youguo Wang, and Liang Zhu. Information sources estimation in time-varying networks. *IEEE Transactions on Information Forensics and Security*, 16:2621–2636, 2021. doi:10.1109/TIFS.2021.3050604.

10 Nicholas A. Cook, Larry Goldstein, and Tobias Johnson. Size biased couplings and the spectral gap for random regular graphs. *Annals of Probability*, 46:72–125, 2018. doi:10.1214/17-AOP1180.

11 Colin Cooper, Tomasz Radzik, and Nicolas Rivera. The coalescing-branching random walk on expanders and the dual epidemic process. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing (PODC 2016)*, 2016. doi:10.1145/2933057.2933119.

12 Colin Cooper, Tomasz Radzik, and Nicolás Rivera. Improved cover time bounds for the coalescing-branching random walk on graphs. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2017)*, 2017. doi:10.1145/3087556.3087564.

13 Edwige Cyffers and Aurélien Bellet. Privacy amplification by decentralization. In *International Conference on Artificial Intelligence and Statistics (AIStat 2020)*, 2020.

14 Debajyoti Das, Sebastian Meiser, Esfandiar Mohammadi, and Aniket Kate. Anonymity trilemma: Strong anonymity, low bandwidth overhead, low latency-choose two. In *IEEE Symposium on Security and Privacy (SP)*, pages 108–126, 2018. doi:10.1109/SP.2018.00011.

15 Damien Desfontaines and Balazs Pejo. Sok: Differential privacies. In *Proceedings on Privacy Enhancing Technologies Symposium (PETS 2020)*, 2020.

16 Claudia Díaz, Stefaan Seys, Joris Claessens, and Bart Preneel. Towards measuring anonymity. In *Privacy Enhancing Technologies*, pages 54–68, 2003.

17 Benjamin Doerr, Mahmoud Fouz, and Tobias Friedrich. Social networks spread rumors in sublogarithmic time. In *Proceedings of the forty-third annual ACM symposium on Theory of computing (STOC 2011)*, 2011.

18 Chinmoy Dutta, Gopal Pandurangan, Rajmohan Rajaraman, and Scott Roche. Coalescing-branching random walks on graphs. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2013)*, 2013. doi:10.1145/2817830.

19 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988. doi:10.1145/42282.42283.

20 Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography: Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006. Proceedings 3*, pages 265–284, 2006.

21 Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3-4):211–407, 2013. doi:10.1561/0400000042.

22 Giulia Fanti, Peter Kairouz, Sewoong Oh, Kannan Ramchandran, and Pramod Viswanath. Rumor source obfuscation on irregular trees. In *International Conference on Measurement and Modeling of Computer Systems, (SIGMETRICS 2016)*, 2016. doi:10.1145/2896377.2901471.

23 Giulia Fanti, Peter Kairouz, Sewoong Oh, Kannan Ramchandran, and Pramod Viswanath. Hiding the rumor source. *IEEE Transactions on Information Theory*, 63(10):6679–6713, 2017. doi:10.1109/TIT.2017.2696960.

24 Chryssis Georgiou, Seth Gilbert, Rachid Guerraoui, and Dariusz R Kowalski. Asynchronous gossip. *Journal of the ACM*, 60(2):1–42, 2013. doi:10.1145/2450142.2450147.

25 Chryssis Georgiou, Seth Gilbert, and Dariusz R. Kowalski. Confidential gossip. In *International Conference on Distributed Computing Systems (DISC 2011)*, 2011. doi:10.1109/ICDCS.2011.71.

26 George Giakkoupis, Rachid Guerraoui, Arnaud Jégou, Anne-Marie Kermarrec, and Nupur Mittal. Privacy-conscious information diffusion in social networks. In *International Symposium on Distributed Computing (DISC 2015)*, 2015.

27 Karol Gotfryd, Marek Klonowski, and Dominik Pająk. On location hiding in distributed systems. In *Structural Information and Communication Complexity*, pages 174–192, 2017.

28 Rachid Guerraoui, Anne-Marie Kermarrec, Anastasiia Kucherenko, Rafael Pinot, and Sasha Voitovych. On the Inherent Anonymity of Gossiping (Full Version), 2023. arXiv:2308.02477.

29 Zeyu Guo and He Sun. Gossip vs. markov chains, and randomness-efficient rumor spreading. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2014)*, 2014.

30 Herbert W Hethcote. The mathematics of infectious diseases. *SIAM review*, 42(4):599–653, 2000. doi:10.1137/S0036144500371907.

31 Justin Hsu, Marco Gaboardi, Andreas Haeberlen, Sanjeev Khanna, Arjun Narayan, Benjamin C Pierce, and Aaron Roth. Differential privacy: An economic method for choosing epsilon. In *IEEE 27th Computer Security Foundations Symposium*, pages 398–410, 2014.

32 Yufan Huang, Richeng Jin, and Huaiyu Dai. Differential privacy and prediction uncertainty of gossip protocols in general networks. In *IEEE Global Communications Conference (GLOBECOM 2020)*, 2020. doi:10.1109/GLOBECOM42002.2020.9322558.

33 Jiaojiao Jiang, Sheng Wen, Shui Yu, Yang Xiang, and Wanlei Zhou. Identifying propagation sources in networks: State-of-the-art and comparative studies. *IEEE Communications Surveys & Tutorials*, 19(1):465–481, 2017. doi:10.1109/COMST.2016.2615098.

34 Richard Karp, Christian Schindelhauer, Scott Shenker, and Berthold Vocking. Randomized rumor spreading. In *41st Annual Symposium on Foundations of Computer Science (FOCS 2000)*, 2000. doi:10.1109/SFCS.2000.892324.

35 John G Kemeny and J Laurie Snell. *Finite markov chains.* Springer New York, NY, 1960.

36 Daniel Kifer and Ashwin Machanavajjhala. Pufferfish: A framework for mathematical privacy definitions. *ACM Transactions on Database Systems (TODS)*, 39(1):1–36, 2014. doi:10.1145/2514689.

37 István Z Kiss, Joel C Miller, Péter L Simon, et al. Mathematics of epidemics on networks. *Cham: Springer*, 598:31, 2017.

38 Dariusz R Kowalski and Christopher Thraves Caro. Estimating time complexity of rumor spreading in ad-hoc networks. In *International Conference on Ad-Hoc Networks and Wireless (ADHOC-NOW 2013)*, 2013.

39 Jaewoo Lee and Chris Clifton. How much is enough? Choosing ε for differential privacy. In *Information Security: 14th International Conference*, pages 325–340, 2011.

40 Friedrich Liese and Igor Vajda. On divergences and informations in statistics and information theory. *IEEE Transactions on Information Theory*, 52(10):4394–4412, 2006. doi:10.1109/TIT.2006.881731.

41 Xuecheng Liu, Luoyi Fu, Bo Jiang, Xiaojun Lin, and Xinbing Wang. Information source detection with limited time knowledge. In *Proceedings of the Twentieth ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc 2019)*, pages 389–390, 2019. doi:10.1145/3323679.3326626.

42 Yang Liu, Junfeng Wu, Ian R. Manchester, and Guodong Shi. Gossip algorithms that preserve privacy for distributed computation part I: The algorithms and convergence conditions. In *IEEE Conference on Decision and Control (CDC 2018)*, 2018. doi:10.1109/CDC.2018.8619783.

43 Guy Melancon. Just how dense are dense graphs in the real world? A methodological note. In *Proceedings of the AVI Workshop on BEyond Time and Errors: Novel Evaluation Methods for Information Visualization (BELIV 2006)*, 2006. doi:10.1145/1168149.1168167.

44 Ilya Mironov. Rényi differential privacy. In *2017 IEEE 30th computer security foundations symposium (CSF)*, pages 263–275. IEEE, 2017. doi:10.1109/CSF.2017.11.

45 Michael Mitzenmacher, Rajmohan Rajaraman, and Scott Roche. Better bounds for coalescing-branching random walks. *ACM Transactions on Parallel Computing*, 5(1):1–23, 2018. doi:10.1145/3209688.

46 Pedro C. Pinto, Patrick Thiran, and Martin Vetterli. Locating the source of diffusion in large-scale networks. *Physical Review Letters*, 109(6), 2012. doi:10.1103/PhysRevLett.109.068702.

47 Boris Pittel. On spreading a rumor. *SIAM Journal on Applied Mathematics*, 47(1):213–223, 1987. doi:10.1137/0147013.

48 Michael K Reiter and Aviel D Rubin. Crowds: Anonymity for web transactions. *ACM transactions on information and system security (TISSEC)*, 1(1):66–92, 1998. doi:10.1145/290163.290168.

49 D. Shah and T. Zaman. Rumors in a network: Who's the culprit? *IEEE Transactions on Information Theory*, 57(8):5163–5181, 2011. doi:10.1109/TIT.2011.2158885.

50 Devavrat Shah and Tauhid Zaman. Detecting sources of computer viruses in networks: Theory and experiment. In *Proceedings of ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2010)*, 2010.

51 Robin Snader and Nikita Borisov. A tune-up for tor: Improving security and performance in the tor network. In *Network and Distributed System Security Symposium (NDSS 2008)*, 2008.

52 Konstantin E. Tikhomirov and Pierre Youssef. The spectral gap of dense random regular graphs. *The Annals of Probability*, 2019.

53 Parv Venkitasubramaniam and Venkat Anantharam. Anonymity under light traffic conditions using a network of mixes. In *46th Annual Allerton Conference on Communication, Control, and Computing (ALLERTON 2008)*, 2008. doi:10.1109/ALLERTON.2008.4797721.

54 Matthew K. Wright, Micah Adler, Brian Neil Levine, and Clay Shields. An analysis of the degradation of anonymous protocols. In *Network and Distributed System Security Symposium (NDSS 2002)*, 2002.

55 Yi Yu, Tengyao Wang, and Richard J. Samworth. A useful variant of the Davis–Kahan theorem for statisticians. *Biometrika*, 102:315–323, 2014.

Durable Algorithms for Writable LL/SC and CAS with Dynamic Joining

Prasad Jayanti
Dartmouth College, Hanover, NH, USA

Siddhartha Jayanti
Google Research, Atlanta, GA, USA

Sucharita Jayanti
Brown University, Providence, RI, USA

Abstract

We present durable implementations for two well known universal primitives – CAS (compare-and-swap), and its ABA-free counter-part LLSC (load-linked, store-conditional). Our implementations satisfy *method-based recoverable linearizability (MRL)* and *method-based detectability (M-detectability)* – novel correctness conditions that require only a simple usage pattern to guarantee resilience to individual process crashes (and system-wide crashes), including in implementations with nesting. Additionally, our implementations are: *writable*, meaning they support a Write() operation; have *constant time complexity* per operation; allow for *dynamic joining*, meaning newly created processes (a.k.a. threads) of arbitrary names can join a protocol and access our implementations; and have *adaptive space complexity*, meaning the space use scales in the number of processes n that actually use the objects, as opposed to previous protocols whose space complexity depends on N, the maximum number of processes that the protocol is designed for. Our durable Writable-CAS implementation, DuraCAS, requires $O(m+n)$ space to support m objects that get accessed by n processes, improving on the state-of-the-art $O(m+N^2)$. By definition, LLSC objects must store "contexts" in addition to object values. Our Writable-LLSC implementation, DuraLL, requires $O(m+n+C)$ space, where C is the number of "contexts" stored across all the objects. While LLSC has an advantage over CAS due to being ABA-free, the object definition seems to require additional space usage. To address this trade-off, we define an *External Context (EC)* variant of LLSC. Our EC Writable-LLSC implementation is ABA-free and has a space complexity of just $O(m+n)$.

To our knowledge, our algorithms are the first durable CAS algorithms that allow for dynamic joining, and are the first to exhibit adaptive space complexity. To our knowledge, we are the first to implement any type of *durable* LLSC objects.

2012 ACM Subject Classification Computer systems organization → Dependable and fault-tolerant systems and networks; Theory of computation → Concurrency; Theory of computation → Concurrent algorithms; Theory of computation → Parallel algorithms; Theory of computation → Data structures design and analysis; Theory of computation → Distributed algorithms

Keywords and phrases durable, recoverable, detectable, persistent memory, dynamic joining, LL/SC, CAS

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.25

Related Version *Full Version*: `https://arxiv.org/abs/2302.00135` [30, 34]

Funding Prasad Jayanti was supported in part by the James Frank Family Professorship. Siddhartha Jayanti was supported in part by an NDSEG Graduate Research Fellowship.

1 Introduction

The advent of *Non-Volatile Memory (NVM)* [27] spurred the development of durable algorithms for the *crash-restart model*. In this model, when a process π crashes, the contents of memory *persist* (i.e., remain unchanged), but π's CPU registers, including its program counter, lose their contents. To understand the difficulty that arises from losing register contents, suppose that π crashes at the point of executing a hardware CAS instruction, $r \leftarrow \textsc{Cas}(X, \mathit{old}, \mathit{new})$, on a memory word X and receiving the response into its CPU register r. When π subsequently restarts, π cannot tell whether the crash occurred before or after the CAS executed, and if the crash occurred after the CAS, π cannot tell whether the CAS was successful or not. Researchers identified this issue and proposed software-implemented *durable objects* [28, 5], which allow a restarted process to *recover* from its crash and *detect* the result of its last operation. The rapid commercial viability of byte-addressable, dense, fast, and cheap NVM chips has made efficient durable object design important.

Writable and non-Writable CAS. The Compare-and-Swap (CAS) instruction is ubiquitous in multiprocessor computation, both in concurrent and parallel algorithms. Recently, there has been a lot of research on implementing durable CAS objects because they are widely employed in practice and are universal: any durable object can be implemented from durable CAS objects [7, 25, 28]. Formally, the state of a CAS object X is simply its value, and the operation semantics are as follows:

- $X.\textsc{Cas}(\mathit{old}, \mathit{new})$: if $X = \mathit{old}$, sets X to *new* and returns *true*; otherwise, returns *false*.
- $X.\textsc{Read}()$: returns the value of X.
- $X.\textsc{Write}(\mathit{new})$: sets X to *new* and returns *true*.

If the object supports all three operations, it is a *Writable-CAS (W-CAS)*; and if it does not support $\textsc{Write}()$, it is a *non-Writable-CAS (nW-CAS)* object.

CAS's ABA problem and LLSC. Although CAS objects are powerful tools in concurrent computing, they also have a significant drawback called the *ABA-problem* [16]. Namely, if a process π reads a value A in X and executes $X.\textsc{Cas}(A, C)$ at a later time, this CAS will succeed *even if* the value of X changed between π's operations, from A to B and then back to A. So while any object *can* be implemented from CAS, the actual process of designing an algorithm to do so becomes difficult. In the non-durable setting, the ABA-problem is often overcome by using the hardware's double-width CAS primitive – in fact, "CAS2 [double-width CAS] operation is the most commonly cited approach for ABA prevention in the literature" [16]. However, all known durable CAS objects, including ours, are only one-word wide – even as they use hardware double-width CAS [5, 7, 6]. Against this backdrop, the durable LLSC objects presented in this paper serve as an invaluable alternate tool for ABA prevention.

LLSC objects are alternatives to CAS objects that have been invaluable in practice, since they are universal and ABA-free [38]. The state of an LLSC object Y is a pair $(Y.\mathit{val}, Y.\mathit{context})$, where $Y.\mathit{val}$ is the *value* and $Y.\mathit{context}$ is a set of processes (initially empty). Process π's operations on the object have the following semantics:

- $Y.\text{LL}()$: adds π to $Y.\mathit{context}$ and returns $Y.\mathit{val}$.
- $Y.\text{VL}()$: returns whether $\pi \in Y.\mathit{context}$.
- $Y.\text{SC}(\mathit{new})$: if $\pi \in Y.\mathit{context}$, sets Y's value to *new*, resets $Y.\mathit{context}$ to the empty set and returns *true*; otherwise, returns *false*.
- $Y.\textsc{Write}(\mathit{new})$ changes Y's value to *new* and resets $Y.\mathit{context}$ to the empty set.

The object is *Writable (W-LLSC)* or *non-Writable (nW-LLSC)* depending on whether the Write() operation is supported.

To our knowledge, there are no earlier durable implementations of ABA-free CAS-like objects, including LLSC.

Wider impact of durable primitives. Durable primitives such as W-CAS and W-LLSC are particular important since they facilitate a plethora of other durable data structures. In particular, let A be an algorithm for implementing a data structure DS using either the read, write, CAS, or the read, write, LL/SC/VL instructions. Then, using durable W-CAS and durable W-LLSC, we can design an algorithm A' that implements a *durable* version of DS from hardware read, write, and CAS, without affecting the asymptotic time or space complexity [5, 7].

Previous work and the state-of-the-art. CAS and LLSC objects share close ties, but they also pose different implementational challenges. In the non-durable context, it is well known that non-writable LLSC (nW-LLSC) objects can be implemented from nW-CAS objects and visa versa in constant time and space. The simple implementation of nW-LLSC from nW-CAS however, requires packing a value-context pair into a single nW-CAS object [3]. Solutions that implement a full-word nW-LLSC from a full-word nW-CAS require a blow-up in time complexity, space complexity, or both [37, 18, 40, 38, 11]. Writability complicates the relationship further. Even in the non-durable context, reductions between W-CAS and W-LLSC have resulted in a blow-up in space complexity and fixing the number of processes *a priori* [29]. Writability can sometimes be added to an object that is non-writable, but this leads to an increase in space complexity [1].

There are no previous works on Durable LLSC. Three previous works have implemented durable CAS objects, all from the hardware CAS instruction: Attiya, Ben-Baruch, and Hendler [5], Ben-Baruch, Hendler, and Rusanovsky [6], and Ben-David, Blelloch, Friedman, and Wei [7]. All three papers provide implementations for a fixed set of N processes with *pid*s $1, \ldots, N$, and achieve constant time complexity per operation. Attiya et al. pioneered this line of research with a durable nW-CAS implementation, which achieves constant time complexity and requires $O(N^2)$ space per object. Ben-Baruch et al. present an nW-CAS implementation with optimal bit complexity. Their algorithm however, requires packing N bits and the object's value into a single hardware variable. Thus, if the value takes 64 bits, then only 64 pre-declared processes can access this object. (Current commodity multiprocessors range up to 224 cores [24], and can support orders-of-magnitude more threads.) Ben-David et al. designed an algorithm for nW-CAS, and then leveraged Aghazadeh, Golab, and Woelfel's writability transformation [1] to enhance that algorithm to include a Write operation, thereby presenting the only previous Writable-CAS implementation. Their nW-CAS algorithm uses a pre-allocated help-array of length $O(N)$, and their W-CAS algorithm uses an additional hazard-pointer array of length $O(N^2)$. Both arrays can be shared across objects, thus the implementation space complexities for m objects are $O(m + N)$ and $O(m + N^2)$, respectively.

Our contributions. We present four wait-free, durable implementations: DuraCAS for Writable-CAS, DuraLL for Writable-LLSC, DurEC for External Context (EC) nW-LLSC, and DurECW for EC W-LLSC (we will specify External Context LL/SC soon). Our implementations achieve the following properties:

1. Constant time complexity: All operations including recovery and detection run in $O(1)$ steps.
2. Dynamic Joining: Dynamically created processes of arbitrary names can use our objects.
3. Full-word size: Our implementations support full-word (i.e., 64-bit) values.
4. Adaptive Space Complexity: We quantify space complexity by the number of memory words needed to support m objects for a total of n processes. The DuraCAS, DurEC, and DurECW implementations require just constant memory per process and per object, and thus each have a space complexity of $O(m+n)$. Since DuraLL must remember contexts, its space complexity is $O(m+n+C)$, where C is the number of contexts that must be remembered.[1]

We believe that our definitions and implementations of the External Context LLSC objects – which are ABA-free, space-efficient alternatives to CAS and LLSC – are of independent interest in the design of both durable and non-durable concurrent algorithms.

To our knowledge, our algorithms are the first durable CAS algorithms that allow for dynamic joining, and are the first to exhibit adaptive space complexity. To our knowledge, we are the first to consider any type of *durable* LLSC objects.

Our approach. We implement universal primitives that allow dynamic joining of new processes, have an *adaptive space complexity* that is constant per object and per process, and give an ABA-free option, while simultaneously achieving constant time complexity. Just like our predecessors, all our implementations rely on just the hardware double-width CAS instruction for synchronization.

A keystone of our approach is the observation that durable nW-LLSC – due to its ABA-freedom – serves as a better stepping stone than even durable nW-CAS on the path from hardware CAS to durable W-CAS. Perhaps less surprisingly, durable nW-LLSC is a great stepping stone towards durable W-LLSC also. However, by definition LLSC objects require more space to remember context for each process – an inherent burden that CAS objects do not have. Thus, using nW-LLSC objects in the construction of our W-CAS would lead to a bloated space complexity. To avoid this drawback, we define an *External Context (EC)* variant of LLSC. An EC LLSC object is like an LLSC object, except that its context is returned to the process instead of being maintained by the object. Thus, our EC nW-LLSC implementation, DurEC, is the building block of all our other implementations.

The state of an EC LLSC object Y is a pair $(Y.val, Y.seq)$, where the latter is a sequence number context. Process π's operations on the object have the following semantics:

- Y.ECLL(): returns $(Y.val,\ Y.seq)$.
- Y.ECVL(s): returns whether $Y.seq = s$.
- Y.ECSC(s, new): if $Y.seq = s$, sets Y's value to new, increases $Y.seq$, and returns *true*; otherwise, returns *false*.
- Y.Write(new): changes Y's value to new and increases $Y.seq$.

The object is *Writable (EC W-LLSC)* or *non-Writable (EC nW-LLSC)* depending on whether the Write() operation is supported.

We design durable implementations of External Context W-LLSC and W-CAS, called DurECW and DuraCAS, respectively; each implementation uses two DurEC base objects. We implement our durable W-LLSC algorithm, DuraLL, by simply internalizing the external

[1] C is the number of process-object pairs $(\pi, \mathcal{O})$, where π has performed an LL() operation on $\mathcal{O}$, and its last operation on $\mathcal{O}$ is not an SC() or Write(). A trivial upper bound is $C \leq nm$.

contexts of a DurECW. All our implementations overcome the need for hazard-pointers and pre-allocated arrays for helping in order to allow dynamic joining and achieve adaptive space complexity. Key to eliminating these arrays are pointer based identity structures called *handles*, which we describe in Section 4. Figure 1 illustrates the differences between our approach and Ben-David et al.'s.

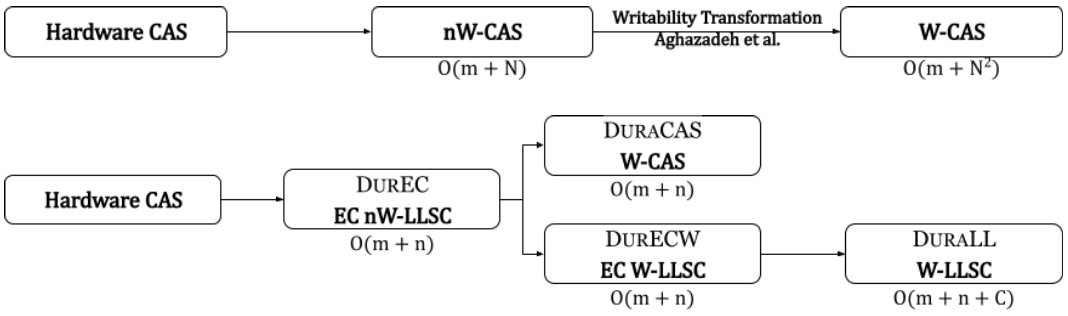


Figure 1 A comparison of Ben-David et al.'s approach (top) and our approach (bottom): each box represents an implementation – the type of the implementation is in bold and its space complexity appears below the box. The names of our implementations appear in the box in SmallCaps. An arrow from A to B means that B is implemented using A.

2 Related Work

We have already detailed the three previous works on durable CAS objects [5, 6, 7] in the introduction. In addition to these durable (single-word) CAS objects, there are implementations of durable multi-word CAS objects[2] by Wang, Levandoski, and Larson [42], and by Guerraoui, Kogan, Marathe, and Zablotchi [22]. Furthermore, LLSC can be implemented using multi-word CAS. However, these software implementations of multi-word CAS are lock-free but not wait-free, and they do not support the Write operation. Thus, using these algorithms, one can implement non-writable lock-free LL/SC, but not the writable and wait-free LL/SC primitive that our algorithm implements. Additionally, they require complex memory management which also leads to an increase in space complexity. We discuss other related work below.

Byte-addressable non-volatile memory laid the foundation for durable objects [27]. Research on durable objects has spanned locks [21, 41, 35, 36, 33, 31, 20, 12, 14, 17, 15, 32], and non-blocking objects – including queues [19], counters [5], registers [5, 6], CAS objects [5, 6, 7], and general transformations and universal constructions [28, 7, 4].

Several models of persistent memory systems have been proposed in the literature. In the *individual-crash model* [2, 5, 7, 10], processes can crash independently, while in the *system-crash model*, all processes crash together [28, 9, 19]. Izraelevitz et al. assume that crashed processes do not restart and the system spawns new processes with process ids that were never used before [28], but most other works assume processes restart with the same ids as before. Some works, such as [28, 19], model volatile caches: when a process performs

[2] A k-word CAS, $\text{Cas}((X_1, \ldots, X_k), (old_1, \ldots, old_k), (new_1, \ldots, new_k))$, has the semantics: if all of $X_1 = old_1, \ldots, X_k = old_k$ then set $X_1 \leftarrow new_1, \ldots, X_k \leftarrow new_k$ and return *true*; otherwise, simply return *false*.

a hardware operation on a shared variable, only the cached copy of the variable is updated, and the update transfers to the NVM only when the cache is explicitly flushed. Others work in the model that each step directly updates the variable's state in the NVM [5, 7, 21].

Several correctness criteria have been proposed for implementations: *recoverable linearizability* by Berryhill, Golab, and Tripunitara [9], *durable and buffered durable linearizability* by Israelevitz, Mendes, and Scott [28], *nested recoverable linearizability* by Attiya, Ben-Baruch, and Hendler [5], *persistent atomicity* by Guerraoui and Levy [23], and *strict linearizability* by Aguilera and Frølund [2]. These consistency criteria are surveyed by Ben-David, Friedman, and Wei [8].

Friedman, Herlihy, Marathe, and Petrank [19] identify that it is not enough for implementations to satisfy a linearizability condition, but they must also support *detection*, i.e., make possible for a process to find out whether its crashed operation took effect and, if it did, what the response was. Li and Golab [39] present a formulation of detectability.

3 Model

Our model is akin to those used in previous works on durable CAS [5, 7]. The system consists of asynchronous processes that communicate by applying atomic operations (Read or CAS) directly to shared variables stored in Non-Volatile Memory (NVM). We use the individual crash model where any process may crash at any time and restart at any later time, and the same process may crash and restart any number of times. When a process crashes, its registers, including its program counter, lose their contents (i.e., they are set to arbitrary values), but the contents of the NVM are unaffected. After a crash, a process eventually restarts (with the same process id as before).

To ensure that our objects are recoverable and detectable, we introduce two new correctness conditions for objects, called *Method-based Recoverable Linearizability (MRL)* and *Method-based Detectability (M-Detectability)*. MRL adapts and combines ideas from the well known notions of Recoverable Linearizability [9] and Nested-safe Recoverable Linearizability [5], and M-Detectability captures the corresponding notion of detectability [19].

MRL is "method-based" in the sense that it facilitates recoverability by requiring that an object $\mathcal{O}$ provide a method $\mathcal{O}.\textsc{Recover}()$ in addition to providing a method for each operation supported by $\mathcal{O}$. When there are no crashes, MRL reduces to standard linearizability [26]. In particular, if a process π invokes a method for an operation and completes the method without crashing, the operation is required to take effect atomically at some instant between the method's invocation and completion. On the other hand, if crashes occur, MRL guarantees the object remains consistent if the following *usage pattern* if followed. If π crashes after invoking some operation $\mathcal{O}.op$ and before that operation completes, when π subsequently restarts, the usage pattern requires that π execute $\mathcal{O}.\textsc{Recover}()$ before invoking any other operation on object $\mathcal{O}$ (if π crashes while executing $\mathcal{O}.\textsc{Recover}()$, it must execute $\mathcal{O}.\textsc{Recover}()$ again after restarting before invoking any other operation on $\mathcal{O}$); when π completes $\mathcal{O}.\textsc{Recover}()$, we deem $\mathcal{O}.op$ completed. If the usage pattern is observed, MRL guarantees that the crashed operation $\mathcal{O}.op$ either never takes effect or takes effect at some point between $\mathcal{O}.op$'s invocation and completion. Notice that the usage pattern allows π to perform any number of other operations on objects other than $\mathcal{O}$ upon restart and even allows for π to never perform any subsequent operation on $\mathcal{O}$; the only requirement is that π calls $\mathcal{O}.\textsc{Recover}()$ before calling any other method on $\mathcal{O}$.

MRL's relationship to Recoverable Linearizability (RL) and Nested-safe Recoverable Linearizability (NRL) can be understood as follows. Just like MRL, RL requires that a crashed operation by process π on object $\mathcal{O}$ either does not take effect at all or takes effect before π's next invocation of an operation on $\mathcal{O}$. However, RL does not specify a mechanism by which π's operation can complete after its crash and before its subsequent operation starts. MRL is similar, but uses the mechanism of the recover method to complete (or ensure non-completion of) crashed operations. The NRL paper uses recover methods. However, in the NRL model each method has its own recover method, recover methods take arguments, and it is assumed that upon a crash, a recover method is called with the same arguments as the corresponding method that failed and the recover method has access to a process specific persistent register which stores the program counter value right before the crash. In contrast to NRL, MRL has only a single recover method, and most importantly, makes no assumptions about method arguments or program counter values being supplied to restarted processes; it simply guarantees that objects remain consistent if the usage pattern is followed (however an implementation may follow it).

Friedman et al. [19] first made the observation that in addition to proper recovery, it is necessary for a process to be able to *detect* whether its crashed operation took effect, and if so, what its return value was. We ensure detectability of our operations in a method-based manner, by requiring that an object $\mathcal{O}$ provide an additional DETECT() method, which returns this information. We note that some operations, such as read or a failed CAS, can safely be repeated, regardless of whether they took effect [5, 7], whereas, a write or a successful CAS that changed the value of the object cannot be repeated safely. Our implementations provide a DETECT() method which guarantees that all unsafe-to-repeat operations are detected along with their responses, and that any operation that is not detected is safe to repeat. In particular, DETECT() returns a pair that satisfies the following property:

Method-based Detectability: If a process calls DETECT() twice – just before executing an operation and just after completing that (possibly crashed[3]) operation – and these successive calls to DETECT() return (d_1, r_1) and (d_2, r_2) respectively, then the following two conditions are satisfied:

1. If $d_2 > d_1$, then the operation took effect and its response is r_2.
2. Otherwise, $d_1 = d_2$ and the operation is safe to repeat.

A *durable* object is one that satisfies method-based recoverable linearizability and method-based detectability.

4 Handles for dynamic joining and space adaptivity

When a process calls a method to execute an operation *op*, the call is of the form $op(p, args)$, where *args* is a list of *op*'s arguments and p identifies the calling process. The methods use p to facilitate helping between processes. In many algorithms, the processes are given *pid*s from 1 to N, and p is the *pid* of the caller [5, 7]. In particular, p is used to index a pre-allocated helping array – in Ben-David et al.'s algorithm this helping array is of length N, one location per process being helped; in Attiya et al.'s algorithm this helping array is of length N^2, one location per helper-helpee pair. Helping plays a central role in detection, thus each process needs to have some area in memory where it can be helped; in fact, using the bit-complexity model, Ben-Baruch et al. proved that the space needed to support a detectable CAS object monotonically increases in the number of processes that access the object [6]. One of our goals in this paper however, is to design objects that can be accessed by a dynamically increasing set of processes, which precludes the use of pre-allocated fixed-size arrays that are indexed by process IDs.

[3] Recall that a crashed operation by π *completes* when π finishes RECOVER() following the crash.

To eliminate the use of arrays for helping, we introduce pointer based structures called *handles*. We use handles to enable dynamic joining and achieve space adaptivity. A handle is a pointer to a constant sized record. The implementation provides a CreateHandle() method, which allocates memory for a new record and returns a pointer h to it called a *handle*. This allocation can be via a persistent memory allocator [13] or any other means that ensures that handles are created in constant time and remembered across crashes without any memory leaks. When a process first wishes to access any of the implemented objects of a given type, it creates for itself a new handle by calling CreateHandle(). From that point on, whenever the process calls any method on any of the implemented objects of that type, it passes its handle h instead of its *pid*, and other processes help it via the handle. This mechanism of handles helps us realize dynamic joining because any number of new processes can join at any time by creating handles for themselves; since the memory per handle is constant, and only the subset of processes that wish to access the implementation need to create handles, the mechanism facilitates space adaptivity.

At first glance, replacing *pid*s with handles to achieve dynamic joining may seem like a simple level of indirection; however, this step actually poses a significant algorithmic challenge. As will become more clear in the next section, the challenge arises from the fact that known algorithms for durable primitives (including ours) must, in principle, store three pieces of information consistently using hardware that only supports double-width CAS. The three pieces to store are the implemented object's value, its sequence number, and the *pid*/handle of the process that last changed the object. (Intuitively, the sequence number helps styme ABA problems, while the handle facilitates helping.) Previous durable CAS algorithms either require the strong assumption that processes never attempt to CAS in the same value twice to avoid sequence numbers [5]; or assume that a *pid* and a sequence number can be packed into a single pointer-sized word [7]. Since handles are pointers, and the object's value can also be pointer-sized, we cannot pack all three pieces of information into a single double-width word. Our DurEC algorithm overcomes this challenge by storing the sequence number with the handle in one variable, and the same sequence number with the value in another variable, and cleverly coordinating between these two variables to create the illusion that all three pieces of information are stored and manipulated atomically together.

5 The DurEC Building Block

In this section, we implement the DurEC algorithm for durable external context non-writable LLSC using hardware CAS. This building block will be central to all of the writable implementations in the remainder of the paper.

Intuitive description of Algorithm DurEC. Each DurEC handle h is a reference to a record of two fields, *Val* and *DetVal*, and each DurEC object $\mathcal{O}$ is implemented from two hardware atomic CAS objects X and Y, where X is a pair consisting of a handle and a sequence number, and Y is a pair consisting of a sequence number and a value. The algorithm maintains the DurEC object $\mathcal{O}$'s state in Y, i.e., $\mathcal{O}.seq = Y.seq$ and $\mathcal{O}.val = Y.val$ at all times. This representation makes the implementation of ECLL and ECVL operations obvious: ECLL(h) simply returns Y and ECVL(h, s) returns whether $Y.seq = s$ (although ECLL and ECVL do not use the handle parameter h, for uniformity we let the handle be the first parameter of all object operations). The complexity lies in the ECSC(h, s, v) operation, which is implemented by the following sequence of steps:

1. If $Y.seq \neq s$, it means $\mathcal{O}.seq \neq s$, so the ECSC operation simply returns *false*. Otherwise, it embarks on the following steps, in an attempt to switch $\mathcal{O}.val$ to v and $\mathcal{O}.seq$ to a greater number.
2. Make v available for all by writing it in the *Val* field of the ECSC operation's handle h.
3. Pick a number $\hat{s}$ that is bigger than both $X.seq$ and $h.DetVal$. (The latter facilitates detection.)
4. Publish the operation's handle along with a greater sequence number by installing $(h, \hat{s})$ in X. If several ECSC operations attempt to install concurrently, only one will succeed. The successful one is the *installer* and the others are *hitchhikers*.
5. The installer and the hitchhikers work together to accomplish two missions, the first of which is to increase the installer's *DetVal* field to the number in $X.seq$. This increase in the *DetVal* field of its handle enables the installer to detect that it installed, even if the installer happens to crash immediately after installing.
6. The second mission is to forward the installer's operation to Y. Since Y is where the DurEC object's state is held, the installer's operation takes effect only when it is reflected in Y's state. Towards this end, everyone reads the installer's value v, made available in the *Val* field of the installer's handle back at Step (2), and attempts to switch $Y.val$ to v, simultaneously increasing $Y.seq$ so that it catches up with $X.seq$. Since all operations attempt this update of Y, someone (not necessarily the installer) will succeed. At this point, $X.seq = Y.seq$ and $Y.val = v$, which means that the installer's value v has made its way to $\mathcal{O}.val$. So, the point where Y is updated becomes the linearization point for the installer's successful ECSC operation. The hitchhikers are linearized immediately after the installer, which causes their ECSC operations to "fail" – return *false*, without changing $\mathcal{O}$'s state – thereby eliminating the burden of detecting these operations.
7. If the installer crashes after installing, upon restart, in the Recover method, it does the forwarding so that the two missions explained above are fulfilled.
8. With the above scheme, all ECSC, ECLL, and ECVL operations, except those ECSC operations that install, are safe to repeat and hence, don't need detection. Furthermore, for each installing ECSC operation, the above scheme ensures that the *DetVal* field of the installer's handle is increased, thereby making the operation detectable.

The formal algorithm is presented in Figure 1. The correspondence between the lines of the algorithm and the steps above is as follows. Lines 6 and 7 implement Steps 1 and 2, respectively. Steps 3 and 4, where the operation attempts to become the installer, are implemented by Lines 8 to 10. The operation becomes the installer if and only if the CAS at Line 10 succeeds, which is reflected in the boolean return value r. The Forward method is called at Line 11 to accomplish the two missions described above. The first three lines of Forward (Lines 13 to 15) implement the first mission of increasing the *DetVal* field of the installer's handle to $X.seq$ (Step 5). Line 13, together with Lines 16 to 19, implement the second mission of forwarding the operation to Y (Step 6). The if-condition and the CAS' arguments at Line 18 ensure that Y is changed only if $Y.seq$ lags behind $X.seq$ and, if it lags behind, it catches up and $Y.val$ takes on the installer's value. The Recover method simply forwards at Line 20, as explained in Step 7. The detect method returns at Line 22 the value in the handle's *DetVal* field, as explained in Step 8, along with *true* (since only successful ECSC operations are detected).

The theorem below summarizes the result:

▶ **Theorem 1.** *Algorithm* DurEC *satisfies the following properties:*

1. *The objects implemented by the algorithm are durable, i.e., they satisfy method-based recoverable linearizability and method-based detectability (with respect to the sequential specification of External Context LLSC).*
2. *All object operations including Recover are wait-free and run in constant time.*
3. *The algorithm supports dynamic joining: a new process can join in at any point in a run (by calling CreateHandle) and start creating* DurEC *objects or accessing existing* DurEC *objects.*
4. *The space requirement is $O(m + n)$, where m is the actual number of* DurEC *objects created in the run, and n is the actual number of processes that have joined in in a run.*

▶ Remark. It is interesting to note that while DurEC is *recoverably linearizable*, it is not *strictly lineariable*, i.e., operations may linearize after a process crashes (but before its next successful completion of Recover()). In particular, if a process crashes after successfully CASing into X and before changing Y, then this operation has not taken effect before the crash, but is guaranteed to take effect in the future (i.e., after the crash).

6 DurECW and DuraLL: durable Writable LLSC implementations

Using DurEC, we design the *writable* external context LLSC implementation DurECW in this section. With DurECW in hand, we obtain our standard durable writable-LLSC implementation DuraLL easily, by simply rolling the context into the object.

6.1 Intuitive description of Algorithm DurECW

A DurECW object $\mathcal{O}$ supports the write operation, besides ECSC, for changing the object's state. Unlike a ECSC(h, s, v) operation, which returns without changing $\mathcal{O}$'s state when $\mathcal{O}.context \neq s$, a Write$(h, v)$ must get v into $\mathcal{O}.val$ unconditionally. In the DurECW algorithm, ECSC() operations help Write() operations and prevent writes from being blocked by a continuous stream of successful ECSC() operations.

Each DurECW object $\mathcal{O}$ is implemented from two DurEC objects, $\mathcal{W}$ and $\mathcal{Z}$, each of which holds a pair, where the first component is a sequence number *seq*, and the second component is a pair consisting of a value *val* and a bit *bit*. Thus, $\mathcal{W} = (\mathcal{W}.seq, (\mathcal{W}.val, \mathcal{W}.bit))$ and $\mathcal{Z} = (\mathcal{Z}.seq, (\mathcal{Z}.val, \mathcal{Z}.bit))$.

The DurECW handle h consists of two DurEC handles, $h.Critical$ and $h.Casual$. The use of two DurEC handles allows us to implement detectability. In particular, if Detect(h) is called on a DurECW object, only the detect value (*DetVal*) of $h.Critical$ is returned. So intuitively, when a DurECW operation α calls methods on $\mathcal{W}$ or $\mathcal{Z}$, it uses $h.Critical$ only if a successful call will make its own ECSC() or Write() operation visible. In all other cases α uses $h.Casual$.

The algorithm maintains the DurECW object $\mathcal{O}$'s state in $\mathcal{Z}$, i.e., $\mathcal{O}.seq = \mathcal{Z}.seq$ and $\mathcal{O}.val = \mathcal{Z}.val$ at all times. This representation makes the implementation of $\mathcal{O}$.ECLL() and $\mathcal{O}$.ECVL() operations obvious: $\mathcal{O}$.ECLL(h) simply returns $(\mathcal{Z}.seq, \mathcal{Z}.val)$ and ECVL(h, s) returns whether $\mathcal{Z}.seq = s$. The complexity lies in the implementation of $\mathcal{O}$.Write(h, v) and $\mathcal{O}$.ECSC(h, s, v) operations, which coordinate their actions using $\mathcal{W}.bit$ and $\mathcal{Z}.bit$. A write operation flips the $\mathcal{W}.bit$ to announce to the ECSC operations that their help is needed to push the write into $\mathcal{Z}$; once the write is helped, the $\mathcal{Z}.bit$ is flipped to announce that help is no longer needed. We maintain the invariant that $\mathcal{W}.bit \neq \mathcal{Z}.bit$ if and only if a write needs help.

Algorithm 1 : The DurEC class for Durable, External Context nW-LLSC objects.

```
class DurEC:
    instance variable (handle*, int) $X$        ▷ $X$ is a pair $(X.hndl, X.seq)$ stored in NVM
    instance variable (int, int) $Y$            ▷ $Y$ is a pair $(Y.seq, Y.val)$ stored in NVM

    struct handle {
        int $DetVal$
        int $Val$
    }

    static procedure CreateHandle()
1:      return new $handle\{DetVal = 0\}$       ▷ $DetVal$ and $Val$ in NVM; $Val$ arbitrarily initialized

    constructor DurEC(int $initval$)
2:      $X \leftarrow (null, 0)$
3:      $Y \leftarrow (0, initval)$

    procedure ECLL(handle* $h$)
4:      return $Y$

    procedure ECVL(handle* $h$, int $s$)
5:      return $Y.seq = s$

    procedure ECSC(handle* $h$, int $s$, int $v$)
6:      if $Y.seq \neq s$ then return $false$
7:      $h.Val \leftarrow v$
8:      $\hat{h} \leftarrow X.hndl$
9:      $\hat{s} \leftarrow \max(h.DetVal, s) + 1$
10:     $r \leftarrow$ Cas$(X, (\hat{h}, s), (h, \hat{s}))$
11:     forward$(h)$
12:     return $r$

    procedure forward(handle* $h$)
13:     $x \leftarrow X$
14:     $\hat{s} \leftarrow x.hndl.DetVal$
15:     if $\hat{s} < x.seq$ then Cas$(x.hndl.DetVal, \hat{s}, x.seq)$
16:     $\hat{v} \leftarrow x.hndl.Val$
17:     $y \leftarrow Y$
18:     if $y.seq < x.seq$ then Cas$(Y, y, (x.seq, \hat{v}))$
19:     return

    procedure Recover(handle* $h$)
20:     forward$(h)$
21:     return

    static procedure Detect(handle* $h$)
22:     return $(h.DetVal, true)$
```

A Write(h, v) operation α consists of the following steps.

(W1) The operation α reads $\mathcal{W}$ and $\mathcal{Z}$ to determine if some write operation is already waiting for help. If not, then α installs its write into $\mathcal{W}$ by setting $\mathcal{W}.val$ to v and flipping $\mathcal{W}.bit$. If several write operations attempt to install concurrently, only one will succeed. The successful one is the *installer* and the others are *hitchhikers.*

(W2) Once a write operation is installed, all processes – installer, hitchhiker, and the ECSC operations – work in concert to forward the installer's operation to $\mathcal{Z}$. Since $\mathcal{Z}$ is where the DurECW object's state is held, the installer's operation takes effect only when it is reflected in $\mathcal{Z}$'s state. Towards this end, everyone attempts to transfer the installer's value from $\mathcal{W}$ to $\mathcal{Z}$. However, a stale ECSC operation, which was poised to execute its ECSC operation on $\mathcal{Z}$, might update $\mathcal{Z}$, causing the transfer to fail in moving the installer's value from $\mathcal{W}$ to $\mathcal{Z}$. So, a transfer is attempted the second time. The earlier success by the poised ECSC operation causes any future attempts by

Algorithm 2 The DurECW class for Durable External Context W-LLSC objects.

```
class DurECW:
    instance variable DurEC $\mathcal{W}$          ▷ $\mathcal{W}$ holds a pair $(\mathcal{W}.seq, (\mathcal{W}.val, \mathcal{W}.bit))$
    instance variable DurEC $\mathcal{Z}$          ▷ $\mathcal{Z}$ holds a pair $(\mathcal{Z}.seq, (\mathcal{Z}.val, \mathcal{Z}.bit))$

    struct handle {
        DurEC.handle Critical
        DurEC.handle Casual
    }

    static procedure CreateHandle()
1:      return new handle{Critical ← DurEC.CreateHandle(), Casual ← DurEC.CreateHandle()}

    procedure DurECW(initval)
2:      $\mathcal{W} \leftarrow$ DurEC((0, 0))
3:      $\mathcal{Z} \leftarrow$ DurEC((initval, 0))

    procedure ECLL(handle* h)
4:      $z \leftarrow \mathcal{Z}$.ECLL(h.Casual)
5:      return (z.seq, z.val)

    procedure ECVL(handle* h, int s)
6:      return $\mathcal{Z}$.ECVL(h.Casual, s)

    procedure ECSC(handle* h, int s, int v)
7:      $z \leftarrow \mathcal{Z}$.ECLL(h.Casual)
8:      if $s \neq z.seq$ return false
9:      transfer-write(h)
10:     $r \leftarrow \mathcal{Z}$.ECSC(h.Critical, s, (v, z.bit))
11:     return r

    procedure Write(handle* h, int v)
12:     $w \leftarrow \mathcal{W}$.ECLL(h.Casual)
13:     $z \leftarrow \mathcal{Z}$.ECLL(h.Casual)
14:     if $z.bit = w.bit$ then $\mathcal{W}$.ECSC(h.Critical, w.seq, (v, 1 − w.bit))
15:     transfer-write(h)
16:     transfer-write(h)
17:     return true

    procedure transfer-write(handle* h)
18:     $\hat{z} \leftarrow \mathcal{Z}$.ECLL(h.Casual)
19:     $\hat{w} \leftarrow \mathcal{W}$.ECLL(h.Casual)
20:     if $\hat{z}.bit \neq \hat{w}.bit$ then $\mathcal{Z}$.ECSC(h.Casual, $\hat{z}$.seq, ($\hat{w}$.val, $\hat{w}$.bit))

    procedure Recover(handle* h)
21:     $\mathcal{W}$.Recover(h.Critical)
22:     $\mathcal{Z}$.Recover(h.Critical)
23:     $\mathcal{W}$.Recover(h.Casual)
24:     $\mathcal{Z}$.Recover(h.Casual)
25:     transfer-write(h)
26:     transfer-write(h)

    static procedure Detect(handle* h)
27:     return DurEC.Detect(h.Critical)
```

similarly poised operations to fail. Consequently, the installer's write value gets moved to $\mathcal{Z}$ by the time the second transfer attempt completes. The point where the move to $\mathcal{Z}$ occurs becomes the linearization point for the installer's write operation. We linearize the writes by the hitchhikers immediately before the installer, which makes their write operations to be overwritten immediately by the installer's write, without anyone ever witnessing their writes. Hence, there is no need to detect these writes: if a hitchhiker crashes during its write, the operation can be safely repeated.

(W3) If the installer crashes after installing, upon restart, in the Recover method, it does the forwarding so that its install moves to $\mathcal{Z}$ and its write operation gets linearized.

An ECSC(h, s, v) operation α consists of the following steps.

(S1) α performs an ECLL() to determine whether the context in $\mathcal{O}$ matches s. If not, it can fail early and return *false*.

(S2) If a WRITE() is already in $\mathcal{W}$ and waiting for help to be transferred to $\mathcal{Z}$, α is obligated to help that write before attempting its SC (to prevent the write from being blocked by a chain of successful ECSC() operations). So it attempts a transfer from $\mathcal{W}$ to $\mathcal{Z}$.

(S3) Finally α executes an ECSC() on $\mathcal{Z}$ in an attempt to make its own operation $\mathcal{O}$ take effect.

The algorithm is formally presented in Algorithm 2. In the algorithm, Lines 12-14 implement step W1 and Lines 15, 16 implement step W2. Step S1 is implemented by Lines 7, 8, step S2 by 9 and S3 by 10 and 11. Note that the ECSC() on line 10 takes care to not change $\mathcal{Z}.bit$. This ensures that the helping mechanism for writes implemented via $\mathcal{W}.bit$ and $\mathcal{Z}.bit$ is not disturbed. The ECSC() operation at Line 14 uses the handle $h.Critical$ because its success implies that the operation is an installer and hence will be a visible write when it linearizes. Similarly the ECSC() on $\mathcal{Z}$ at Line 10 uses $h.Critical$ because its success makes the ECSC() on $\mathcal{O}$ visible.

If a WRITE() or a ECSC() method crashes while executing an operation on $\mathcal{W}$ or $\mathcal{Z}$, upon restart, Lines 21 to 24 of RECOVER() ensure that $\mathcal{W}$.RECOVER() or $\mathcal{Z}$.RECOVER() is executed before any other operation is executed on $\mathcal{W}$ or $\mathcal{Z}$ (the relative order of lines 21-24 is unimportant). Consequently, the durable objects $\mathcal{W}$ and $\mathcal{Z}$ behave like atomic EC objects.

The theorem below summarizes the result:

▶ **Theorem 2.** *Algorithm DurECW satisfies the following properties:*

1. *The objects implemented by the algorithm are durable, i.e., they satisfy method-based recoverable linearizability and method-based detectability (with respect to the sequential specification of External Context Writable-LLSC).*
2. *All object operations including Recover are wait-free and run in constant time.*
3. *The algorithm supports dynamic joining: a new process can join in at any point in a run (by calling CreateHandle) and start creating DurECW objects or accessing existing DurECW objects.*
4. *The space requirement is $O(m + n)$, where m is the actual number of DurECW objects created in the run, and n is the actual number of processes that have joined in in a run.*

6.2 The DuraLL Algorithm

Given the durable EC W-LLSC object DurECW, rolling the context into the implementation to produce a durable standard W-LLSC object is simple. Each of our implemented DuraLL objects simply maintains a single DurECW object X. The handle of the DuraLL object simply maintains a single DurECW handle to operate on X, and a hashmap, *cntxts*, that maps objects to contexts.

We present the DuraLL code as Algorithm 4 in the Appendix A. The LL() operation on a DuraLL object by handle h simply performs an ECLL() on X and stores the returned context in $h.cntxts$ under the key *self* (which is the reference of the current object). Correspondingly, VL() retrieves the context from $h.cntxts$, and uses it to perform an ECVL() on X. The SC() operation also retrieves the context and performs an ECSC() on the internal object, but then cleverly removes the key corresponding to the current object from $h.cntxts$, since, regardless of whether the SC() succeeds, the stored context is bound to be out-of-date. The WRITE() operation does not need a context, so it simply writes to X, but also cleverly removes the current object's key from $h.cntxts$ to save some space. In order to be space-efficient, RECOVER() also removes the current object from *h.cntxts* if the context stored for the object is out-of-date. Since DURALL is just a wrapper around DURECW, its DETECT() operation simply returns the result of detecting DURECW.

▶ **Theorem 3.** *Algorithm* DURALL *satisfies the following properties:*

1. *The objects implemented by the algorithm are durable, i.e., they satisfy method-based recoverable linearizability and method-based detectability (with respect to the sequential specification of Writable LLSC).*
2. *All object operations including Recover are wait-free and run in constant time.*
3. *The algorithm supports dynamic joining: a new process can join in at any point in a run (by calling CreateHandle) and start creating* DURALL *objects or accessing existing* DURALL *objects.*
4. *The space requirement is $O(m+n+C)$, where m is the actual number of* DURALL *objects created in the run, n is the actual number of processes that have joined in in a run, and C is the number of "contexts" stored across all objects.*

7 DuraCAS: a durable implementation of Writable CAS

We present in Figure 3 Algorithm DURACAS, which implements a durable writable CAS object $\mathcal{O}$ from two DurEC objects, $\mathcal{W}$ and $\mathcal{Z}$. The algorithm bears a lot of similarity to Algorithm DURECW of the previous section. In fact, DURACAS has only three extra lines. For readability, we starred their line numbers (Lines **6***, **10***, and **13***) and kept the line numbers the same for the common lines.

The ideas underlying this algorithm are similar to DURECW, so we explain here only the three differences: (1) Lines 7 to 10 are executed only once in Algorithm DURECW, but are repeated twice in the current algorithm; (2) Line 8 differs in the two algorithms; and (3) Line 13* is introduced in the current algorithm.

The change in Line 8 accounts for the fact that the success of a CAS() operation depends on the value in $\mathcal{O}$ rather than the context. If the value in $\mathcal{O}$ (and therefore $\mathcal{Z}$) is different from *old* at Line 7, the CAS returns *false* (and linearizes at Line 7). If $\mathcal{O}.val = old$ and the CAS does not plan to change the value (i.e., $old = new$) it returns *true* without changing $\mathcal{Z}$.

To understand why Lines 7 to 10 are repeated in the current algorithm, consider the following scenario. A handle h executes $\mathcal{O}.CAS(h, old, new)$, where $old \neq new$. When h executes Line 7, $\mathcal{Z}$'s value is *old*, so *z.val* gets set to *old* at Line 7. Handle h progresses to Line 10, but before it executes Line 10, some handle h' invokes $\mathcal{O}.\text{WRITE}(h', old)$ and executes it to completion, causing $\mathcal{Z}.seq$ to take on a value greater than *z.seq*. Handle h now executes the ECSC at Line 10 and fails since $\mathcal{Z}.seq \neq z.seq$. If h acts as it did in Algorithm DURECW, h would complete its $\mathcal{O}.CAS(h, old, new)$ operation, returning *false*. However, *false* is an incorrect response by the specification of CAS because $\mathcal{O}.val = old$ for the full duration of the operation $\mathcal{O}.CAS(h, old, new)$. To overcome this race condition, h repeats Lines 7 to 10.

If the same race condition repeats each time h repeats Lines 7 to 10, the method $\mathcal{O}.CAS$ would not be wait-free. Line 13* is introduced precisely to prevent this adverse possibility. When a handle h' executes Lines 12 to 14 of $\mathcal{O}.\text{WRITE}(h', v)$ in the previous DURECW algorithm, h' would always try to install its value v in $\mathcal{W}$ (at Line 14) and later move it to $\mathcal{Z}$, thereby increasing $\mathcal{Z}.seq$ and causing concurrent $\mathcal{O}$.ECSC() operations to fail. This was precisely what we wanted because the specification of an SC operation requires that if *any* $\mathcal{O}$.WRITE() takes effect, regardless of what value it writes in $\mathcal{O}$, it must change $\mathcal{O}.context$ and thus cause concurrent $\mathcal{O}$.ECSC() operations to fail. The situation however, is different

Algorithm 3 The DuraCAS class for Durable, Writable-CAS objects.

```
     class DuraCAS:
         instance variable  DurEC  $\mathcal{W}$        ▷ $\mathcal{W}$ holds a pair $(\mathcal{W}.seq, (\mathcal{W}.val, \mathcal{W}.bit))$
         instance variable  DurEC  $\mathcal{Z}$        ▷ $\mathcal{Z}$ holds a pair $(\mathcal{Z}.seq, (\mathcal{Z}.val, \mathcal{Z}.bit))$

         struct handle {
             DurEC.handle* Critical
             DurEC.handle* Casual
         }

         static procedure CreateHandle()
1:           return new handle{Critical ← DurEC.CreateHandle(), Casual ← DurEC.CreateHandle()}

         procedure DuraCAS(int initval)
2:           $\mathcal{W} \leftarrow$ DurEC((0, 0))
3:           $\mathcal{Z} \leftarrow$ DurEC((initval, 0))

         procedure Read(handle* h)
4:           $z \leftarrow \mathcal{Z}$.ECLL(h.Casual)
5:           return $z.val$

6:

         procedure CAS(handle* h, int old, int new)
6*:           for $i \leftarrow 1$ to 2
7:               $z \leftarrow \mathcal{Z}$.ECLL(h.Casual)
8:               if $z.val \neq old$ then return false else if $old = new$ then return true
9:               transfer-write(h)
10:              if $\mathcal{Z}$.ECSC(h.Critical, $z.seq$, $(new, z.bit)$) then
10*:                 return true
11:          return false

         procedure Write(handle* h, int v)
12:          $w \leftarrow \mathcal{W}$.ECLL(h.Casual)
13:          $z \leftarrow \mathcal{Z}$.ECLL(h.Casual)
13*:          if $z.val = v$ then return ack
14:          if $z.bit = w.bit$ then $\mathcal{W}$.ECSC(h.Critical, $w.seq$, $(v, 1 - w.bit)$)
15:          transfer-write(h)
16:          transfer-write(h)
17:          return ack

         procedure transfer-write(handle* h)
18:          $\hat{z} \leftarrow \mathcal{Z}$.ECLL(h.Casual)
19:          $\hat{w} \leftarrow \mathcal{W}$.ECLL(h.Casual)
20:          if $\hat{z}.bit \neq \hat{w}.bit$ then $\mathcal{Z}$.ECSC(h.Casual, $\hat{z}.seq$, $(\hat{w}.val, \hat{w}.bit)$)

         procedure Recover(handle* h)
21:          $\mathcal{W}$.Recover(h.Critical)
22:          $\mathcal{Z}$.Recover(h.Critical)
23:          $\mathcal{W}$.Recover(h.Casual)
24:          $\mathcal{Z}$.Recover(h.Casual)
25:          transfer-write(h)
26:          transfer-write(h)

         static procedure Detect(handle* h)
27:          return DurEC.Detect(h.Critical)
```

when implementing $\mathcal{O}.CAS$, where a $\mathcal{O}$.Write() that does not change the value in $\mathcal{O}$ should not cause a concurrent $\mathcal{O}.CAS$ to fail. Hence, if a $\mathcal{O}$.Write(h', v) operation is writing the same value as $\mathcal{O}$'s current value, then it should simply return (since $\mathcal{O}.val$ already has v) and, importantly, not change $\mathcal{Z}.seq$ (because changing $\mathcal{Z}.seq$ would cause any concurrent CAS operation to fail). Line 13* implements precisely this insight by ensuring that two Write$(-, v)$ operations both change $\mathcal{Z}$ only if there is some Cas$(-, v, v')$ or Write$(-, v')$ operation that changes $\mathcal{Z}$ in between (for some $v' \neq v$).

The theorem below summarizes the result:

▶ **Theorem 4.** *Algorithm DuraCAS satisfies the following properties:*

1. *The objects implemented by the algorithm are durable, i.e., they satisfy method-based recoverable linearizability and method-based detectability (with respect to the sequential specification of Writable CAS).*
2. *All object operations including Recover are wait-free and run in constant time.*
3. *The algorithm supports dynamic joining: a new process can join in at any point in a run (by calling CreateHandle) and start creating DuraCAS objects or accessing existing DuraCAS objects.*
4. *The space requirement is $O(m + n)$, where m is the actual number of DuraCAS objects created in the run, and n is the actual number of processes that have joined in in a run.*

8 Discussion and Remarks

In this paper, we have designed constant time implementations for durable CAS and LLSC objects. To our knowledge, DuraCAS is the first CAS implementation to allow for dynamic joining. DuraCAS also has state-of-the-art space complexity – allowing adaptivity and requiring only constant space per object and per process that actually accesses the protocol – and is writable. To our knowledge, ours are the first implementations of durable LLSC objects. LLSC objects are universal and ABA-free, thus we believe that the dynamically joinable LLSC implementations in this paper will be useful in the construction of several more complex durable objects. The external context variant of LLSC is particularly space efficient, making it a powerful building block for concurrent algorithms; we witnessed this property even in the constructions of this paper, where the EC nW-LLSC object DurEC served as the primary building block for all our other implementations, including our EC W-LLSC implementation DurECW and its direct descendent DuraLL (for W-LLSC). All the implementations in this paper were enabled by handles – a pointer-based mechanism we introduced to enable threads created on-the-fly to access our implementations. We believe that along with the specific implementations of this paper, the use of handles as an algorithmic tool can play an important role in the design of future durable algorithms.

We end with two open problems. Handles enable dynamic joining, but once a handle h is used, any other process can have a stale pointer to h that may be dereferenced at any point in the future. A mechanism for enabling space adaptivity for both dynamic joining and *dynamic leaving*, which would enable a process to reclaim its entire memory footprint once it is done using a durable implementation is our first open problem. Our second open problem is to prove (or disprove) an $\Omega(m + n)$ space lower bound for supporting m objects for n processes for any durable CAS or durable LLSC type.

References

1 Zahra Aghazadeh, Wojciech Golab, and Philipp Woelfel. Making objects writable. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 385–395, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2611462.2611483.

2 Marcos K. Aguilera and Svend Frølund. Strict linearizability and the power of aborting. In *techreport*, 2003.

3 James H. Anderson and Mark Moir. Universal constructions for multi-object operations. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 184–193, New York, NY, USA, 1995. Association for Computing Machinery. doi:10.1145/224964.224985.

4 Hagit Attiya, Ohad Ben-Baruch, Panagiota Fatourou, Danny Hendler, and Eleftherios Kosmas. Detectable recovery of lock-free data structures. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, pages 262–277, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3503221.3508444.

5 Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. Nesting-safe recoverable linearizability: Modular constructions for non-volatile memory. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, PODC '18, pages 7–16, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3212734.3212753.

6 Ohad Ben-Baruch, Danny Hendler, and Matan Rusanovsky. Upper and lower bounds on the space complexity of detectable objects. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, PODC '20, pages 11–20, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3382734.3405725.

7 Naama Ben-David, Guy E. Blelloch, Michal Friedman, and Yuanhao Wei. Delay-free concurrency on faulty persistent memory. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '19, pages 253–264, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3323165.3323187.

8 Naama Ben-David, Michal Friedman, and Yuanhao Wei. Brief announcement: Survey of persistent memory correctness conditions. In Christian Scheideler, editor, *36th International Symposium on Distributed Computing, DISC 2022, October 25-27, 2022, Augusta, Georgia, USA*, volume 246 of *LIPIcs*, pages 41:1–41:4. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.DISC.2022.41.

9 Ryan Berryhill, Wojciech M. Golab, and Mahesh Tripunitara. Robust shared objects for non-volatile main memory. In Emmanuelle Anceaume, Christian Cachin, and Maria Gradinariu Potop-Butucaru, editors, *19th International Conference on Principles of Distributed Systems, OPODIS 2015, December 14-17, 2015, Rennes, France*, volume 46 of *LIPIcs*, pages 20:1–20:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015. doi:10.4230/LIPIcs.OPODIS.2015.20.

10 Guy E. Blelloch, Phillip B. Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. The parallel persistent memory model. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, pages 247–258, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3210377.3210381.

11 Guy E. Blelloch and Yuanhao Wei. LL/SC and atomic copy: Constant time, space efficient implementations using only pointer-width CAS. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, volume 179 of *LIPIcs*, pages 5:1–5:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.DISC.2020.5.

12 Philip Bohannon, Daniel Lieuwen, Avi Silberschatz, S. Sudarshan, and Jacques Gava. Recoverable User-Level Mutual Exclusion. In *In Proc. 7th IEEE Symposium on Parallel and Distributed Processing*, 1995.

13 Wentao Cai, Haosen Wen, H. Alan Beadle, Chris Kjellqvist, Mohammad Hedayati, and Michael L. Scott. Understanding and optimizing persistent memory allocation. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2020, pages 60–73, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3381898.3397212.

14 David Yu Cheng Chan and Philipp Woelfel. Recoverable mutual exclusion with constant amortized RMR complexity from standard primitives. In *ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, 2020. doi:10.1145/3382734.3405736.

15 David Yu Cheng Chan and Philipp Woelfel. Tight lower bound for the RMR complexity of recoverable mutual exclusion. In *ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, 2021. doi:10.1145/3465084.3467938.

16 D. Dechev, P. Pirkelbauer, and B. Stroustrup. Understanding and effectively preventing the aba problem in descriptor-based lock-free designs. In *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 185–192, 2010. doi:10.1109/ISORC.2010.10.

17 Sahil Dhoked and Neeraj Mittal. An adaptive approach to recoverable mutual exclusion. In *PODC '20: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, August 3-7, 2020*. ACM, 2020. doi:10.1145/3382734.3405739.

18 Simon Doherty, Maurice Herlihy, Victor Luchangco, and Mark Moir. Bringing practical lock-free synchronization to 64-bit applications. In Soma Chaudhuri and Shay Kutten, editors, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, PODC 2004, St. John's, Newfoundland, Canada, July 25-28, 2004*, pages 31–39. ACM, 2004. doi:10.1145/1011767.1011773.

19 Michal Friedman, Maurice Herlihy, Virendra J. Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. In Andreas Krall and Thomas R. Gross, editors, *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018*, pages 28–40. ACM, 2018. doi:10.1145/3178487.3178490.

20 Wojciech M. Golab and Danny Hendler. Recoverable mutual exclusion in sub-logarithmic time. In Elad Michael Schiller and Alexander A. Schwarzmann, editors, *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, pages 211–220. ACM, 2017. doi:10.1145/3087801.3087819.

21 Wojciech M. Golab and Aditya Ramaraju. Recoverable mutual exclusion: [extended abstract]. In George Giakkoupis, editor, *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, pages 65–74. ACM, 2016. doi:10.1145/2933057.2933087.

22 Rachid Guerraoui, Alex Kogan, Virendra J. Marathe, and Igor Zablotchi. Efficient multi-word compare and swap. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, volume 179 of *LIPIcs*, pages 4:1–4:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.DISC.2020.4.

23 Rachid Guerraoui and Ron R. Levy. Robust emulations of shared memory in a crash-recovery model. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, ICDCS '04, pages 400–407, USA, 2004. IEEE Computer Society.

24 happyware.com. Supermicro 8-socket intel xeon 7u rack server. https://happyware.com/uk-en/supermicro/sys-7089p-tr4t. Accessed: August 1, 2022.

25 Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991. doi:10.1145/114005.102808.

26 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990. doi:10.1145/78969.78972.

27 Intel. Intel optane technology, 2020. URL: https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html.

28 Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In Cyril Gavoille and David Ilcinkas, editors, *Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings*, volume 9888 of *Lecture Notes in Computer Science*, pages 313–327. Springer, 2016. doi:10.1007/978-3-662-53426-7_23.

29 Prasad Jayanti. A complete and constant time wait-free implementation of cas from ll/sc and vice versa. In *Proceedings of the 12th International Symposium on Distributed Computing*, DISC '98, pages 216–230, Berlin, Heidelberg, 1998. Springer-Verlag.

30 Prasad Jayanti, Siddhartha Jayanti, and Sucharita Jayanti. Durable algorithms for writable ll/sc and cas with dynamic joining, 2023. arXiv:2302.00135.

31 Prasad Jayanti, Siddhartha Jayanti, and Anup Joshi. A recoverable mutex algorithm with sub-logarithmic rmr on both cc and dsm. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, pages 177–186, New York, NY, USA, 2019. Association for Computing Machinery. `doi:10.1145/3293611.3331634`.

32 Prasad Jayanti, Siddhartha Jayanti, and Anup Joshi. Constant rmr system-wide failure resilient durable locks with dynamic joining. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '23, pages 227–237, New York, NY, USA, 2023. Association for Computing Machinery. `doi:10.1145/3558481.3591100`.

33 Prasad Jayanti, Siddhartha V. Jayanti, and Anup Joshi. Optimal recoverable mutual exclusion using only FASAS. In Andreas Podelski and François Taïani, editors, *Networked Systems - 6th International Conference, NETYS 2018, Essaouira, Morocco, May 9-11, 2018, Revised Selected Papers*, volume 11028 of *Lecture Notes in Computer Science*, pages 191–206. Springer, 2018. `doi:10.1007/978-3-030-05529-5_13`.

34 Prasad Jayanti, Siddhartha Visveswara Jayanti, and Sucharita Jayanti. Brief announcement: Efficient recoverable writable-cas. In *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing*, PODC '23, pages 366–369, New York, NY, USA, 2023. Association for Computing Machinery. `doi:10.1145/3583668.3594592`.

35 Prasad Jayanti and Anup Joshi. Recoverable FCFS mutual exclusion with wait-free recovery. In Andréa W. Richa, editor, *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, volume 91 of *LIPIcs*, pages 30:1–30:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. `doi:10.4230/LIPIcs.DISC.2017.30`.

36 Prasad Jayanti and Anup Joshi. Recoverable mutual exclusion with abortability. In Mohamed Faouzi Atig and Alexander A. Schwarzmann, editors, *Networked Systems - 7th International Conference, NETYS 2019, Marrakech, Morocco, June 19-21, 2019, Revised Selected Papers*, volume 11704 of *Lecture Notes in Computer Science*, pages 217–232. Springer, 2019. `doi:10.1007/978-3-030-31277-0_14`.

37 Prasad Jayanti and Srdjan Petrovic. Efficient and practical constructions of ll/sc variables. In *Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing*, PODC '03, pages 285–294, New York, NY, USA, 2003. Association for Computing Machinery. `doi:10.1145/872035.872078`.

38 Prasad Jayanti and Srdjan Petrovic. Efficiently implementing LL/SC objects shared by an unknown number of processes. In Ajit Pal, Ajay D. Kshemkalyani, Rajeev Kumar, and Arobinda Gupta, editors, *Distributed Computing - IWDC 2005, 7th International Workshop, Kharagpur, India, December 27-30, 2005, Proceedings*, volume 3741 of *Lecture Notes in Computer Science*, pages 45–56. Springer, 2005. `doi:10.1007/11603771_5`.

39 Nan Li and Wojciech M. Golab. Detectable sequential specifications for recoverable shared objects. In Seth Gilbert, editor, *35th International Symposium on Distributed Computing, DISC 2021, October 4-8, 2021, Freiburg, Germany (Virtual Conference)*, volume 209 of *LIPIcs*, pages 29:1–29:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPIcs.DISC.2021.29`.

40 Maged M. Michael. Practical lock-free and wait-free LL/SC/VL implementations using 64-bit CAS. In Rachid Guerraoui, editor, *Distributed Computing, 18th International Conference, DISC 2004, Amsterdam, The Netherlands, October 4-7, 2004, Proceedings*, volume 3274 of *Lecture Notes in Computer Science*, pages 144–158. Springer, 2004. `doi:10.1007/978-3-540-30186-8_11`.

41 Aditya Ramaraju. RGLock: Recoverable mutual exclusion for non-volatile main memory systems. Master's thesis, University of Waterloo, 2015. URL: `https://uwspace.uwaterloo.ca/handle/10012/9473`.

42 Tianzheng Wang, Justin J. Levandoski, and Per-Åke Larson. Easy lock-free indexing in non-volatile memory. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pages 461–472. IEEE Computer Society, 2018. `doi:10.1109/ICDE.2018.00049`.

A DuraLL Implementation

Algorithm 4 The DuraLL class for Durable Writable-LLSC objects.

```
class DuraLL:
    instance variable DurECW $X$                    ▷ $X$ holds the central EC W-LLSC object.

    struct handle {
        DurECW.handle $ECWH$
        HashMap (DuraLL → int) $cntxts$
    }

    static procedure CreateHandle()
1:      return new $handle\{ECWH \leftarrow$ DurECW.CreateHandle()$, cntxts \leftarrow$ HashMap(DuraLL → int)$\}$

    procedure DuraLL($initval$)
2:      $X \leftarrow$ DurECW($initval, 0$)

    procedure LL(handle* $h$)
3:      $x \leftarrow X$.ECLL($h.ECWH$)
4:      $h.cntxts(self) \leftarrow x.seq$
5:      return $x.val$

    procedure VL(handle* $h$)
6:      if $self \notin h.cntxts.keys$ then return $false$
7:      return $X$.ECVL($h.ECWH, h.cntxts(self)$)

    procedure SC(handle* $h$, int $val$)
8:      if $self \notin h.cntxts.keys$ then return $false$
9:      $r \leftarrow X$.ECSC($h.ECWH, h.cntxts(self), val$)
10:     $h.cntxts$.Remove($self$)
11:     return $r$

    procedure Write(handle* $h$, int $val$)
12:     $X$.Write($h.ECWH, val$)
13:     $h.cntxts$.Remove($self$)
14:     return $true$

    procedure Recover(handle* $h$)
15:     $X$.Recover($h.ECWH$)
16:     if $self \in h.cntxts.keys$ then
            if $\neg X$.ECVL($h.ECWH, h.cntxts(self)$) then $h.context$.Remove($self$)

    static procedure Detect(handle* $h$)
17:     return DurECW.Detect($h.ECWH$)
```

Cordial Miners: Fast and Efficient Consensus for Every Eventuality

Idit Keidar
Technion, Haifa, Israel

Oded Naor
Technion, Haifa, Israel
StarkWare Industries Ltd., Netanya, Israel

Ouri Poupko
Ben-Gurion University, Beer Sheva, Israel

Ehud Shapiro
Weizmann Institute of Science, Rehovot, Israel

Abstract

Cordial Miners are a family of efficient Byzantine Atomic Broadcast protocols, with instances for asynchrony and eventual synchrony. They improve the latency of state-of-the-art DAG-based protocols by almost 2× and achieve optimal good-case complexity of $O(n)$ by forgoing Reliable Broadcast as a building block. Rather, Cordial Miners use the *blocklace* – a partially-ordered counterpart of the totally-ordered blockchain data structure – to implement the three algorithmic components of consensus: Dissemination, equivocation-exclusion, and ordering.

2012 ACM Subject Classification Computing methodologies → Distributed algorithms

Keywords and phrases Byzantine Fault Tolerance, State Machine Replication, DAG, Consensus, Blockchain, Blocklace, Cordial Dissemination

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.26

Related Version *Full Version*: `https://arxiv.org/abs/2205.09174`

Funding *Oded Naor*: Oded Naor is grateful to the Azrieli Foundation for the award of an Azrieli Fellowship, and to the Technion Hiroshi Fujiwara Cyber-Security Research Center for providing a research grant.

Acknowledgements Ehud Shapiro is the Incumbent of The Harry Weinrebe Professorial Chair of Computer Science and Biology at the Weizmann Institute.

1 Introduction

The problem of ordering transactions in a permissioned Byzantine distributed system, also known as *Byzantine Atomic Broadcast (BAB)*, has been investigated for four decades [30], and in the last decade, has attracted renewed attention due to the emergence of cryptocurrencies.

Recently, a line of works [4, 14, 20, 33, 21, 27] suggests ordering transactions using a distributed Directed Acyclic Graph (DAG) structure, in which each vertex contains a block of transactions as well as references to previously sent vertices. The DAG is distributively constructed from messages of *miners* running the consensus protocol. While building the DAG structure, each miner also totally orders the vertices in its DAG locally. That is, as the DAG is being constructed, a consensus on its ordering emerges without additional communication among the miners.

Table 1 Performance summary. Bullshark is for the ES model, and DAG-Rider is for the asynchronous model. Both protocols employ RB, which requires at least two rounds of communication of simple messages for optimal latency [2] and $O(n^2)$ amortized message complexity, or four rounds with erasure coding when using Das et al. [15] and $O(n)$ amortized message complexity.

Protocol	**Reliable Broadcast Used**	**Latency**				**Amortized Message Complexity**
		Eventual Synch.		**Async.**		
		Good	Expected	Good	Expected	
Cordial Miners (this work)	None	3	4.5	5	7.5	good-case: $O(n)$ worst-case: $O(n^2)$
Bullshark (for ES) **DAG-Rider** (for asynch.)	Optimal latency [2]	4	9	8	12	good- & worst-case: $O(n^2)$
	Das et al. [15]	8	18	16	24	good- & worst-case: $O(n)$

The two state-of-the-art protocols in this context are DAG-Rider [21] and Bullshark [33]. DAG-Rider works in the asynchronous setting, in which the adversary controls the finite delay on message delivery between miners, and Bullshark works in the Eventual Synchrony (ES) model, in which eventually all messages between correct miners are delivered within a known time-bound.

Both protocols use *Reliable Broadcast (RB)* [7] as a building block to disseminate vertices in the DAG. RB ensures that Byzantine miners cannot equivocate, i.e., they cannot successfully send two conflicting vertices to the correct miners. By using RB to exclude equivocation, the DAGs of all correct miners eventually contain the same vertices.

But using RB has costs in terms of message complexity and latency. The well-known Bracha RB [7] protocol entails $O(n^2)$ message complexity for each broadcast message, where n is the number of miners, and has a latency of 3 rounds of communication. The lower bound for RB is 2 rounds [2], and the message complexity lower bound is $O(n^2)$ [19]. Recent RB protocols [15, 16] improve the message complexity to $O(n)$ in some cases by using erasure codes [5], but require between 4 to 5 rounds of communication.

DAG-Rider and Bullshark need to invoke a sequence of RB instances several times to reach a single instance of consensus. E.g., DAG-Rider requires 6 sequential instances of RB in the expected case, making its latency between 12 to 24 rounds of communication, depending on the RB protocol it uses. Bullshark requires between 9 to 18 rounds in the expected case in the ES model.

It is within this context that we introduce *Cordial Miners* – a family of simple, efficient, self-contained Byzantine Atomic Broadcast [9] protocols that forgo RB, and present two of its instances for the models ES and asynchrony.

The ES Cordial Miners protocol reduces the expected latency from 9 rounds in today's state-of-the-art to 4.5, and the good case latency from 4 to 3. The asynchronous version of Cordial Miners improves the expected latency from 12 rounds to 7.5, and the good case latency from 8 to 5. This is while maintaining the same amortized quadratic message complexity in the worst case. Cordial Miners also demonstrates better performance with $O(n)$ complexity in the good case when the actual number of Byzantine miners is $O(1)$ and the network is synchronous. Protocols that use RB do not differ in their performance between the good and worst cases. Tab. 1 summarizes Cordial Miners' performance compared to DAG-Rider (for asynchrony) and Bullshark (for ES).

The crux of the Cordial Miners protocols is that instead of using RB to eliminate equivocation (and absorbing its rather high latency), miners cooperatively create a data structure that accommodates equivocations, termed *blocklace*, which is a partially-ordered counterpart of the blockchain data structure [29]. When a miner wishes to disseminate a block, it simply sends it to all other miners, taking a single round of communication, instead of at least two when using reliable broadcast.

Although the blocklace may contain equivocating blocks created by Byzantine miners, they are excluded by the ordering protocol, which is locally computed by each miner without inducing any extra communication or latency. This is realized by the function τ that converts the partially-ordered blocklace to a totally-ordered sequence of blocks while excluding equivocations along the way. Thus, by "complicating" the local ordering task to exclude equivocations, we forgo the extra communication rounds and latency associated with RB.

Roadmap. The rest of the paper is structured as follows: §2 describes the models and defines the problem; §3 provides intuition and overview of the different components; §4 introduces the blocklace data structure; §5 explains the τ function that locally turns the blocklace into a totally-ordered sequence of blocks; §6 describes the entire Cordial Miners protocols for the two network models; §7 presents the performance analysis; §8 is related work; and lastly, §9 concludes the paper. To accommodate the space limitations some details are deferred to the appendices. App. A describes a formal mathematical model for cordial miners, and some explanatory figures are deferred to App. B. Two additional appendices: one that details the full proofs and another that describes further future directions and optimizations appear in the full version of this paper [22].

2 Model and Problem Definition

We assume a set Π of $n \geq 3$ *miners* (aka agents, processes), of which at most $f < n/3$ may be *faulty* (act under the control of the adversary, be "Byzantine"), and the rest are *correct* (also honest or non-faulty). Each miner is equipped with a single and unique cryptographic key-pair, with the public key known to others. Miners can create, sign, and send messages to each other, where any message sent from one correct miner to another is eventually received. In addition, each miner can sequentially *output* (aka "deliver") messages (e.g., to a local output device or storage device). Thus, each miner outputs a *sequence* of messages.

Let Λ denote the empty sequence; for a set X, X^* is the set of all sequences over X; for sequences x and y, $x \preceq y$ denotes that x is a prefix of y; $x \cdot y$ denotes the concatenation of x and y; and x, y are *consistent* if $x \preceq y$ or $y \preceq x$.

The problem we aim to solve in this paper is to devise an ordering consensus protocol that is safe and live:

▶ **Definition 1** (Safety and Liveness of an Ordering Consensus Protocol). *An ordering consensus protocol is:*

Safe *if output sequences of correct miners are consistent.*

Live *if every message sent by a correct miner is eventually output by every correct miner with probability 1.*

Here, we aim to devise safe and live ordering consensus protocols for models of distributed computing with two types of adaptive adversaries that can corrupt up to f miners throughout the run: First, *Asynchrony*, in which the adversary controls the finite delay of every message. Second, *Eventual Synchrony (ES)*, in which there is a point in time, known as the *Global Stabilization Time (GST)*. After GST, the adversary controls the delivery time of messages sent between correct miners, but they must be delivered within a known bound Δ. We further assume the adversary is computationally bounded and, therefore, cannot break cryptographic signatures.

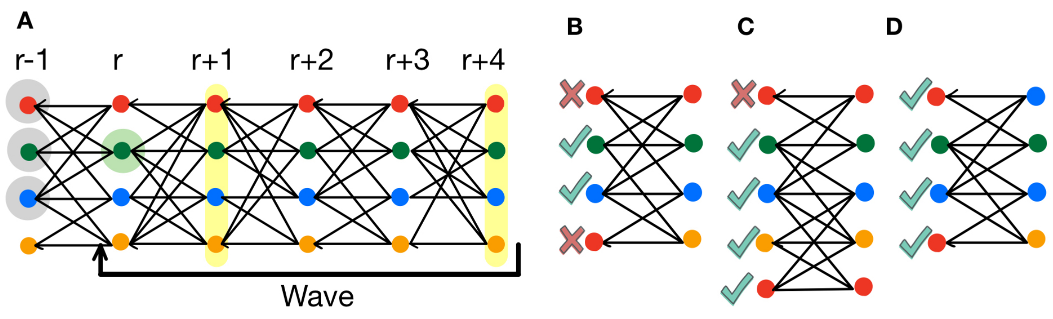


Figure 1 The blocklace data structure, equivocations, approval, and ratification. Four miners (red, green, blue, yellow). Each circle represents a block and each line a hash pointer to the left block. (A) A single wave consisting of five consecutive rounds. The green block in round r with the halo is the leader block. Each of the highlighted blocks in yellow in rounds $r+1$ and $r+4$ have a path to the leader block, making it a final leader block. The blocks with a gray halo are ordered by τ when the leader block becomes final. (B) The red equivocates, with the top red block approved by the green block of the next round, the bottom red block approved by the yellow block of the next round, and the blue block of the next round, observing both equivocating red blocks, approves neither, and hence neither of the red blocks has the three approvals (including the red block itself) needed for ratification. (C) Here the blue block of the next round observes only the bottom red block and hence approves it, which together with the yellow block and the red block itself form a supermajority, and hence the bottom red block is ratified, but not the top one. (D) Here the blue miner equivocates in the next round, with the top blue block of the next round approving the top red block, which together with the green and red form a supermajority that ratifies it. Similarly, the bottom blue block, the yellow block, and the red (which is not illustrated) ratify the bottom red block. Indeed, with two equivocators (red and blue) out of four, an equivocation can be ratified.

We note that safety and liveness, combined with message uniqueness (e.g., a block in a blocklace, see next), imply the standard Byzantine Atomic Broadcast guarantees: Agreement, Integrity, Validity, and Total Order [9, 21]. Hence, protocols that address the problem defined here are in fact protocols for Byzantine Atomic Broadcast.

Next, we provide an overview of the Cordial Miners protocol, including the blocklace, the dissemination of blocks, and the local ordering of the blocks to a final sequence.

3 Cordial Miners Overview

In the Cordial Miners protocols, the miners jointly built the *blocklace* data structure, a partially-ordered counterpart of the totally-ordered blockchain. A blocklace created by four miners, each of a different color, is is illustrated in Fig. 1. The *depth* of a block in a blocklace is the length of the maximal path emanating from it, and a *round* of a blocklace consists of blocks of the same depth. A set of blocks by more than $\frac{1}{2}(n+f)$ miners is termed a *supermajority*; note that if $f=0$ then a supermajority is a simple majority. Correct miners are *cordial* in that they wait for round r to attain a supermajority before contributing a block to round $r+1$.

Fig. 1.A presents a blocklace constructed by four miners s.t. each column is a single round representing blocks from different miners, and each row in the same color consists of blocks from the same miner. Thus, each correct miner creates a single block in each round. Note that different miners can have different partial views of the blocklace, and the goal is to “converge” the order of the blocks to a consistent order for all the miners.

Each block holds a set of transactions as well as hash pointers (the edges in the DAG) to blocks of previous rounds. When a miner observes that round r has attained a supermajority (is *cordial*) it creates a new block b of round $r+1$ with pointers to the *tips* of its blocklace up to round r, which are the blocks in the blocklace with no incoming edges from blocks of depth up to r. The tips must include a supermajority of blocks of round r, but possibly also blocks of earlier rounds not already observed by the blocks received in round r. (One block observes another if there is a path of pointers from one to the other.) E.g., if the figure represents the local blocklace of the red miner, then since round $r+4$ is cordial, the red miner can create a new block b in round $r+5$ with pointers to all the blocks in round $r+4$. The miner then sends b to all other miners. The blocklace data structure is defined in §4.

Next, we explain how Cordial Miners use the blocklace for the three algorithmic components of consensus: Dissemination, equivocation-exclusion, and ordering.

Dissemination. In the good case, dissemination is realized simply by each miner sending each new block to all other miners. However, faulty miners may fail to do so, possibly intentionally, and send new blocks only to some of the miners.

The principle of cordial dissemination [29] is: *Send to others blocks you know and think they need.* Its blocklace-based Byzantine-resilient implementation uses each block in the blocklace as an ack/nak message: A new block created by a correct miner p points, directly or indirectly, to the blocks in p's local blocklace. It thus discloses the blocks known by p at the time of its creation and, by omission, also of the blocks not yet known to p. This way, a miner q that receives p's block can send back to p any block known to q and not known to p according to the disclosure made by p's block. E.g., the green block in round $r+4$ serves as an ack message for all the blocks that it observes, including the red, green, and blue blocks in round $r+3$. It also serves as a nak message for the yellow block of round $r+3$. As an example of cordial dissemination, when the red miner sends the block it creates to the green miner in round $r+5$, it will also send to the green miner the yellow block in round $r+3$. The dissemination protocol is formally defined in §6.

Equivocation exclusion. Two blocks b_1, b_2 of the same miner are *equivocating* if neither observes the other, i.e., there is no path of pointers from b_1 to b_2 or from b_2 to b_1. Since Cordial Miners do not use RB to disseminate blocks, the blocklace created by Cordial Miners may include equivocations created by Byzantine miners, which are later excluded when each miner locally orders the blocks in its blocklace to a sequence of final blocks. The Cordial Miners protocol uses supermajority approval to exclude equivocations s.t. for each set of equivocating blocks, at most, one is included in the final output. In addition, after detecting an equivocation, correct miners ignore the Byzantine miner by not including direct pointers to their blocks. Thus, a Byzantine miner that equivocates is eventually detected, which results in it eventually being ignored by all correct miners. Equivocation exclusion is part of the τ ordering function which is detailed in §5.

Ordering. Ordering the partially-ordered blocklace can be achieved by topological sort of the DAG. The challenge is to ensure that all correct miners exclude equivocations and order the blocks identically so that they all produce the same total order. To this end, the blocklace is divided into *waves*, each consisting of several rounds, the number of which is different for ES and asynchrony (3 and 5 rounds per wave, respectively). E.g., Fig. 1.A depicts the asynchronous version which has 5 rounds in each wave.

For each wave, one of the miners is elected as the *leader*, and if the first round of the wave has a block produced by the leader, then it is the *leader block*. The figure depicts the green block in round r in the green halo as the leader block of that wave. When a wave ends, i.e., when the last round of the wave is cordial, the leader block becomes *final* if it has sufficient blocks that *approve* it, namely, it is not equivocating and there is a supermajority where each block observes a supermajority that observes the leader block. The figure shows two supermajorities, highlighted in yellow, where each block in the supermajority of round $r+4$ observes the supermajority of round $r+1$. The supermajority in round $r+1$ observes the leader block, and the leader block is not equivocating, making it final.

A final leader block b serves as the "anchor" of the ordering function τ, which topologically sorts (while excluding equivocations) all the blocks observed by b that have not been ordered yet. Thus, each time a wave ends with a final leader block, a portion of its preceding blocklace is ordered. In the figure, the blocks in round $r-1$ with a grey halo are ordered when the leader block in round r is final since it observes them. In case a wave ends with no final leader block, unordered blocks will be ordered when some subsequent wave ends with a final leader block. The full details of τ are in §5.

4 The Blocklace

A blocklace [28] is a partially-ordered counterpart of the totally-ordered blockchain data structure: In a blocklace, each block may contain a finite set of cryptographic hash pointers to previous blocks, in contrast to one pointer (or zero for the initial/genesis block) in a blockchain. Thus, a blocklace induces a DAG in which vertices represent its blocks and edges represent the pointers among its blocks. Next, we present the basic definitions of a blocklace, which appear as pseudocode in Alg. 1. A formal mathematical description of these definitions appears in App. A.

4.1 Blocklace Basics

In addition to the set of miners Π, we assume a given set of block **payloads** $\mathcal{A}$, typically sets of transactions, and a cryptographic hash function *hash*. A **block** consists of a payload $a \in \mathcal{A}$ and a set of hash pointers to previously created blocks, signed by its creator p, in which case it is also referred to as a $\boldsymbol{p}$**-block** (Def. 14). A block **acknowledges** another block if it contains a hash pointer to it, and is **initial** if the set of hash pointers is empty. A *blocklace* is a set of blocks (Def. 15). Note that *hash* being cryptographic implies that a blocklace that includes a cycle cannot be effectively computed, and thus a blocklace B induces a DAG, with blocks as vertices in B and an edge among two vertices if the first includes a hash pointer to the second.

We say that a block b **observes** another block b', denoted $b \succeq b'$, if there is a path from block b to b'. If b is a p-block in a blocklace B, we say that miner p **observes** b' **in** B (Def. 16). We note that "observe" is the transitive closure of "acknowledge". Each miner maintains a local blocklace of blocks it created and received. With a correct miner p, any newly created p-block observes all the blocks in p's local blocklace.

The main violation a Byzantine miner q can perform is an **equivocation**, by creating a pair of q-blocks that do not observe each other (See Fig. 1.B). Such a miner q is an **equivocator** (Def. 17). If the payloads of the two blocks are financial transactions, the equivocation may represent an attempt at double-spending. As any p-block is cryptographically signed by p, an equivocation by p is a volitional fault of p, to which p can be held accountable.

When a block b observes another block b', and does not observe any equivocating block (a block b'' that together with b' forms an equivocation), we say that b **approves** b' (Def. 18). Note that a block b by a correct miner can observe two equivocating blocks b', b'', which means that b approves neither b' nor b'' (See Fig. 1.B). Block approval is not transitive. If b^+ approves b and b approves b', yet b^+ also observes b'' (which together with b' forms an equivocation), then b^+ does not approve b'.

Algorithm 1 Cordial Miners: Blocklace Utilities. Code for miner p.

```
    Local variables:
        struct block b:                                   ▷ The structure of a block b in a blocklace, Def. 14
            b.creator – the miner that created b
            b.payload – a set of transactions
            b.pointers – a possibly-empty set of hash pointers to other blocks
        blocklace ← {}                                    ▷ The local blocklace of miner p
 1: procedure create_block(d):       ▷ Add to blocklace a new block b pointing to its tips of depth ≤ d
 2:     new b                                             ▷ Allocate a new block structure
 3:     b.payload ← payload()        ▷ e.g., dequeue a payload from a queue of proposals (aka mempool)
 4:     b.creator ← p
 5:     b.pointers ← hash(tips), where tips are the tips of blocklace_prefix(d), at most two tips per miner
    ▷ Def. 19; two-tips limitation to prevent a Byzantine miner from flooding the blocklace before being
    excommunicated
 6:     return b
 7: procedure hash(b): return hash value of b                                          ▷ Def. 14
 8: procedure $b \succeq b'$:                                ▷ Def. 16, also refereed as b observes b'
 9:     return $\exists b_1, b_2, \ldots, b_k \in blocklace$, $k \geq 1$, s.t. $b_1 = b$, $b_k = b'$ and $\forall i \in [k-1]$: $hash(b_{i+1}) \in b_i.pointers$
10: procedure closure(B):  return $\{b' \in blocklace : b \in B \wedge b \succeq b'\}$   ▷ Def. 19. Also referred to as [B]. If
    $B = \{b\}$ is a singleton we use $[b]$ instead of $[\{b\}]$.
11: procedure equivocation($b_1, b_2$):                                       ▷ Def. 17, Fig. 1.B
12:     return $b_1.creator = b_2.creator \wedge b_1 \not\succeq b_2 \wedge b_2 \not\succeq b_1$
13: procedure equivocator(q, B):                        ▷ Def. 17, Fig. 1; more faults can be added
14:     return $(\exists b_1, b_2 \in B : b_1.creator = b_2.creator = q \wedge equivocation(b_1, b_2))$
15: procedure correct_block(b):                        ▷ See Def. 25; other conditions can be added
16:     return $\{b'.creator : hash(b') \in b.pointers\}$ is a supermajority $\wedge \neg equivocator(b.creator, [b])$
17: procedure approves($b, b_1$):  return $b_1 \in [b] \wedge \forall b_2 \in [b] : \neg equivocation(b_1, b_2)$   ▷ Def. 18, Fig. 1.C
18: procedure ratifies($B_1, b_2$):                                            ▷ Def. 22, Fig. 1.C
19:     return $\{b.creator : b \in [B_1] \wedge approves(b, b_2)\}$ is a supermajority
20: procedure super_ratifies($B_1, b_2$):                                      ▷ Def. 22, Fig. 1.A
21:     return $\{b.creator : b \in [B_1] \wedge ratifies([b], b_2)\}$ is a supermajority
22: procedure depth(b):
23:     return max $\{k : \exists b' \in blocklace$ with a path from $b$ to $b'$ of length $k\}$.      ▷ Def. 20
24: procedure blocklace_prefix(d): return $\{b \in blocklace : depth(b) \leq d\}$               ▷ Def. 20
25: procedure cordial_round(r):
26:     return $\{b.creator : b \in blocklace \wedge depth(b) = r\}$ is a supermajority          ▷ Def. 25
27: procedure completed_round( ):
28:     return max $\{r : cordial\_round(r)\}$
29: procedure last_block(p):                               ▷ The p-block with the highest round
30:     return $b \in blocklace$ s.t. $b.creator = p \wedge (\forall b' \in blocklace : b'.creator = p \implies b' \not\succ b)$
```

A miner p **approves** b' **in** a blocklace B, if p has a p-block b in B that approves b' (Def. 18). This holds even if p has a later p-block b^+ in B that observes an equivocation b' and b''. Namely, if miner p approves b' in B it also approves b' in any $B' \supset B$.

A miner p can approve both equivocating blocks b' and b'' in a blocklace B, but only if p is an equivocator. An example will be if B includes a p-block b that observes b' but not b'', and another block b^+ that observes b'' but not b', which can happen only if b and b^+ do not observe each other, namely form an equivocation.

The **closure** of a block b, denoted $[b]$, is the set of all blocks observed by b. The closure of a set of blocks B, denoted $[B]$, is the union of the closures of the blocks in B. A blocklace is **closed** if it does not contain "dangling pointers' (a pointer to a block that is not in the blocklace). In other words, B is closed if $B = [B]$. A block b is a **tip** of a blocklace B if there are no other blocks $b' \in B$ that observe b (Def. 19). The **depth** (or **round**) of a block b is the length of the longest path emanating from b. The **depth-d prefix of B**, denoted $B(d)$, is the set of all blocks with depth less than or equal to d. The **depth-d suffix of B**, is the set of all blocks with depth greater than d (Def. 20).

4.2 Blocklace Safety

Note that as equivocation is a fault, at most f miners may equivocate. Ensuring that the majority of correct miners approve a given block, requires approval from a **supermajority** of all miners, that is more than $\frac{n+f}{2}$ of the miners. A set of blocks is a **supermajority** if it includes blocks from a supermajority of miners (Def. 21). We show that there cannot be a supermajority approval of an equivocation.

A block b **ratifies** a block b' if the closure of b includes a supermajority of blocks that approve b'. A set of blocks B **super-ratifies** a block b', if it includes a supermajority of blocks that ratify b' (Def. 22 and Fig. 1).

The rounds in the blocklace are divided into **waves**, such that each wave has a fixed length of $w \geq 1$, defined as the **wavelength** (Def. 23), and the wave consists of all the blocks in those rounds. E.g., if the wavelength is 2, then the blocks in rounds 0 and 1 are in the first wave, and the blocks in rounds 3 and 4 are included in the second wave. We assume the existence of a leader selection function that chooses randomly for each wave w a single miner who will be the **leader** of that wave. A p-block b is a **leader block** of wave w if p is chosen as the leader of w and the blocklace contains b in the first round of w. E.g., if miner p is chosen as the leader of the first wave, and p has a block b in round 0, then b is the leader block of the first wave. We use leader blocks as part of the ordering function τ which is detailed in §5 and is used to totally order the blocklace.

Note that an equivocating leader can have several leader blocks in the same round. A leader block is **final** (Def. 24) if it is super-ratified within its wave, i.e., we say that the leader block b of round r is final if the blocklace prefix $B(r + w - 1)$ super-ratifies b.

The following notion of blocklace safety is the basis for the monotonicity of the blocklace ordering function τ, and hence for the safety of a protocol that uses τ for blocklace ordering.

▶ **Definition 2** (Blocklace Leader Safety). *A blocklace B is **leader-safe** if every final leader block in B is ratified by every subsequent leader block in B.*

A sufficient condition for blocklace leader safety is for every block in the blocklace to acknowledge blocks by at least a supermajority of miners (see Fig. 2). Such a block is a **cordial block** and a blocklace with only cordial blocks is a **cordial blocklace** (Def. 25). A **correct block** b is a p-block s.t. b is a cordial block and p does not equivocate in $[b]$ (Def 26).

▶ **Proposition 3.** *A cordial blocklace is leader-safe.*

4.3 Blocklace Liveness

Next, we discuss conditions that ensure blocklace leader liveness.

▶ **Definition 4** (Blocklace Leader Liveness). *A blocklace B is **leader-live** if for every block $b \in B$ by a miner not equivocating in B there is a final leader block in B that observes b.*

Given a blocklace, a set of miners P is (mutually) **disseminating** if every block by a miner in P is eventually observed by every miner in P (Def. 27). We show that dissemination is unbounded, meaning that if a set of miners P is disseminating in B then B is infinite, and in particular any suffix of B has blocks from any member of P. It follows that a cordial blocklace with a non-equivocating and disseminating supermajority of miners is leader-live (Fig. 3).

Algorithm 2 Cordial Miners: Ordering of a Blocklace with τ. Pseudocode for miner $p \in \Pi$, including Algorithms 1 & 4.

```
    Local Variable:
        $outputBlocks \leftarrow \{\}$
31: procedure $\tau()$:                                          ▷ Called from Algorithm 3
32:     $\tau'(last\_final\_leader())$
33: procedure $\tau'(b_1)$:
34:     if $b_1 \in outputBlocks \vee b_1 = \emptyset$ then return
35:     $b_2 \leftarrow previous\_ratified\_leader(b_1)$
36:     $\tau'(b_2)$                                                 ▷ Recursive call to $\tau'$
37:     output $xsort(b_1, [b_1] \setminus [b_2])$                 ▷ Output a new equivocation-free suffix
38:     $outputBlocks \leftarrow outputBlocks \cup xsort(b_1, [b_1] \setminus [b_2])$
39: procedure $xsort(b, B)$:                                     ▷ Exclude equivocations and sort
40:     return topological sort wrt $\succ$ of the set $\{b' \in B : approves(b, b')\}$
41: procedure $previous\_ratified\_leader(b_1)$:
42:     return $arg_{b \in R}$ max $depth(b)$
43:     where $R = \{b \in [b_1] \setminus \{b_1\} : b.creator = leader(depth(b)) \wedge ratifies([b_1], b)\}$
44: procedure $last\_final\_leader()$:                           ▷ Fig. 2
45:     return $arg_{u \in U}$ max $depth(u)$ where
46:     $U = \{b \in blocklace : b.creator = leader(depth(b)) \wedge final\_leader(b)\}$
47: procedure $final\_leader(b)$:                                ▷ Def. 24
48:     return $super\text{-}ratifies((blocklace\_prefix(depth(b) + w - 1), b)$
    procedure $leader()$ (Def. 23) and wavelength $w$ are defined in Alg. 4.
```

▶ **Proposition 5** (Blocklace Leader Liveness Condition)**.** *If $B \subset \mathcal{B}$ is a cordial blocklace with a non-equivocating and disseminating supermajority of miners, such that for every $r > 0$ there is a final leader block of round $r' > r$, then B is leader-live.*

5 Blocklace Ordering with τ

Here we present a deterministic function τ that, given a blocklace B, employs final leaders to topologically sort B into a sequence of its blocks, respecting $\succ$. The intention is that in a blocklace-based ordering consensus protocol, each miner would use τ to locally convert their partially-ordered blocklace into the totally-ordered output sequence of blocks.

The section concludes with Theorem 8, which provides sufficient conditions for the safety and liveness of any blocklace-based ordering consensus protocol that employs τ. The proof method is novel, in that it does not argue operationally, about events and their order in time, but rather about the properties of an infinite data structure – the blocklace. In the following section, we prove that the Cordial Miners protocols, which employ Alg. 2 that realizes τ, satisfy these conditions, and thus establish their safety and liveness. The operation of τ is depicted in Fig. 4.

We show that τ is monotonic, in that if it is repeatedly called with an ever-increasing blocklace then its output is an ever-increasing sequence of blocks. This monotonicity ensures finality, as it implies that any output will not be undone by a subsequent output. With τ, final leaders are the anchors of finality in the growing chain, each "writes history" backward till the preceding final leader.

The following recursive ordering function τ maps a blocklace into a sequence of blocks, excluding equivocations along the way. Formally, the entire sequence is computed backward from the last super-ratified leader, afresh by each application of τ. Practically, a sequence up to a super-ratified leader is final (Prop. 9) and hence can be cached, allowing the next call to τ with a new super-ratified leader to be computed backward only till the previously-cached super-ratified leader, while producing as output all the blocks approved by the new super-ratified leader (the approval ensures that the new fragment does not introduce equivocations) that are not observed by the previously-cached final leader.

▶ **Definition 6** (τ). *We assume a fixed topological sort function $xsort(b, B)$ (exclude and sort) that takes a block b and a blocklace B, and returns a sequence consistent with $\succ$ of all the blocks in B that are approved by b. The function $\tau : 2^{\mathcal{B}} \to \mathcal{B}^*$ is defined for a blocklace $B \subset \mathcal{B}$ backward, from the last output element to the first, as follows: If B has no final leaders then $\tau(B) := \Lambda$ (empty sequence). Else, let b be the last final leader in B. Then $\tau(B) := \tau''(b)$, where τ' is defined recursively:*

$$\tau'(b) := \begin{cases} xsort(b, [b]) & \text{if } [b] \text{ has no leader ratified by } b, \text{ else} \\ \tau'(b') \cdot xsort(b, [b] \setminus [b']) & \text{if } b' \text{ is the last leader} \\ & \text{ratified by } b \text{ in } [b] \end{cases}$$

Note that when τ' is called with a leader b, it makes a recursive call with a leader ratified by b, which is not necessarily super-ratified.

A pseudo-code implementation of τ is presented as Alg. 2. The algorithm is a literal implementation of the mathematics described above: It maintains *outputBlocks* that includes the prefix of the output τ that has already been computed. Upon adding a new block to its blocklace (Line 31), it computes the most recent final leader b_1 according to Definition 24, and applies τ to it, realizing the mathematical definition of τ (Def. 6), with the optimization, discussed above, that a recursive call with a block that was already output is returned. Hence the following proposition:

▶ **Proposition 7** (Correct implementation of τ). *The procedure τ in Alg. 2 correctly implements the function τ in Definition 6.*

The following theorem provides a sufficient condition for the safety and liveness (Def. 1) of any blocklace ordering consensus protocol that employs τ, and thus offers conditions for solving the problem defined in §2:

▶ **Theorem 8** (Sufficient Condition for the Safety and Liveness of a Blocklace-Based Ordering Consensus Protocol). *Assume a given blocklace-based consensus protocol that employs τ for ordering. If in every run of the protocol all correct miners have in the limit the same blocklace B that is leader-safe and leader-live, then the protocol is safe and live.*

Next, we provide a proof outline of Theorem 8.

τ Safety. A safe blocklace ensures a final leader is ratified by any subsequent leader, final or not. Hence the following:

▶ **Proposition 9** (Monotonicity of τ). *Let B be a cordial blocklace with a supermajority of correct miners. Then τ is monotonic wrt the superset relation among closed subsets of B, namely for any two closed blocklaces $B_2 \subseteq B_1 \subseteq B$, $\tau(B_2) \preceq \tau(B_1)$.*

The following proposition ensures that if there is a supermajority of correct miners, which jointly create a cordial blocklace, then the output sequences computed by any two miners based on their local blocklaces would be consistent. This establishes the safety of τ under these conditions.

▶ **Proposition 10** (τ Safety). *Let B be a blocklace with a supermajority of correct miners. Then for every $B_1, B_2 \subseteq B$, $\tau(B_1)$ and $\tau(B_2)$ are consistent.*

Algorithm 3 **Cordial Miners: Blocklace-Based Dissemination**
Code for miner p, including Algorithms 1, 2 & 4.

```
    Local variables:
        $r \leftarrow 0$                                   ▷ The current round of $p$, see Def. 20
49: upon receipt of $b : b.pointers \subseteq hash(blocklace) \wedge correct\_block(b)$ do ▷ Received "out of order" blocks are buffered; incorrect blocks are ignored
50:     $blocklace \leftarrow blocklace \cup \{b\}$
51:     $\tau()$                                           ▷ Defined in Algorithm 2
52:     if $completed\_round() \geq r$ then                ▷ Defined in Algorithm 1, line 27
53:         $es\_advance\_round()$       ▷ Advance round conditions for ES, no-op for asynchrony. Defined in Algorithm 4
54:         $b \leftarrow create\_block(completed\_round())$
55:         $r \leftarrow depth(b)$                        ▷ Advance round
56:         for $q \in \Pi$ do                             ▷ Cordial Dissemination
57:             send $\{b\} \cup blocklace\_prefix(r-2) \setminus [last\_block(q)]$ to $q$
```

Table 2 Cordial Miners' differences between Eventual Synchrony and Asynchrony.

Property	**Asynchrony**	**Eventual Synchrony**
Wavelength w:	5 (Line 58)	3 (Line 66)
Leader Selection:	Retrospective via coin toss (Line 61)	Prospective by a known order (Line 76)
Condition for advancing round:	None (Line 59)	*timeout* or finality conditions (Line 67)

τ Liveness. While τ does not output all the blocks in its input, as blocks not observed by the last final leader in its input are not in its output, the following observation and proposition set the conditions for τ liveness:

▶ **Observation 11** (τ output). *If a p-block $b \in B$ by a miner p not equivocating in B is observed by a final leader in B, then $b \in \tau(B)$.*

▶ **Proposition 12** (τ Liveness). *Let $B_1 \subset B_2 \subset \ldots$ be a sequence of finite blocklaces for which $B = \bigcup_{i \geq 1} B_i$ is a cordial leader-live blocklace. Then for every block $b \in B$ by a correct miner in B there is an $i \geq 1$ such that $b \in \tau(B_i)$.*

Thus, we conclude that the safety and liveness properties of τ carry over to Alg. 2.

Next, we prove that the two Cordial Miners consensus protocols – for eventual synchrony and asynchrony – satisfy the conditions of Theorem 8, and hence are safe and live.

6 The Cordial Miners Protocols

So far, we presented the blocklace and how a blocklace can be totally ordered using τ. Next, we show how miners disseminate their blocks to form a blocklace.

The shared components of the Cordial Miners protocols are specified via pseudocode in Algs. 1 (blocklace utilities), 2 (the ordering function τ), and 3 (dissemination). Alg. 4 details the differences between the Cordial Miners protocols for ES and asynchrony. We begin by explaining the dissemination protocol.

Algorithm 4 Cordial Miners: Specific Utilities. Code for miner p.

4.1 Procedures for Asynchrony

```
58: $w \leftarrow 5$
59: procedure $es\_advance\_round()$:                                         ▷ No-op
60:     return
61: procedure $leader(d)$:
62:     if $d \bmod w = 0$ then
63:         return $q \in \Pi$ via a shared coin tossed at round $d + w - 1$
64:     else
65:         return $\perp$
```

4.2 Procedures for Eventual Synchrony

```
66: $w \leftarrow 3$
67: procedure $es\_advance\_round()$:
68:     return max $r : cordial\_round(r) \wedge$                    ▷ Last cordial round, Algorithm 1
69:     $((r \bmod w = 0 \implies$            ▷ First round of the wave, leader is included in the round.
70:     $\exists b \in blocklace : (leader(r) = b.creator) \wedge$
71:     $((r \bmod w = 1 \implies$   ▷ Second round of the wave, round $r - 1$ leader is ratified by round $r$ blocks
72:     $\exists b \in blocklace : (leader(r-1) = b.creator \wedge ratifies(blocklace\_prefix(r), b))) \wedge$
73:     $((r \bmod w = 2 \implies$            ▷ Third round, round $r - 2$ leader is super-ratified by $r$ blocks
74:     $\exists b \in blocklace : (leader(r-2) = b.creator \wedge super\text{-}ratifies(blocklace\_prefix(r), b)))$
75:     $\vee\ timeout)$   ▷ Or timeout occurred. $timeout$ is measured from when round $r$ is cordial. This is $p$'s
    estimation of $\Delta$.
76: procedure $leader(d)$:
77:     if $d \bmod w = 0$ then
78:         return $q \in \Pi$ selected deterministically
79:     else
80:         return $\perp$
```

6.1 Dissemination (Alg. 3)

A correct block is buffered until it has no dangling pointers, and then it is received (Line 49). We prove that an equivocating miner eventually can only produce incorrect blocks (Def. 26) and therefore is eventually excommunicated by all correct miners. After including a received block in its local blocklace, a miner calls τ (Line 51), which outputs new blocks if the received block results in the blocklace having a new final leader block.

If there is a new completed round in the blocklace (Line 52), the miner creates a new block b (Line 54), computes the new round (Line 55), and sends b to its fellow miners. The package sent to miner q contains any blocks up to the previous round that p knows that q might not know, based on the last block received from q (Line 57). Note that as the network is reliable, **send** is defined to be idempotent, namely to send each block to each miner at most once.

We note that there is a tradeoff between latency and message complexity, and there is a range of possible optimizations and heuristics. These are discussed in [22]. Here, we present a version of Cordial Miners protocols in which every block is communicated among every pair of correct miners in the worst case.

6.2 Specific utilities (Alg. 4)

Overview. There are several differences between the Cordial Miners protocols for ES and asynchrony, which are specified in Alg. 4 and summarized in Tab. 2.

First, in asynchrony, each wave consists of 5 rounds, and the leader block in the first round is chosen randomly using a shared coin tossed in the last round of the wave, i.e., the leader election is retrospective. We expand on the coin below. In ES, each wave has 3 rounds, and the leader block is elected in advance using any deterministic prospective method, e.g., round robin.

The reason a wave in asynchrony is longer is to counter the adversary: If the adversary knows in advance the leader block in the first round of the wave, it can manipulate block arrival times s.t. a wave with a final leader block will never happen. We prove that by using such coin at the last round of the wave, the adversary cannot affect the probability the the leader block is final. In ES, a wave consists of three rounds. We prove that this is sufficient to allow super-ratification of the leader block, making it final in case the leader is an honest miner.

Another difference is if an honest miner waits before proceeding to the next round when the current round becomes cordial. In asynchrony, the miner proceeds immediately to the next round when it is cordial (Line 59). In ES, a miner advances to the next round after a round either if *timeout* passes, or conditions for leader block finality occur (Line 67). The conditions are: if this is the first round of a wave, then the round contains the leader block (Line 69). If this is the second round, then the miner advances immediately if the round has a supermajority of blocks that ratifies the leader block (Line 71), and lastly, if the third round of a wave has a supermajority of blocks that super-ratifies the leader block (Line 73). These conditions are to prevent the adversary from ordering the messages after GST, in particular, the leader block and the blocks that super-ratify it, as the leader is known in advance.

Algorithm walkthrough. The *leader* (Lines 61, 76) procedure, which is called as part of τ, is an implementation of Def. 23.

The Cordial Miners asynchrony protocol, for which $w = 5$ (Line 58), elects leaders retrospectively using a shared random coin. To elect the leader of round r, when $r \bmod 5 = 0$, all correct miners toss the coin in round $r+3$ and know in round $r+4$ the elected leader of round r, as follows. We assume two **shared random coin** functions: *toss_coin* and *combine_tosses*. The function $toss\_coin(p_s, d)$ takes the secret key p_s of miner $p \in \Pi$ and a round number $d \geq 0$ as input, and produces p's share of the coin of round d, $s_{p,d}$, as output. If the protocol needs to compute the shared random coin for round d, then $s_{p,d}$ is incorporated in the payload of the d-depth p-block of every correct miner p. The function $combine\_tosses(S, d)$ takes a set S of shares $s_{p,d}$, $d \geq 0$, for which $|\{p : s_{p,d} \in S\}| > f+1$, and returns a miner $q \in \Pi$. The properties of a similar function were presented in [21], which details how to implement such a coin as part of a distributed blocklace-like structure.

We formally define the shared coin in definition 28. Examples of such a coin implementation using threshold signatures [6, 23, 31] are in [10, 21]. The ES protocol elects leaders in a prospective manner via a fixed deterministic function, e.g., round-robin between the miners.

6.3 Correctness Proof Outline

The main theorem we prove is the following:

▶ **Theorem 13** (Cordial Miners Protocols Safety and Liveness). *The protocols for eventual synchrony and asynchrony specified in Algs. 1, 2, 3, & 4 are safe and live (Def. 1).*

We argue that in the limit the blocklaces of correct miners that participate in a run of a Cordial Miners protocol are identical, are leader-safe, and leader-live.

A formal description of blocklace-based protocols in terms of asynchronous multiagent transition systems with faults has been carried out in reference [28]. Here, we employ pseudocode, presented in Algorithms 1, 2, 3 & 4 to describe the correct behaviors of a miner in a protocol, and discuss only informally the implied multiagent transition system and its computations. A run of the protocol by the miners Π results in a sequence of configurations $\rho = c_0, c_1, \ldots$, each encoding the local state of each miner. A miner is *correct* in a run ρ if it behaves according to the pseudocode during ρ, *faulty* otherwise. As stated above, we assume that there are at most $f < n/3$ faulty miners in any run. We use $B_p(c)$ to denote the local blocklace of miner $p \in \Pi$ in configuration c, $B_p(\rho)$ to denote the blocklace of miner p in the limit, $B_p(\rho) := \bigcup_{c\in\rho} B_p(c)$, and $B(\rho)$ to denote the unions of the blocklaces of all correct miners in the limit, $B(\rho) := \bigcup_{p\in P} B_p(\rho)$, where $P \subseteq \Pi$ is the set of correct miners in run ρ.

We start by showing miner asynchrony (not to be confused with the model of asynchrony), that is, if a miner can create a block, then it can still create it regardless of additional blocks it receives from other miners. Miner asynchrony combined with the standard notion of *fairness*, that a transition that is enabled infinitely often in a run is eventually taken in the run, implies that once a Cordial Miners block creation transition is enabled then it will eventually be taken. We conclude that every miner p correct in a run produces the blocklace of the run, namely $B_p(\rho) = B(\rho)$. We can now argue the safety of the Cordial Miners protocols.

We now proceed to argue the liveness of the Cordial Miners protocols. We show that the Cordial Miners eventual synchrony protocol is leader-live with probability 1. We note that, following GST, the probability of a leader block being final is at least $\frac{|P|}{n}$, where $P \subseteq \Pi$ is the set of correct miners, and given that $w = 3$, if $\frac{|P|}{n} > \frac{2}{3}$, then the expected latency is at most $3/(2/3) = 4.5$ rounds.

The next proposition ensures that all correct miners eventually repel all equivocators and stop observing their blocks. We define an **equivocator-repelling** block recursively (Def. 29), through the set of blocks B that it acknowledges, terminating in an initial block, where $B = \emptyset$. Note that a block (or blocklace) that is equivocator-repelling may include equivocations, for example, two equivocating blocks each observed by a different block in B. However, once an equivocation by miner q is observed by a block b, q would be repelled: Any block that observes b would not acknowledge any q-block, preventing any further q-blocks from joining the blocklace. Also note that equivocators are eventually excommunicated since they eventually cannot produce correct blocks.

We argue in a lemma the existence of a blocklace **common core**, which is the blocklace-variant of the notion of a common core that appears in [3, 17]. Its proof is an adaptation to the cordial blocklace setting of the common core proof in [17], which in turn is derived from the proof of get-core in [3]. Fig. 5 illustrates its proof as well as the proof of the following Corollary about the existence of a super-ratified common core.

The lemma and corollary require an equivocators-free section of the blocklace, which may be the entire equivocation-free suffix of the blocklace as in the proof. But the proof also holds if there is a long enough stretch of rounds without equivocation, in which case a common core also exists. We conclude that if a Cordial Miners protocol relies on the common core for liveness dissemination, and cordiality are sufficient to ensure it. Finally, we complete the proof of liveness of the Cordial Miners protocols.

This concludes the proof outline that Cordial Miners is live and safe and thus completes the proof of Theorem 13.

7 Performance Analysis

We analyze the performance of Cordial Miners assuming the maximum number of Byzantine miners, i.e., $n = 3f + 1$. For the good case bit complexity, we assume $f \in O(1)$ and the network is synchronous.

Latency (See Table 1). Latency is defined as the number of blocklace rounds between every two consecutive final leaders, i.e., the number of blocklace rounds between two instances where τ outputs new blocks. This is also equivalent to the number of communication rounds since we do not use RB to disseminate blocks. The good case latency for both models is simply the wavelength.

For the expected case, in the asynchronous instance of the protocol, each wave w consists of 5 rounds. The probability that the leader block is final in the first round r of w, namely that a supermajority of the blocks in $r + 4$ each super-ratify the leader block at r is $\frac{2}{3}$. Therefore, in the expected case a leader block is final every 1.5 waves, and therefore the expected latency is $1.5w = 7.5$ rounds of communication.

The adversary can equivocate or not be cordial up to f times, but after each Byzantine process p equivocates, all correct processes eventually detect the equivocation and do not consider p's blocks as part of their cordial rounds when building the blocklace. Thus, in an infinite run, equivocations do not affect the overall expected latency.

In the ES version, each wave w consists of 3 rounds. The probability that the leader block is final is if the leader block is created by a correct miner, i.e., the probability is $\frac{2}{3}$, i.e., same as asynchrony. Thus, in the expected case, the latency is $1.5w = 4.5$ rounds.

Bit complexity. An equivocator is eventually excommunicated, and therefore eventually the number of equivocating blocks that are disseminated is limited. Each block in the blocklace is linear in size since it has a linear number of hash pointers to previous blocks. A Byzantine miner can cause the block it creates to be sent to all miners by all the other correct miners, causing the block's bit complexity to be $O(n^3)$ per such block. Thus, in the worst case, where $f \in \Theta(n)$, the asymptotic bit complexity is $O(n^3)$ per block. But, since the block size is $O(n)$, we can batch $O(n)$ transactions in it without increasing its asymptotic size. Therefore, we can amortize the bit complexity by a linear factor for each transaction, causing the amortized bit complexity per transaction to be $O(n^2)$ in the worst case.

For the good case in the ES version, where $f \in O(1)$ and the network is synchronous after GST, every block created by a correct miner is sent once from its creator to the other miners. Miners wait for *timeout* time after a round r is cordial before they move to the next round, which ensures that all blocks sent by correct miners in round r arrive to all other correct miners before they move to round $r+1$. Therefore, blocks by correct miners in round $r+1$ observe all blocks by correct miners in round r. Thus, the Byzantine miners can cause only a constant number of blocks per round to be sent by every correct miner to every other correct miner. Therefore, the bit complexity of sending each block in the good case is $O(n^2)$, and by batching $O(n)$ transaction per block, we get an amortized bit complexity of $O(n)$ per transaction.

8 Related Work

The use of a DAG-like structure to solve consensus has been introduced in previous works, especially in asynchronous networks. Hashgraph [4] builds an unstructured DAG, with each block containing two references to previous blocks, and on top of the DAG, the miners run an inefficient binary agreement protocol. This leads to expected exponential time complexity. Aleph [20] builds a structured round-based DAG, where miners proceed to the next round once they receive $2f + 1$ DAG vertices from other miners in the same round. On top of the DAG construction protocol, a binary agreement protocol decides on the order of vertices to commit. Blockmania [13] uses a variant of PBFT [11] in the ES model and also uses reliable broadcast to disseminate blocks. Both protocols have higher latency than Cordial Miners since they use RB. GHOST [32], IOTA [25], and Avalanche [26] are DAG protocols for the permissionless model.

As mentioned in the introduction, the two state-of-the-art DAG-based protocols are DAG-Rider [21] and Bullshark [33]. DAG-Rider is a BAB protocol for the asynchronous model in which the miners jointly build a DAG of blocks, with blocks as vertices and pointers to previously created blocks as edges, divided into strong and weak edges. Strong edges are used for the commit rule, and weak edges are used to ensure fairness. Narwhal [14] is an implementation based on DAG-Rider for a relaxed networking model and works well assuming messages arrival is not bounded, but also not controlled by the adversary. Tusk [14] is a similar consensus protocol to DAG-Rider built on top of Narwhal. Bullshark [33] is a variation of DAG-Rider designed for the ES model with about half the latency of DAG-Rider. Cordial Miners outperform these protocols in terms of latency (for the same message complexity). Other DAG-based protocols include [12, 18], which are for a non-Byzantine failure model.

Another category of Byzantine consensus protocols is Leader-based. Examples include PBFT [11], Tendermint [8], HotStuff [34, 24], and VABA [1]. In these protocols, a designated leader proposes a block, sends them to the miners, and collects votes on its proposal, and a Byzantine leader can result in wasted time in which no blocks are output. Another difference is that these protocols are unbalanced in terms of the network as the leader is in charge of disseminating its block, collecting votes, and disseminating them, while the other miners only need to vote. On the other hand, DAG-based protocols like Cordial Miners are symmetric in that all miners perform exactly the same tasks.

9 Conclusion

We presented Cordial Miners, a family of low-latency, high-efficiency consensus protocols with instances for eventual synchrony and asynchrony. Cordial Miners achieve that by forgoing Reliable Broadcast and using the blocklace for the three major tasks of consensus – dissemination, equivocation exclusion, and ordering.

References

1 Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically optimal validated asynchronous byzantine agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 337–346, 2019.

2 Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. Good-case latency of Byzantine broadcast: A complete categorization. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 331–341, 2021.

3 Hagit Attiya and Jennifer Welch. *Distributed computing: fundamentals, simulations, and advanced topics*, volume 19. John Wiley & Sons, 2004.

4 Leemon Baird. The swirlds Hashgraph consensus algorithm: Fair, fast, Byzantine fault tolerance. Report, Swirlds, 2016.

5 Michael Ben-Or, Ran Canetti, and Oded Goldreich. Asynchronous secure computation. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 52–61, 1993.

6 Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *International conference on the theory and application of cryptology and information security*, pages 514–532. Springer, 2001.

7 Gabriel Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

8 Ethan Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains.* PhD thesis, University of Guelph, 2016.

9 Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Annual International Cryptology Conference*, pages 524–541. Springer, 2001.

10 Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in Constantinople: Practical asynchronous byzantine agreement using cryptography. *Journal of Cryptology*, 18(3):219–246, 2005.

11 Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pages 173–186, New Orleans, Louisiana, USA, 1999. USENIX Association.

12 Gregory V Chockler, Nabil Huleihel, and Danny Dolev. An adaptive totally ordered multicast protocol that tolerates partitions. In *Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 237–246, 1998.

13 George Danezis and David Hrycyszyn. Blockmania: from block DAGs to consensus. *arXiv preprint arXiv:1809.01620*, 2018.

14 George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: a dag-based mempool and efficient bft consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 34–50, 2022.

15 Sourav Das, Zhuolun Xiang, and Ling Ren. Asynchronous data dissemination and its applications. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2705–2721, 2021.

16 Sourav Das, Zhuolun Xiang, and Ling Ren. Near-optimal balanced reliable broadcast and asynchronous verifiable information dispersal. *Cryptology ePrint Archive*, 2022.

17 Danny Dolev and Eli Gafni. Some garbage in-some garbage out: Asynchronous t-byzantine as asynchronous benign t-resilient system with fixed t-trojan-horse inputs. *arXiv preprint arXiv:1607.01210*, 2016.

18 Danny Dolev, Shlomo Kramer, and Dalia Malki. Early delivery totally ordered multicast in asynchronous environments. In *FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing*, pages 544–553. IEEE, 1993.

19 Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for byzantine agreement. *Journal of the ACM (JACM)*, 32(1):191–204, 1985.

20 Adam Gągol and Michał Świętek. Aleph: A leaderless, asynchronous, byzantine fault tolerant consensus protocol. *arXiv preprint arXiv:1810.05256*, 2018.

21 Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is dag. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 165–175, 2021.

22 Idit Keidar, Oded Naor, and Ehud Shapiro. Cordial miners: A family of simple, efficient and self-contained consensus protocols for every eventuality. *arXiv preprint arXiv:2205.09174*, 2022.

23 Benoît Libert, Marc Joye, and Moti Yung. Born and raised distributively: Fully distributed non-interactive adaptively-secure threshold signatures with short shares. *Theoretical Computer Science*, 645:1–24, 2016.

24 Dahlia Malkhi and Kartik Nayak. Hotstuff-2: Optimal two-phase responsive bft. *Cryptology ePrint Archive*, 2023.

25 Serguei Popov. The tangle. `https://assets.ctfassets.net/r1dr6vzfxhev/2t4uxvsIqk0EUau6g2sw0g/45eae33637ca92f85dd9f4a3a218e1ec/iota1_4_3.pdf`, 2018.

26 Team Rocket, Maofan Yin, Kevin Sekniqi, Robbert van Renesse, and Emin Gün Sirer. Scalable and probabilistic leaderless bft consensus through metastability. *arXiv preprint arXiv:1906.08936*, 2019.

27 Maria A Schett and George Danezis. Embedding a deterministic BFT protocol in a block DAG. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 177–186, 2021.

28 Ehud Shapiro. Multiagent transition systems: Protocol-stack mathematics for distributed computing. *arXiv preprint arXiv:2112.13650*, 2021.

29 Ehud Shapiro. Grassroots distributed systems: Concept, examples, implementation and applications. *arXiv preprint arXiv:2301.04391*, 2023.

30 Robert Shostak, Marshall Pease, and Leslie Lamport. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

31 Victor Shoup. Practical threshold signatures. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 207–220. Springer, 2000.

32 Yonatan Sompolinsky and Aviv Zohar. Secure high-rate transaction processing in Bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 507–527. Springer, 2015.

33 Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: Dag bft protocols made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2705–2718, 2022.

34 Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.

A Formal Model

The following is a mathematical formal definition of the cordial miners consensus protocols.

▶ **Definition 14** (Block, Acknowledge). *A **block** b is a triple $b = (p, a, H)$ signed by p, referred to as a p-**block**, s.t. $p \in \Pi$ is the miner that creates b, $a \in \mathcal{A}$ is the **payload** of b, and H is a finite set of hash pointers to blocks. Namely, for each $h \in H$, $h = hash(b')$ for some block b'. In which case we also say that b **acknowledges** b'. If $H = \emptyset$ then b is **initial**.*

▶ **Definition 15** (Blocklace). *Let $\mathcal{B}$ be the maximal set of blocks over Π, $\mathcal{A}$, and hash for which the induced directed graph $(\mathcal{B}, \mathcal{E})$ is acyclic. A **blocklace** over $\mathcal{A}$ is a set of blocks $B \subseteq \mathcal{B}$.*

▶ **Definition 16** ($\succ$, Observe). *Given two blocks b, b', the strict partial order $\succ$ is defined by $b' \succ b$ if there is a nonempty path from b' to b. A block b' **observes** b if $b' \succeq b$. Given a blocklace B, Miner p **observes** b **in** B if there is a p-block $b' \in B$ that observes b. A group of miners $Q \subseteq \Pi$ **observes** b **in** B if every miner $p \in Q$ observes b.*

▶ **Definition 17** (Equivocation, Equivocator). *A pair of p-blocks $b \neq b' \in \mathcal{B}$, $p \in \Pi$, form an **equivocation** by p if they are not consistent wrt $\succ$, namely $b' \not\succ b$ and $b \not\succ b'$. A miner p is an **equivocator in** B, $equivocator(p, B)$, if B has an equivocation by p.*

▶ **Definition 18** (Approval). *Given blocks $b, b' \in \mathcal{B}$, the block b **approves** b' if b observes b' and does not observe any block b'' that together with b' forms an equivocation. A miner $p \in \Pi$ **approves** b' **in** B if there is a p-block $b \in B$ that approves b'. A set of miners $Q \subseteq \Pi$ **approve** b' **in** B if every miner $p \in Q$ approves b' in B.*

▶ **Definition 19** (Closure, Closed, Tip). *The **closure of** $b \in \mathcal{B}$ **wrt** $\succ$ is the set $[b] := \{b' \in \mathcal{B} : b \succeq b'\}$. The **closure of** $B \subset \mathcal{B}$ **wrt** $\succ$ is the set $[B] := \bigcup_{b \in B}[b]$. A blocklace $B \subseteq \mathcal{B}$ is **closed** if $B = [B]$. A block $b \in \mathcal{B}$ is a **tip** of B if $b \notin [B \setminus \{b\}]$.*

▶ **Definition 20** (Block Depth/Round, Blocklace Prefix & Suffix). *The **depth** (or **round**) of a block $b \in \mathcal{B}$, $depth(b)$, is the maximal length of any path of pointers emanating from b. For a blocklace $B \subseteq \mathcal{B}$ and $d \geq 0$, the **depth-d prefix of** B is $B(d) := \{b \in B : depth(b) \leq d\}$, and the **depth-$d$ suffix of** B is $\bar{B}(d) := B \setminus B(d)$.*

▶ **Definition 21** (Supermajority). *A set of miners $P \subset \Pi$ is a **supermajority** if $|P| > \frac{n+f}{2}$. A set of blocks B is a **supermajority** if the set of miners $P = \{p \in \Pi : \exists b \in B \text{ is a } p\text{-block}\}$ is a supermajority.*

▶ **Definition 22** (Ratified and Super-Ratified Block). *A block $b \in \mathcal{B}$ is (i) **ratified by a set of blocks** $B \subseteq \mathcal{B}$, if $[B]$ includes a supermajority of blocks that approve b; (ii) **ratified by a block** b if it is ratified by the set of blocks $[b]$; and (iii) **super-ratified** by blocklace $B \subset \mathcal{B}$ if $[B]$ includes a supermajority of blocks, each of which ratifies b*

▶ **Definition 23** (Wavelength, Leader Selection Function, Leader Block). *Given a **wavelength** $w \geq 1$, a **leader selection function** is a partial function $l : \mathbb{N} \mapsto \Pi$ satisfying (i) **coverage:** $\forall r \in \mathbb{N}$: $l(r) \in \Pi$ if r **mod** $w = 0$ else $l(r) = \bot$ and (ii) **fairness:** with probability 1 $\forall r \in \mathbb{N}, p \in \Pi\ \exists r' > r : l(r') = p$. A p-block b is a **leader block** if $l(depth(b)) = p$.*

▶ **Definition 24** (Final Leader Block). *Let $B \subseteq \mathcal{B}$ be a blocklace. A leader block $b \in B$ of round r is **final** in B if it is super-ratified in $B(r + w - 1)$.*

▶ **Definition 25** (Cordial Block, Blocklace). *A block $b \in \mathcal{B}$ of round r is **cordial** if $r = 1$ or it acknowledges blocks by a supermajority of miners of round $r - 1$. A blocklace $B \subset \mathcal{B}$ is **cordial** if all its blocks are cordial.*

▶ **Definition 26.** *A p-block $b \in \mathcal{B}$ in **correct**, if it is cordial and p does doe equivocate in $[b]$.*

▶ **Definition 27** (Disseminating). *Given a blocklace $B \subseteq \mathcal{B}$, a set of miners $P \subseteq \Pi$ is **mutually disseminating** in B, or **disseminating** for short, if for any $p, q \in P$ and any p-block $b \in B$ there is a q-block $b' \in B$ such that $b' \succ b$. The blocklace B is **disseminating** if it has a disseminating supermajority.*

▶ **Definition 28** (Shared Random Coin). *Assume some $d > 0$ and let $S = \{toss\_coin(p_s, d) : p \in P\}$ for a set of miners $P \subseteq \Pi$, $|P| > f + 1$. For the shared random coin, the function combine_tosses has the following properties:*

Agreement *If both $S', S'' \subseteq S$ and both $|S'|, |S''| > f + 1$, then $combine\_tosses(S', d) = combine\_tosses(S'', d)$*

Termination *$combine\_tosses(S, d) \in \Pi$.*

Fairness *The coin is fair, i.e., for every set S computed as above and any $p \in \Pi$, the probability that $p = combine\_tosses(S, d)$ is $\frac{1}{n}$.*

Unpredictability *If $S' \subset S$, $|S'| < f + 1$, then the probability that the adversary can use S' to guess the value of $combine\_tosses(S, d)$ is less than $\frac{1}{n} + \epsilon$.*

▶ **Definition 29** (Equivocator-Repelling). *Let $b \in \mathcal{B}$ be a p-block, $p \in \Pi$, that acknowledges a set of blocks $B \subset \mathcal{B}$. Then b is **equivocator-repelling** if p does not equivocate in $[b]$ and all blocks in B are equivocator-repelling. A blocklace B is **equivocator-repelling** if every block $b \in B$ is equivocator-repelling.*

B Figures

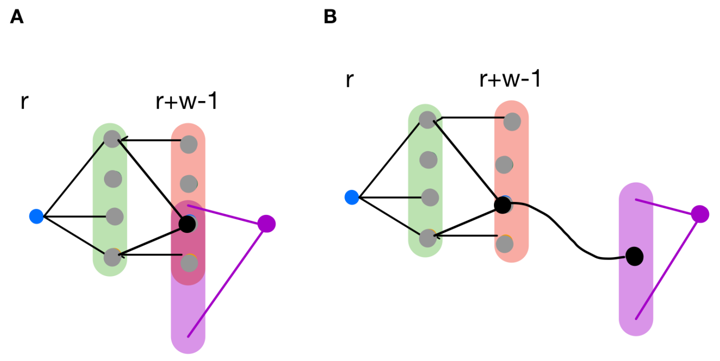


Figure 2 Finality of a Super-Ratified Leader (Definition 24): Assume that a leader block (blue dot) is super-ratified. A ratifying supermajority is represented by a thick red line, each member of which observes a possibly different approving supermajority represented by a green thick line. We show that the blue leader is ratified by any subsequent cordial leader. (A) The successive cordial leader (purple dot) is one round following the ratifying supermajority. Being cordial, it observes a supermajority (thick purple line) that must have an intersection (black dot) with the ratifying supermajority, hence it observes an approving supermajority and thus ratifies the blue leader. (B) A successive leader is more than one round following the ratifying supermajority. Being cordial, it observes a supermajority (thick purple line). There must be a correct miner common to the purple and red supermajority, with blocks in both (black dots); being a correct miner, its later block observes the earlier block (black line). Hence the purple leader observes the approving supermajority (via black lines) and hence ratifies the blue leader.

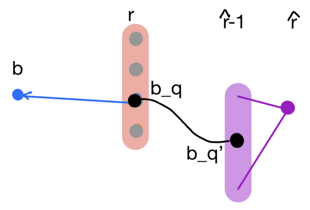


Figure 3 Liveness Condition, Proposition 5.

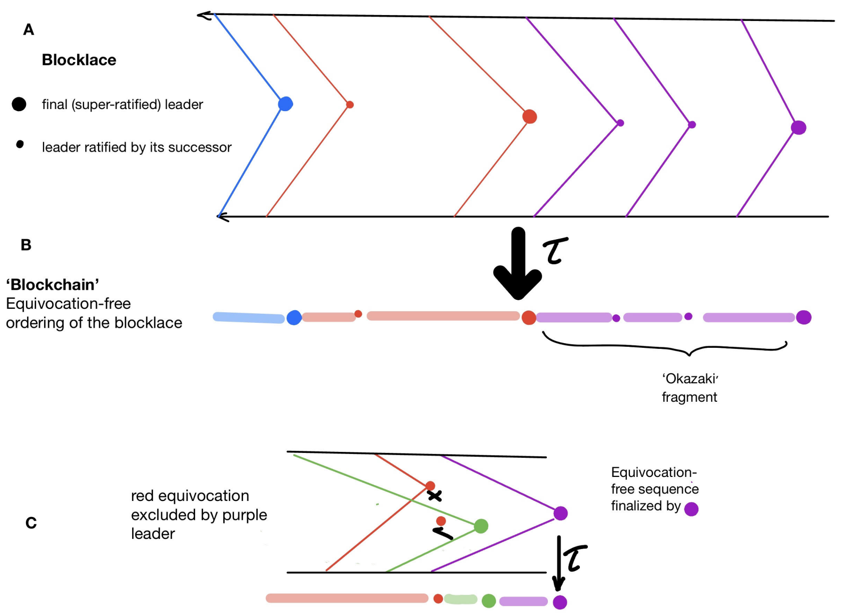
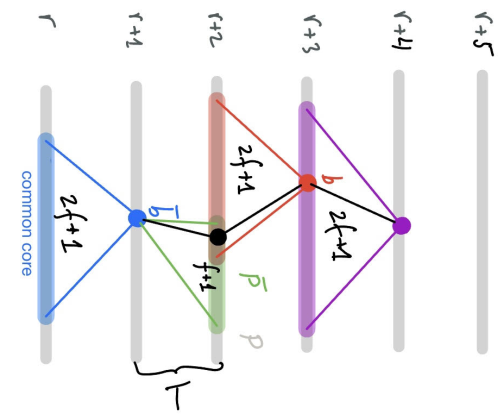
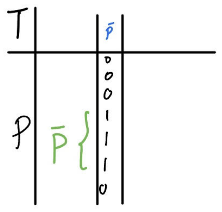

A
Blocklace
● final (super-ratified) leader
• leader ratified by its successor
B
τ
'Blockchain'
Equivocation-free ordering of the blocklace
'Okazaki' fragment
C
red equivocation excluded by purple leader
τ
Equivocation-free sequence finalized by ●

Figure 4 The Operation of τ, Safety and Liveness: (A) *The Input of τ*: A blocklace with final leaders (large dots) and leaders ratified by their successors (small dots). Each leader observes the portion of the blocklace below it (including the lines emanating from it). (B) *The Output of τ*: A sequence of blocks consisting of fragments. The sequence of fragments is computed recursively backward, starting from the last final leader, and back from each leader to the previous leader it ratifies. The input to computing the fragment consists of the portion of the blocklace observed by the current leader but not observed by the previous ratified leader. The output from each fragment is a sequence of blocks computed forward by topological sort of the input blocklace fragment, respecting $\succ$ and using the leader of the fragment to resolve and exclude equivocations. Final leaders are final, hence the backward computation starting from the last purple final leader need not proceed beyond the recursive call to the previous red final leader, as the output sequence up to the previous final leader has already been computed by the previous invocation of τ. **Safety Requirement:** A final leader (large dot) is ratified by any subsequent leader (large or small dot). **Liveness Requirement:** Any leader will eventually have a subsequent final leader (large dot) with probability 1. (C) *Leader-Based Equivocation Exclusion*: The green fragment created by the green leader includes the V-marked red block, since the green leader does not observe the red equivocation. However, the red X-marked red block is excluded from the purple fragment created by the purple leader, since the purple leader observes the equivocation among the two red blocks.

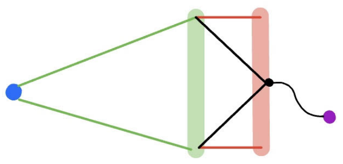
A



B



C



■ **Figure 5 Common Core, Ratified Common Core, Safety and Liveness of Decision Rule for Asynchrony:** (A) Rounds r to $r+3$ relate to the proof of the existence of a common core at round r is established. Round $r+4$ relates to establishing that all cordial blocks at round $r+4$ ratify all members of the common core of round r via a supermajority at round $r+3$. (B) *The common-core table* T used in the proof to relate rounds $r+1$ and $r+2$. (C) *The decision rule for asynchrony*: Protocol wavelength is 5. **Liveness:** Common-core ensures that the blue leader at round r is super-ratified by a red supermajority at round $r+4$ with probability $\frac{2f+1}{3f+1}$, thus ensuring liveness and expected latency of 6 rounds. **Safety:** A blue leader is approved by every cordial block at round $r+3$ (green) and hence is ratified by every cordial block at round $r+4$ (red) and beyond.

Fast Reconfiguration for Programmable Matter

Irina Kostitsyna
TU Eindhoven, The Netherlands

Tom Peters
TU Eindhoven, The Netherlands

Bettina Speckmann
TU Eindhoven, The Netherlands

Abstract

The concept of programmable matter envisions a very large number of tiny and simple robot particles forming a smart material. Even though the particles are restricted to local communication, local movement, and simple computation, their actions can nevertheless result in the global change of the material's physical properties and geometry.

A fundamental algorithmic task for programmable matter is to achieve global shape reconfiguration by specifying local behavior of the particles. In this paper we describe a new approach for shape reconfiguration in the *amoebot* model. The amoebot model is a distributed model which significantly restricts memory, computing, and communication capacity of the individual particles. Thus the challenge lies in coordinating the actions of particles to produce the desired behavior of the global system.

Our reconfiguration algorithm is the first algorithm that does not use a canonical intermediate configuration when transforming between arbitrary shapes. We introduce new geometric primitives for amoebots and show how to reconfigure particle systems, using these primitives, in a linear number of activation rounds in the worst case. In practice, our method exploits the geometry of the symmetric difference between input and output shape: it minimizes unnecessary disassembly and reassembly of the particle system when the symmetric difference between the initial and the target shapes is small. Furthermore, our reconfiguration algorithm moves the particles over as many parallel shortest paths as the problem instance allows.

2012 ACM Subject Classification Theory of computation → Computational geometry; Theory of computation → Self-organization

Keywords and phrases Programmable matter, amoebot model, shape reconfiguration

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.27

Related Version *Full Version*: `https://arxiv.org/abs/2202.11663` [22]

1 Introduction

Programmable matter is a smart material composed of a large quantity of robot particles capable of communicating locally, performing simple computation, and, based on the outcome of this computation, changing their physical properties. Particles can move through a programmable matter system by changing their geometry and attaching to (and detaching from) neighboring particles. By instructing the particles to change their local adjacencies, we can program a particle system to reconfigure its global shape. Shape assembly and reconfiguration of particle systems have attracted a lot of interest in the past decade and a variety of specific models have been proposed [5, 20, 25, 28, 18, 12, 24, 26]. We focus on the *amoebot* model [13], which we briefly introduce below. Here, the particles are modeled as independent agents collaboratively working towards a common goal in a distributed fashion. The model significantly restricts computing and communication capacity of the individual particles, and thus the challenge of programming a system lies in coordinating local actions of particles to produce a desired behavior of the global system.

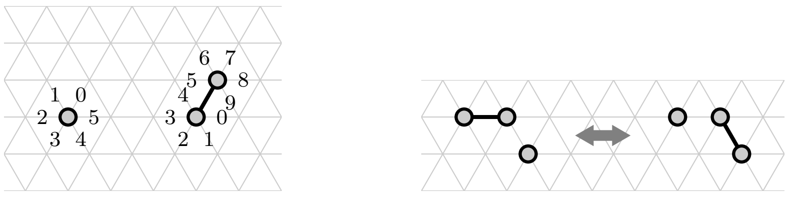


Figure 1 Left: particles with ports labeled, in contracted and expanded state. Right: handover operation between two particles.

A fundamental problem for programmable matter is shape reconfiguration. To solve it we need to design an algorithm for each particle to execute, such that, as a result, the programmable matter system as a whole reconfigures into the desired target shape. Existing solutions first build an intermediate *canonical configuration* (usually, a line or a triangle) and then build the target shape from that intermediate configuration [14, 16]. However, in many scenarios, such as shape repair, completely deconstructing a structure only to build a very similar one, is clearly not the most efficient strategy.

We propose the first approach for shape reconfiguration that does not use a canonical intermediate configuration when transforming between two arbitrary shapes. Our algorithm exploits the geometry of the symmetric difference between the input shape I and the target shape T. Specifically, we move the particles from $I \setminus T$ to $T \setminus I$ along shortest paths through the overlap $I \cap T$ over as many parallel shortest paths as the problem instance allows. In the worst case our algorithm works as well as existing solutions. However, in practice, our approach is significantly more efficient when the symmetric difference between the initial and the target shape is small.

Amoebot model. Particles in the amoebot model occupy nodes of a plane triangular grid G. A particle can occupy either a single node or a pair of adjacent nodes: the particle is contracted and expanded, respectively. The particles have constant memory space, and thus have limited computational power. They are disoriented (no common notion of orientation) and there is no consensus on chirality (clockwise or counter-clockwise). The particles have no ids and execute the same algorithm, i.e. they are identical. They can reference their neighbors using six (for contracted) or ten (for expanded particles) *port identifiers* that are ordered clockwise or counter-clockwise modulo six or ten (see Figure 1 (left)). Particles can communicate with direct neighbors by sending messages over the ports. Refer to Daymude et al. [10] for additional details.

Particles can move in two different ways: a contracted particle can *expand* into an adjacent empty node of the grid, and an expanded particle can *contract* into one of the nodes it currently occupies. Each node of G can be occupied by at most one particle, and we require that the particle system stays connected at all times. To preserve connectivity more easily, we allow a *handover* variant of both move types, a simultaneous expansion and contraction of two neighboring particles using the same node (Figure 1 (right)). A handover can be initiated by any of the two particles; if it is initiated by the expanded particle, we say it *pulls* its contracted neighbor, otherwise we say that it *pushes* its expanded neighbor.

Particles operate in activation cycles: when activated, they can read from the memory of their immediate neighbors, compute, send constant size messages to their neighbors, and perform a move operation. Particles are activated by an asynchronous adversarial but fair scheduler (at any moment in time t, any particle must be activated at some time in the future $t' > t$). If two particles are attempting conflicting actions (e.g., expanding into the same node), the conflict is resolved by the scheduler arbitrarily, and exactly one of these actions succeeds. We perform running time analysis in terms of the number of *rounds*: time intervals in which all particles have been activated at least once.

We say that a *particle configuration* $\mathcal{P}$ is the set of all particles and their internal states. Let $G_{\mathcal{P}}$ be the subgraph of the triangular grid G induced by the nodes occupied by particles in $\mathcal{P}$. Let a *hole* in $\mathcal{P}$ be any interior face of $G_{\mathcal{P}}$ with more than three vertices. We say a particle configuration $\mathcal{P}$ is *connected* if there exists a path in $G_{\mathcal{P}}$ between any two particles. A particle configuration $\mathcal{P}$ is *simply connected* if it is connected and has no holes.

Related work. The amoebot model, which is both natural and versatile, was introduced by Derakhshandeh et al. [13] in 2014 and has recently gained popularity in the algorithmic community. A number of algorithmic primitives, such as leader election [15, 16, 17], spanning forests [15], and distributed counters [8, 27], were developed to support algorithm design.

Derakhshandeh et al. [14] designed a reconfiguration algorithm for an amoebot system which starts from particles forming a large triangle and targets a shape consisting of a constant number of unit triangles (such that their description fits into the memory of a single particle). In their approach the initial large triangle is partitioned into the unit triangles, which move in a coordinated manner to their corresponding position within the target shape. Derakhshandeh et al. make some assumptions on the model, including a sequential activation scheduler (at every moment in time only one particle can be active), access of particles to randomization, and common particle chirality. Due to these assumptions, and the fact that the initial shape is compact, the reconfiguration process takes $O(\sqrt{n})$ number of rounds for a system with n particles.

Di Luna et al. [16] were the first to reconfigure an input particle system into a line (a canonical intermediate shape). They then simulate a Turing machine on this line, and use the output of the computation to direct the construction of the target structure in $O(n \log n)$ rounds. Their main goal was to lift some of the simplifying assumptions on the model of [14]: their algorithm works under a synchronous scheduler, is deterministic, does not rely on the particles having common chirality, and requires only the initial structure to be simply connected. However, just as the work by Derakhshandeh et al. [14], their method only works for structures of constant description size.

Cannon et al. [4] consider a stochastic variation of the amoebot model. Viewed as an evolving Markov chain, the particles make probabilistic decisions based on the structure of their local neighborhoods. In this variant, there exist solutions for compressing a system into a dense structure [4], simulating ant behavior to build a shortcut bridge between two locations [1], and separating a system into multiple components by the *color* of particles [3].

Problem description. An instance of the *reconfiguration problem* consists of a pair of simply connected shapes (I, T) embedded in the grid G (see Figure 2). The goal is to transform the initial shape I into the target shape T. Initially, all particles in I are contracted. The problem is solved when there is a contracted particle occupying every node of T.

We make a few assumptions on the input and on the model. Most of our assumptions fall into at least one of the following categories: they are natural for the problem statement, they can be lifted with extra care, or they are not more restrictive than existing work.

Assumptions on I and T. We assume that the input shape I and the target shape T have the same number n of nodes. We call $I \cap T$ the *core* of the system, the particles in $I \setminus T$ the *supply*, and the particles in $T \setminus I$ the *demand*. In our algorithm, the core nodes are always occupied by particles, and the supply particles move through the core to the demand.

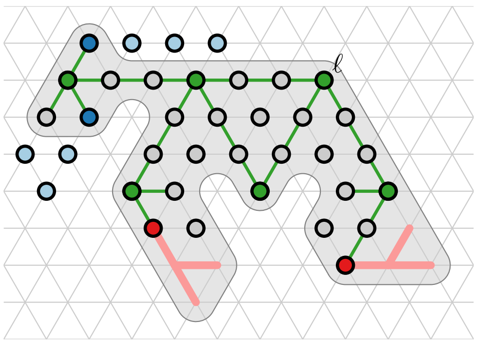


Figure 2 The particles form the initial shape I. The target shape T is shaded in gray. Supply particles are blue, supply roots dark blue. Demand roots (red) store spanning trees of their demand components. The graph G_L on the coarse grid is shown in green and leader ℓ is marked. Particles on G_L that are green are grid nodes, other particles on G_L are edge nodes.

We assume that

A1 the core $I \cap T$ is a non-empty simply connected component. This is a natural assumption in the shape repair scenario when the symmetric difference between the initial and the target shape is small.

A2 each demand component D of $T \setminus I$ has a constant description complexity.

In Section 6 we discuss a possible strategy for lifting A1. Below A6 we explain why A2 is not more restrictive than the input assumptions in the state-of-the art.

Assumptions on initial particle state. A standard assumption is that initially all particles have the same state. In this paper we assume that a preprocessing step has encoded T into the particle system and thus the states of some particles have been modified from the initial state. The reasons for this assumption are twofold: first, to limit the scope of this paper we chose to focus on the reconfiguration process; and second, the encoding preprocessing step, whose specification we omit, can be derived in a straightforward manner from existing primitives and algorithms. Below we describe more precisely what our assumptions are on the outcome of the preprocessing step.

To facilitate the navigation of particles through the core $I \cap T$, and in particular, to simplify the crossings of different flows of particles, we use a coarsened (by a factor of three) version of the triangular grid. Let G_L be the intersection of the coarse grid with $I \cap T$ (the green nodes Figure 2). We assume that after the preprocessing step

A3 a coarsened grid G_L has been computed, by definition every node in the core is either in G_L or is adjacent to a node in G_L. For simplicity of presentation we assume that the shape of the core is such that G_L is connected;

A4 particles know whether they belong to the supply $I \setminus T$, the demand $I \cap T$, or to the core $I \cap T$;

A5 every connected component C of the supply has a representative particle in the core $I \cap T$ adjacent to C, the *supply root* of C; correspondingly, every connected component D of the demand has one designated particle from the core adjacent to D, the *demand root* of D;

A6 each demand root d stores a complete spanning tree rooted at d of the corresponding demand component D in its memory explicitly or implicitly, by encoding construction rules for D.

If the core does not naturally induce a connected G_L, we can restore connectivity by locally deviating from the grid lines of the coarse grid and adapting the construction and use of G_L accordingly. Assumptions A2 and A4–A6 allow us to focus our presentation on the reconfiguration algorithm itself. Encoding a general target shape into the initial structure remains a challenging open problem. However, compared to prior work, our assumptions are not very strong: all other existing algorithms support only fixed, simple target shapes which can be easily encoded in the initial shape. For example, Derakhshandeh et al. [14] assume that the initial shape I is a large triangle and that the target shape T consists of only a constant number of unit triangles. Here a leader particle can easily compute the information necessary for the pre-processing step and broadcast it through the system.

Contribution and organization. We present the first reconfiguration algorithm for programmable matter that does not use a canonical intermediate configuration. Instead, our algorithm introduces new geometric primitives for amoebots which allow us to route particles in parallel from supply to demand via the core of the particle system. A fundamental building block of our approach are *feather trees* [21] which are a special type of *shortest path trees* (SP-trees). SP-trees arrange particles in a tree structure such that the paths from leaves to the root are shortest paths through the particle structure. The unique structure of feather trees allows us to use multiple overlapping trees in the particle system to enable particle navigation along shortest paths between multiple sources of supply and demand. In Section 2 we give all necessary definitions and show how to efficiently construct SP-trees and feather trees using a grid variant of shortest path maps [23].

In Section 3 we then show how to use feather trees to construct a *supply graph*, which directs the movement of particles from supply to demand. In Section 4 we describe in detail how to navigate the supply graph and also discuss the coarse grid G_L which we use to ensure the proper crossing of different flows of particles travelling in the supply graph. Finally, in Section 5 we summarize and analyze our complete algorithm for the reconfiguration problem. We show that using a sequential scheduler we can solve the particle reconfiguration problem in $O(n)$ activation rounds. When using an asynchronous scheduler our algorithm takes $O(n)$ rounds in expectation, and $O(n \log n)$ rounds with high probability. All omitted proofs and missing details can be found in the full version of our paper [22].

In the worst case our algorithm is as fast as existing algorithms, but in practice our method exploits the geometry of the symmetric difference between input and output shape: it minimizes unnecessary disassembly and reassembly of the particle system. Furthermore, if the configuration of the particle system is such that the feather trees are balanced with respect to the amount of supply particles in each sub-tree, then our algorithm finishes in a number of rounds close to the diameter of the system (instead of the number of particles). Our reconfiguration algorithm also moves the particles over as many parallel shortest paths as the problem instance allows. In Section 6 we discuss these features in more detail and also sketch future work.

2 Shortest path trees

To solve the particle reconfiguration problem, we need to coordinate the movement of the particles from $I \setminus T$ to $T \setminus I$. Among the previously proposed primitives for amoebot coordination is the *spanning forest primitive* [15] which organizes the particles into trees to facilitate movement while preserving connectivity. The root of a tree initiates the movement, and the remaining particles follow via handovers between parents and children. However, the

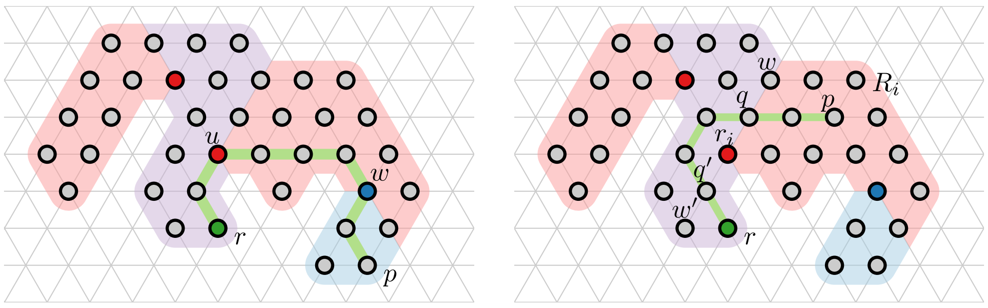


Figure 3 Shortest path map of node r. Any shortest path between r and p must pass through the roots of the respective SPM regions (u, w, and r_i). The region R_0 (in purple) consists of the particles $\mathcal{P}$-visible to r. The red and the blue particles are the roots of the corresponding SPM regions. Right: any path not going through the root of a visibility region can be shortened.

spanning forest primitive does not impose any additional structure on the resulting spanning trees. We propose to use a special kind of shortest path trees (SP-trees), called *feather trees*, which were briefly introduced in [21].

To ensure that all paths in the tree are shortest, we need to control the growth of the tree. One way to do so, is to use breadth-first search together with a token passing scheme, which ensures synchronization between growing layers of the tree, this is also known as a β-synchronizer [2]. See the full version of our paper for details [22].

▶ **Lemma 1.** *Given a connected particle configuration $\mathcal{P}$ with n particles, we can create an SP-tree using at most $O(n^2)$ rounds.*

2.1 Efficient SP-trees

To create SP-trees more efficiently for simply connected particle systems, we describe a version of the shortest path map (SPM) data structure [23] on the grid (see Figure 3 (left)). We say that a particle $q \in \mathcal{P}$ is *$\mathcal{P}$-visible* from a particle $p \in \mathcal{P}$ if there exists a shortest path from p to q in G that is contained in $G_{\mathcal{P}}$. This definition of visibility is closely related to staircase visibility in rectilinear polygons [6, 19]. Let $\mathcal{P}$ be simply connected, and let $R_0 \subseteq \mathcal{P}$ be the subconfiguration of all particles $\mathcal{P}$-visible from some particle r. By analogy with the geometric SPM, we refer to R_0 as a *region*. If $R_0 = \mathcal{P}$ then SPM(r) is simply R_0. Otherwise, consider the connected components $\{\mathcal{R}_1, \mathcal{R}_2, \dots\}$ of $\mathcal{P} \setminus R_0$. The *window* of $\mathcal{R}_i$ is a maximal straight-line chain of particles in R_0, each of which is adjacent to a particle in $\mathcal{R}_i$ (e.g., in Figure 3 (right), chain (r_i, w) is a window). Denote by r_i the closest particle to r of the window W_i of $\mathcal{R}_i$. Then SPM(r) is recursively defined as the union of R_0 and SPM(r_i) in $\mathcal{R}_i \cup W_i$ for all i. Let $R_i \subseteq \mathcal{R}_i \cup W_i$ be the set of particles $\mathcal{P}$-visible from r_i. We call R_i the *visibility region* of r_i, and r_i the *root* of R_i. Note that by our definition the particles of a window between two adjacent regions of a shortest path map belong to both regions.

▶ **Lemma 2.** *Let r_i be the root of a visibility region R_i in a particle configuration $\mathcal{P}$. For any particle p in R_i, the shortest path from r to p in $\mathcal{P}$ passes through r_i.*

▶ **Corollary 3.** *Any shortest path π between r and any other particle p in $\mathcal{P}$ must pass through the roots of the SPM regions that π crosses.*

If a particle p is $\mathcal{P}$-visible from r then there is a 60°-*angle monotone path* [11] from r to p in $G_{\mathcal{P}}$. That is, there exists a 60°-cone in a fixed orientation, such that for each particle q on the path from q to p lies completely inside this cone translated to q (see Figure 4 (left)).

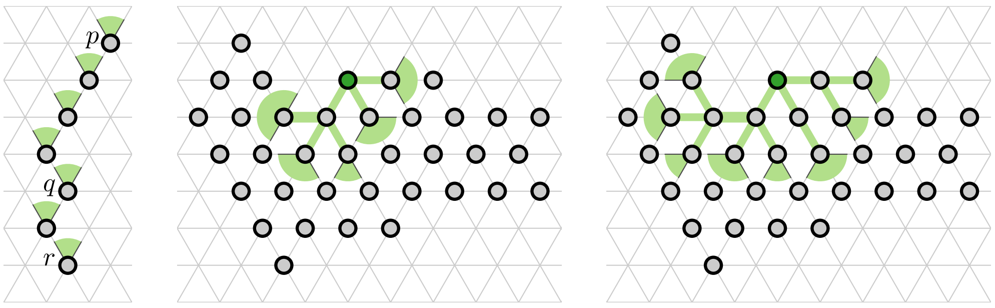


Figure 4 Left: An angle monotone path from r to p. For every particle q, the remainder of the path lies in a $60°$-cone. Middle: Growing an SP-tree using cones of directions. The particle on the left just extended its cone to $180°$. Right: A couple activations later.

We use a version of such cones to grow an SP-tree efficiently. Each node that is already included in the tree carries a cone of valid growth directions (see Figure 4 (middle)). When a leaf of the tree is activated it includes any neighbors into the tree which are not part of the tree yet and lie within the cone. A cone is defined as an interval of ports. The cone of the root r contains all six ports. When a new particle q is included in the tree, then its parent p assigns a particular cone of directions to q. Assume parent p has cone c and that q is connected to p via port i of p. By definition $i \in c$, since otherwise p would not include q into the tree. We intersect c with the $120°$-cone $[i-1, i+1]$ and pass the resulting cone c' on to q. (Recall that the arithmetic operations on the ports are performed modulo 6.) When doing so we translate c' into the local coordinate system of q such that the cone always includes the same global directions. This simple rule for cone assignments grows an SP-tree in the visibility region of the root r and it does so in a linear number of rounds.

▶ **Lemma 4.** *Given a particle configuration $\mathcal{P}$ with diameter d which is $\mathcal{P}$-visible from a particle $r \in \mathcal{P}$, we can grow an SP-tree in $\mathcal{P}$ from r using $O(d)$ rounds.*

We now extend this solution to arbitrary simply-connected particle systems using the shortest-path map $\mathrm{SPM}(r)$. The SP-tree constructed by the algorithm above contains exactly the particles of the visibility region R_0 of $\mathrm{SPM}(r)$. As any shortest path from a window particle to r passes through the root of that window, any window incident to R_0 forms a single branch of the SP-tree. To continue the growth of the tree in the remainder of $\mathcal{P}$, we extend the cone of valid directions for the root particles of the regions of $\mathrm{SPM}(r)$ by $120°$. A particle p can detect whether it is a root of an $\mathrm{SPM}(r)$-region by checking its local neighborhood. Specifically, let the parent of p lie in the direction of the port $i+3$. If (1) the cone assigned to p by its parent is $[i-1, i]$ (or $[i, i+1]$), (2) the neighboring node of p in the direction $i+2$ (or $i-2$) is empty, and (3) the node in the direction $i+1$ (or $i-1$) is not empty, then p is the root of an $\mathrm{SPM}(r)$-region, and thus p extends its cone to $[i-1, i+2]$ (or $[i-2, i+1]$) (see Figure 4 (middle)). Note that an extended cone becomes a $180°$-cone.

▶ **Lemma 5.** *Let $\mathrm{SPM}(r)$ be the shortest-path map of a particle r in a simply-connected particle system $\mathcal{P}$. A particle $u \in \mathcal{P}$ extends its cone during the construction of an SP-tree if and only if it is the root of a region in $\mathrm{SPM}(r)$.*

Lemmas 4 and 5 together imply Theorem 6.

▶ **Theorem 6.** *Given a simply-connected particle configuration $\mathcal{P}$ with diameter d and a particle $r \in \mathcal{P}$ we can grow an SP-tree in $\mathcal{P}$ from r using $O(d)$ rounds.*

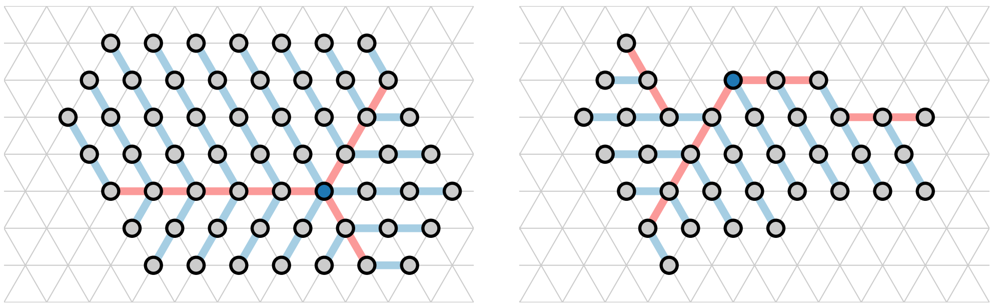


Figure 5 Two feather trees growing from the dark blue root. Shafts are red and branches are blue. Left: every particle is reachable by the initial feathers; Right: additional feathers are necessary.

2.2 Feather trees

These SP-trees, although efficient in construction, are not unique: the exact shape of the tree depends on the activation sequence of its particles. Our approach to the reconfiguration problem is to construct multiple overlapping trees which the particles use to navigate across the structure. As the memory capacity of the particles is restricted, they cannot distinguish between multiple SP-trees by using ids. Thus we need SP-trees that are unique and have a more restricted shape, so that the particles can distinguish between them by using their geometric properties. In this section we hence introduce *feather trees* which are a special case of SP-trees that use narrower cones during the growth process. As a result, feather trees bifurcate less and have straighter branches.

Feather trees follow the same construction rules as efficient SP-trees, but with a slightly different specification of cones. We distinguish between particles on *shafts* (emanating from the root or other specific nodes) and *branches* (see Figure 5 (left)). The root r chooses a maximal independent set of neighbors N_{ind}; it contains at most three particles and there are at most two ways to choose. The particles in N_{ind} receive a standard cone with three directions (a 3-cone), and form the bases of shafts emanating from r. All other neighbors of r receive a cone with a single direction (a 1-cone), and form the bases of branches emanating from r. For a neighbor p across the port i, p receives the cone $[i-1, i+1]$, translated to the coordinate system of p, if p is in N_{ind}, and the cone $[i]$ otherwise. The shaft particles propagate the 3-cone straight, and 1-cones into the other two directions, thus starting new branches. Hence all particles (except for, possibly, the root) have either a 3-cone or a 1-cone. The particles with 3-cones lie on shafts and the particles with 1-cones lie on branches.

We extend the construction of the tree around reflex vertices on the boundary of $\mathcal{P}$ in a similar manner as before. If a branch particle p receives a 1-cone from some direction $i+3$, and the direction $i+2$ (or $i-2$) does not contain a particle while the direction $i+1$ (or $i-1$) does, then p initiates a growth of a new shaft in the direction $i+1$ (or $i-1$) by sending there a corresponding 3-cone (see Figure 5 (right)).

Feather trees are a more restricted version of SP-trees (Theorem 6). For every feather tree, there exists an activation order of the particles such that the SP-tree algorithm would create this specific feather tree. This leads us to the following lemma.

▶ **Lemma 7.** *Given a simply connected particle configuration $\mathcal{P}$ with diameter d and a particle $r \in \mathcal{P}$, we can grow a feather tree from r in $O(d)$ rounds.*

Every particle is reached by a feather tree exactly once, from one particular direction. Hence a feather tree is independent of the activation sequence of the particles. In the following we describe how to navigate a set of overlapping feather trees. To do so, we first identify a useful property of shortest paths in feather trees.

We say that a vertex v of $G_{\mathcal{P}}$ is an *inner vertex*, if v and its six neighbors lie in the core $I \cap T$. All other vertices of the core are *boundary vertices*. A *bend* in a path is formed by three consecutive vertices that form a 120° angle. We say that a bend is an *boundary bend* if all three of its vertices are boundary vertices; otherwise the bend is an *inner bend*.

▶ **Definition 8** (Feather Path). *A path in $G_{\mathcal{P}}$ is a* feather path *if it does not contain two consecutive inner bends.*

We argue that every path π from the root to a leaf in a feather tree is a feather path. This follows from the fact that inner bends can occur only on shafts, and π must alternate visiting shafts and branches.

▶ **Lemma 9.** *A path between a particle s and a particle t is a feather path if and only if it lies on a feather tree rooted at s.*

Navigating feather trees. Consider a directed graph composed of multiple overlapping feather trees with edges pointing from roots to leaves. Due to its limited memory, a particle cannot store the identity of the tree it is currently traversing. Despite that, particles can navigate down the graph towards the leaf of some feather tree and remain on the correct feather tree simply by counting the number of inner bends and making sure that the particle stays on a feather path (Lemma 9). The number of inner bends can either be zero or one, and thus storing this information does not violate the assumption of constant memory per particle. Thus, when starting at the root of a feather tree, a particle p always reaches a leaf of that same tree. In particular, it is always a valid choice for p to continue straight ahead (if feasible). A left or right 120° turn is a valid choice if it is a boundary bend, or if the last bend p made was boundary.

When moving against the direction of the edges, up the graph towards the root of some tree, we cannot control which root of which feather tree a particle p reaches, but it still does so along a shortest path. In particular, if p's last turn was on an inner bend, then its only valid choice is to continue straight ahead. Otherwise, all three options (straight ahead or a 120° left or right turn) are valid.

3 Supply and demand

Each supply root organizes its supply component into an SP-tree; the supply particles will navigate through the supply roots into the core $I \cap T$ and towards the demand components along a *supply graph*. The supply graph, constructed in $I \cap T$, serves as a navigation network for the particles moving from the supply to the demand along shortest paths. Let $G_{I\cap T}$ be the subgraph of G induced on the nodes of $I \cap T$. We say a supply graph S is a subgraph of $G_{I\cap T}$ connecting every supply root s to every demand root d such that the following three *supply graph properties* hold:

1. for every pair (d, s) a shortest path from d to s in S is also a shortest path in $G_{I\cap T}$,
2. for every pair (d, s) there exists a shortest path from d to s in S that is a feather path,
3. every particle p in S lies on a shortest path for some pair (d, s).

We orient the edges of S from demand to supply, possibly creating parallel edges oriented in opposite directions. For a directed edge from u to v in S, we say that u is the predecessor of v, and v is the successor of u.

To create the supply graph satisfying the above properties, we use feather trees rooted at demand roots. Every demand root initiates the growth of its feather tree. When a feather tree reaches a supply root, a *supply found token* is sent back to the root of the tree. Note that if several feather trees overlap, a particle p in charge of forwarding the token up the tree cannot always determine which specific direction the corresponding root of the tree is. It thus sends a copy of the token to all *valid* parents (predecessors of p on every possible feather path of the token), and so the token eventually reaches all demand roots reachable by a feather path from the node it was created in. To detect if a token has already made an inner bend, the supply found token carries a flag β that is set once the token makes an inner bend. Specifically, a particle p that receives a supply found token t does the following:

1. p marks itself as part of the supply graph S, and adds the direction i that t came from as a valid successor in S,
2. from the set of all its predecessors (for all incoming edges), p computes the set U of valid predecessors. When p is not a reflex vertex, $U = [i + 3]$ if $\beta = 1$, and $U = [i + 2, i + 4]$ if $\beta = 0$. For reflex vertices, β is reset to 0 and more directions might be valid. Particle p then adds U to the set of its valid predecessors in S,
3. p sends copies of t (with an updated value of β) to the particles in U, and
4. p stores t in its own memory.

Each particle p, from each direction i, stores at most one token with flag β set to 1 and one with β set to 0. Hence p can store the corresponding information in its memory. If a particle p already belongs to the supply graph S when it is reached by a feather tree F, then p checks if its predecessor q in F is a valid parent for any token t stored in p, and, if so, adds q to the set of its valid predecessors in S (as in step (2)). For each of these tokens, but for at most one for each values of β, p sends a copy of t to q (as in step (3)).

▶ **Lemma 10.** *Given a simply-connected particle configuration $\mathcal{P}$ with diameter d, a set of particles marked as supply roots, and a set of particles marked as demand roots, a supply graph can be constructed in $O(d)$ rounds.*

Bubbles. Particles move from supply to demand. However, for ease of presentation and analysis, we introduce the abstract concept of demand *bubbles* that move from demand to supply, in the direction of edges of S, see Figure 6. Let us assume for now that the supply graph S has been constructed (in fact, its construction can occur in parallel with the reconfiguration process described below). Starting with a corresponding demand root d, each demand component D is constructed by particles flowing from the core $I \cap T$, according to the spanning tree of D stored in d. Every time a leaf particle expands in D, it creates a *bubble* of demand that needs to travel through d down S to the supply, where it can be

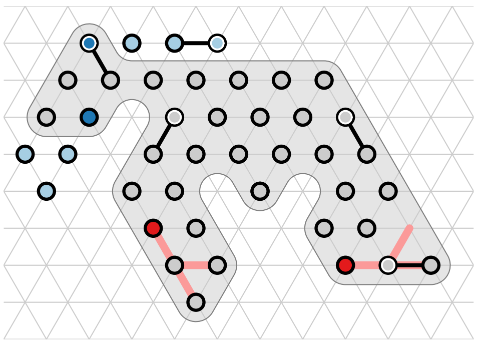


Figure 6 A reconfiguration process with five expanded particles holding bubbles (outlined in white). Supply particles are blue, supply roots dark blue. Demand roots (red) store spanning trees of their demand components. The supply graph is not shown.

resolved. Bubbles move via a series of handovers along shortest paths in S. An expanded particle p holding a bubble b stores two values associated with it. The first value β is the number of inner bends b took since the last boundary bend ($\beta \in \{0, 1\}$), and is used to route the bubbles in S. The second value δ stores the general direction of b's movement; $\delta = \textsc{s}$ if b is moving forward to supply, and $\delta = \textsc{d}$ if b is moving backwards to demand.

If a particle p holding a bubble b wants to move b to a neighboring particle q, p can only do so if q is contracted. Then, p initiates a pull operation, and thereby transfers b and its corresponding values to q. Thus the particles are pulled in the direction of a demand root, but the bubbles travel along S from a demand root towards the supply.

A supply component may become empty before all bubbles moving towards it are resolved. In this case, the particles of S have to move the bubbles back up the graph. Particles do not have sufficient memory to store which specific demand root bubbles came from. However, because of the first supply graph property, every demand root has a connection to the remaining supply. While a bubble is moved back up along S, as soon as there is a different path towards some other supply root, it is moved into that path. Then, the edges connecting to the now empty supply are deleted from S. Moreover, other edges that now point to empty supply or to deleted edges are themselves deleted from S. As the initial and target shapes have the same size, the total number of bubbles equals the number of supply particles. Therefore, once all bubbles are resolved, the reconfiguration problem is solved.

In the remainder of the paper we may say "a bubble activates" or "a bubble moves". By this we imply that "a particle holding a bubble activates" or "a particle holding a bubble moves the bubble to a neighboring particle by activating a pull handover".

4 Navigating the supply graph

When the demand and supply roots are connected with the supply graph S, as described in Section 3, the reconfiguration process begins. Once a demand root d has received a supply found token from at least one successor (d is added to S), it begins to construct its demand component D along the spanning tree $\mathcal{T}_D$ that d stores, and starts sending demand bubbles into S. The leafs of the partially built spanning tree $\mathcal{T}_D$ carry the information about their respective sub-trees yet to be built, and pull the chains of particles from d to fill in those sub-trees, thus generating bubbles that travel through d into S.

With each node v of S, for every combination of the direction i to a predecessor of v and a value of β, we associate a value $\lambda(i, \beta) \in \{\text{true}, \text{false}\}$, which encodes the liveliness of feather paths with the corresponding value of β from the direction i through v to some supply. If $\lambda(i, \beta) = \text{false}$ then, for a given value of β, there are no feather paths through v to non-empty supply in S. Note that here we specifically consider nodes of S, and not the particles occupying them. When particles travel through S, they maintain the values of λ associated with the corresponding nodes. Initially, when S is being constructed, $\lambda = \text{true}$ for all nodes, all directions, and all β. When a bubble b travels down S to a supply component that turns out to be empty, b reverses its direction. Then, for all the nodes that b visits while reversing, the corresponding value λ is set to false, thus marking the path as dead.

For an expanded particle p occupying two adjacent nodes of S, denote the predecessor node as v_a, and the successor node as v_b. By our convention, we say the bubble in p occupies v_b. When particle p with a bubble b activates, it performs one of the following operations. It checks them in order and performs the first action available.

1. If $\delta = \text{S}$ (b is moving to supply) and v_b is inside a supply component, then p pulls on any contracted child in the spanning tree of the supply; if v_b is a leaf, then p simply contracts into v_a, thus resolving the bubble.
2. If $\delta = \text{S}$ and $v_b \in S$, p checks which of the successors of v_b in S lie on a feather path for b and are alive (i.e., their corresponding $\lambda = \text{true}$). If there is such a successor q that is contracted, p pulls q and sends it the corresponding values of β and δ (while updating β if needed), i.e., p transfers b to q. Thus the bubble moves down S to supply.
3. If $\delta = \text{S}$, $v_b \in S$, and v_b does not have alive successors in S that lie on a feather path for b, then p reverses the direction of b to $\delta = \text{D}$, and sets the value $\lambda(i, \beta)$ of v_b to false, where i is the direction from v_b to v_a. The bubble does not move.
4. If $\delta = \text{D}$ (b is moving to demand) and there exists an alive successor node of either v_b or v_a that lies on a feather path for b and is occupied by a contracted particle q, then p switches the direction of b to $\delta = \text{S}$, pulls on q, and transfers to q the bubble b with its corresponding values (while updating β if needed). The bubble changes direction and moves onto a feather path which is alive.
5. If $\delta = \text{D}$ and none of the successors of v_b or v_a in S are alive for b, then p sets the corresponding values λ of v_b and v_a to false, and checks which predecessors of v_a lie on a feather path for b (note, this set is non-empty). If there is such a predecessor q that is contracted, then p pulls q and transfers it the bubble b. Thus the bubble moves up in S.

Coarse grid. Particles can only make progress if they have a contracted successor. To ensure that flows of particles along different feather paths can cross without interference, we introduce a coarsened grid, and devise a special crossing procedure.

Rather than constructing supply graph S on the triangular grid G, we now do so using a grid G_L that is coarsened by a factor of three and is laid over the core $I \cap T$ (see Figure 2). Then, among the nodes of G we distinguish between those that are also *grid nodes* of G_L, *edge nodes* of G_L, and those that are neither. By our assumptions on the input, the graph G_L is connected. Note, that then, every node of the particle system is either a part of G_L (is a grid or an edge node), or is adjacent to a node of G_L. To ensure that all particles agree on the location of G_L, we assume that we are given a leader particle ℓ in the core $I \cap T$, that initiates the construction of G_L.

In between two grid nodes of G_L, there are exactly two edge nodes of G_L. While bubbles are in the core $I \cap T$, we let them occupy only edge nodes of G_L. To resolve crossing paths in S, every time a bubble wants to cross a grid node of G_L occupied by a particle p, the bubble sends a request to p. Then if p receives multiple of these requests, it decides which of the bubbles can cross. To avoid potential head-on collisions of bubbles on a single edge of G_L corresponding to a bidirectional edge of S, our algorithm temporarily disallows the movement of bubbles in one of the two directions. Only after the chosen direction is marked as dead, the other direction becomes available for bubble traversal. That is, at any moment in time, the supply graph, does not contain bidirectional edges. See Appendix B for details.

5 Algorithm

To summarize, our approach consists of three phases. In the first phase, the leader particle initiates the construction of the coarse grid G_L over the core $I \cap T$. In the second phase, the particles grow feather trees, starting from the demand roots. If a feather tree reaches supply, that information is sent back up the tree and the particles form the supply graph S. In the last phase, particles move from supply to demand along S. Note that, for the particle system as a whole, these phases may overlap in time. For example, the reconfiguration process may begin before the supply graph is fully constructed. Each individual particle can move on to executing the next phase of the algorithm once the previous phase for it is finished.

For the purposes of analysis, we view the reconfiguration as bubbles of demand traveling along S from demand to supply. Bubbles turn around on dead paths where all supply has been consumed. To ensure proper crossing of different bubble flows, we let the grid nodes of S to act as traffic conductors, letting some bubbles cross while others wait for their turn.

Correctness. For simplicity of presentation, we first show correctness of the algorithm under a sequential scheduler. We then extend the algorithm and its analysis to the case of an asynchronous scheduler. To show correctness, we need to show two properties, *safety* and *liveness*. The algorithm is safe if $\mathcal{P}$ never enters an invalid state, and is live if in any valid state there exists a particle that, when activated, can make progress towards the goal. Lemma 10 proves the correctness of the phase of the construction of the supply graph. As the particles start executing the reconfiguration phase only after the construction of the supply graph is finished for them locally, for the purposes of proving the correctness of the reconfiguration phase, we may assume that the supply graph has been constructed in the particle system as a whole. A state of $\mathcal{P}$ is valid when it satisfies the following properties:

- Particle configuration $\mathcal{P}$ is connected.
- It holds that $\#b + \#d = \#s$, where $\#b$, $\#d$, and $\#s$ are the number of bubbles, demand spots, and supply particles respectively. That is, the size of supply matches exactly the size of demand. This assumes that initial and target shapes have the same size.
- There are no bidirectional edges in S allowed for traversal.
- For every pair of demand root d and supply root s with a non-empty supply, there exists a feather path from d to s in S; furthermore, all such feather paths are alive (i.e., the corresponding values of λ are set to true).
- Any expanded particle has both nodes on a single feather path of S.
- Any expanded particle on an alive feather path moves to supply ($\delta = \mathrm{s}$).
- Any node of S with $\lambda(i, \beta) = $ true, for some combination of direction i and value β, is connected by an alive feather path to some demand root d.

For the property of liveness, we need to show that progress can always be made. We say the particle system makes progress whenever (1) a bubble moves on its path (this includes resolving the bubble with supply, or when a demand root creates a new bubble), and (2) a bubble changes its value for direction δ. In the full version of our paper [22], we show by induction that during the execution of our algorithm, the configuration stays valid at any point in time, and that at any point in time, there is a bubble that can make progress.

▶ **Lemma 11.** *A particle configuration $\mathcal{P}$ stays valid and live at all times.*

Running time analysis. We begin by arguing that the total distance traveled by each bubble is linear in the number of particles in the system. We first limit the length of the path each bubble takes.

▶ **Lemma 12.** *The total path of a bubble b in a particle configuration $\mathcal{P}$ with n particles has size $O(n)$.*

Next, we give a lower bound on the number of rounds it takes each bubble to traverse its path. We analyze the progress made by the particle system by creating a specific series of $O(n)$ configurations that represent a lower bound on the overall progress; we show that any activation sequence chosen by an adversarial scheduler results in at least as much progress. This approach has been previously used for analyzing the running time of a moving line of particles [14], and for a tree moving from the root [7]. Together with the proof of correctness, we conclude with the following theorem:

▶ **Theorem 13.** *The particle reconfiguration problem for particle configuration $\mathcal{P}$ with n particles can be solved using $O(n)$ rounds of activation under a sequential scheduler.*

Extending the analysis to an asynchronous scheduler. Unlike in the case of a sequential scheduler, when at each moment only one particle can be active, under an asynchronous scheduler multiple particles may be active at the same time. This may lead to concurrency issues. Specifically, when actions of two or more particles conflict with each other, not all of them can be finished successfully. To extend our algorithm to the case of an asynchronous scheduler, we explore what kind of conflicts may arise, and ensure that our algorithm can deal with them. There can be three types of conflicts:

1. Two particles try to expand into the same empty node.
2. Two contracted particles try to push the same expanded particle.
3. Two expanded particles try to pull on the same contracted particle.

The first two kinds of conflicts never arise in our algorithm, as (1) every empty node (demand spot) can be moved into from only a single direction, according to the spanning tree stored by the corresponding demand root, and (2) no particle ever performs a push operation. The third conflict could potentially arise in our approach when two bubbles traveling on crossing paths pass the junction node. However, as crossing of a junction is controlled by the junction particle, only one particle at a time is given permission to pull on the junction node. Thus, under an asynchronous scheduler, all actions initiated by the particles always succeed.

We are left to analyze whether the asynchronous setting can lead to deadlocks in our algorithm, where concurrency issues result in some particle not being able to make progress. As mentioned above, crossings of junctions are controlled by the junction particles, and thus cannot lead to deadlocks. Furthermore, the algorithm forbids the simultaneous existence of bidirectional edges in S, thus there cannot be deadlocked bubbles moving in the opposite directions over the same edge. The only remaining case of a potential deadlock is when the algorithm temporarily blocks one of the two directions of a bidirectional edge. In a sequential schedule, this choice is made by one of the two edge nodes of G_L corresponding to the bidirectional edge of S. However, in an asynchronous schedule, both edge nodes may become active simultaneously, and choose the opposite directions of S. To resolve this case, and to break the symmetry of two nodes activating simultaneously and choosing the opposite directions, we need to utilize the power of randomness. We assume that the particles have access to a constant number of random bits. Daymude et al. [9] show that, in that case, a mechanism exists that allows the particles to lock their local neighborhood, and to perform their actions as if the neighboring particles were inactive. Such a locking mechanism increases the running time only by a constant factor in expectation, and by a logarithmic factor if the particle needs to succeed with high probability. Thus, with a locking mechanism, we can ensure that our algorithm can select one of the two directions of a bidirectional edge in S.

▶ **Theorem 14.** *The particle reconfiguration problem for particle configuration $\mathcal{P}$ with n particles can be solved in $O(n)$ rounds of activation in expectation (or in $O(n \log n)$ rounds of activation with high probability) under an asynchronous scheduler, if each particle has access to a constant number of random bits.*

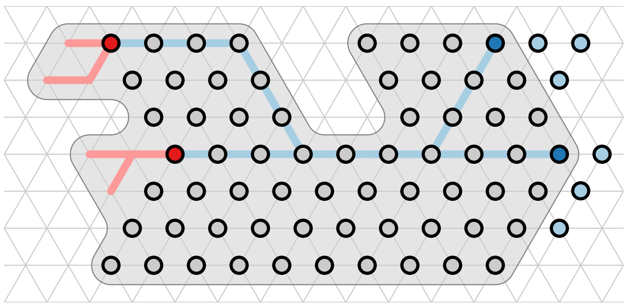


Figure 7 Shortest paths lead through a bottleneck, slowing down the algorithm in practice.

The running time of our algorithm is worst-case optimal: any algorithm to solve the particle reconfiguration problem needs at least a linear number of rounds.

▶ **Theorem 15.** *Any algorithm that successfully solves the particle reconfiguration problem needs $\Omega(n)$ rounds.*

6 Discussion

We presented the first reconfiguration algorithm for programmable matter that does not use a canonical intermediate configuration. In the worst-case, our algorithm requires a linear number of activation rounds and hence is as fast as existing algorithms. However, in practice, our algorithm can exploit the geometry of the symmetric difference between input and output and can create as many parallel shortest paths as the problem instance allows.

We implemented our algorithm in the Amoebot Simulator[1]. In the following screenshots and the accompanying videos of complete reconfiguration sequences[2] supply particles are colored green, demand roots red, and supply roots cyan. The dark blue particles are part of the supply graph and therefore lie on a feather path from a demand root to a supply root.

Figure 8 illustrates that our algorithm does indeed create a supply graph which is based on the shortest paths between supply and demand and hence facilitates parallel movement paths if the geometry allows. Activation sequences are randomized, so it is challenging to prove statements that capture which supply feeds which demand, but generally we observe that close supply and demand nodes will connect first. An interesting open question in this context is illustrated in Figure 7: here we see two supplies and two demands, but the shortest paths have a common bottleneck, which slows down the reconfiguration in practice. Is there an effective way to include near-shortest paths in our supply graph to maximize parallelism? One could also consider temporarily adding particles to the symmetric difference. Note, though, that additional parallelism leads only to constant factor improvements in the running time; this is of course still meaningful in practice.

As we already hinted in the introduction, the running time of our algorithm is linked to the balance of supply in the feather trees. Recall that the trees are rooted at demand and grow towards supply; bubbles move from demand to supply. Because bubbles decide randomly at each junction which path to follow, each sub-tree will in expectation receive a similar amount of bubbles. If all sub-trees of a junction contain an equal amount of supply particles, then we say this junction is *balanced*. If all junctions are balanced, then we say a feather tree is balanced; similarly we speak about a balanced supply graph.

[1] `https://github.com/SOPSLab/AmoebotSim`
[2] `https://github.com/PetersTom/AmoebotVideos`

The worst case running time of $O(n)$ rounds is triggered by unbalanced supply graphs. See, for example, Figure 9 (right), where at each junction the sub-tree rooted in the center carries all remaining supply, except for two particles. Such situations arise when there are many small patches of supply and only a few locations with demand. In the realistic scenario of shape repair we have exactly the opposite situation with many small damages (demand) and one large reservoir of supply stored for repairs. Here the supply graph will naturally be balanced and the running time is proportional to the diameter of the core $I \cup T$. Figure 9 (left) shows an example with only one supply and demand; here the running time of our algorithm is even proportional to the distance between supply and demand within $I \cup T$.

Feather trees are created to facilitate particles traveling on (crossing) shortest paths between supply and demand; they do not take the balance of the supply graph into account. A challenging open question in this context is whether it is possible to create supply graphs which retain the navigation properties afforded by feather tress but are at the same time balanced with respect to the supply.

As a final example, Figure 10 gives an impression of the natural appearance of the reconfiguration sequences produced by our algorithm. In the remainder of this section we discuss ways to lift A1: the core $I \cap T$ is non-empty and simply connected. If the core consist of more than one component, then we can choose one component as the core for the reconstruction. The other components can then be interpreted by the algorithm as both supply and demand. Consequently they will first be deconstructed and then reassembled. Clearly this is not an ideal solution, but it does not affect the asymptotic running time and does not require any major changes to our algorithm. If the core is not simply connected, that is, it contains holes, then our procedure for creating feather trees may no longer create shortest path trees or it might not even terminate. The slow $O(n^2)$ algorithm will however terminate and produce shortest path. A very interesting direction for future work are hence efficient algorithms for shortest path trees for shapes with holes.

References

1 Marta Andrés Arroyo, Sarah Cannon, Joshua J. Daymude, Dana Randall, and Andréa W. Richa. A stochastic approach to shortcut bridging in programmable matter. *Natural Computing*, 17(4):723–741, 2018. `doi:10.1007/s11047-018-9714-x`.

2 Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM (JACM)*, 32(4):804–823, 1985.

3 Sarah Cannon, Joshua J. Daymude, Cem Gökmen, Dana Randall, and Andréa W. Richa. A Local Stochastic Algorithm for Separation in Heterogeneous Self-Organizing Particle Systems. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2019)*, volume 145 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 54:1–54:22, 2019. `doi:10.4230/LIPIcs.APPROX-RANDOM.2019.54`.

4 Sarah Cannon, Joshua J. Daymude, Dana Randall, and Andréa W. Richa. A Markov Chain Algorithm for Compression in Self-Organizing Particle Systems. In *Proc. ACM Symposium on Principles of Distributed Computing (PODC)*, pages 279–288, 2016. `doi:10.1145/2933057.2933107`.

5 Kenneth C. Cheung, Erik D. Demaine, Jonathan R. Bachrach, and Saul Griffith. Programmable Assembly With Universally Foldable Strings (Moteins). *IEEE Transactions on Robotics*, 27(4):718–729, 2011. `doi:10.1109/TRO.2011.2132951`.

6 Joseph C. Culberson and Robert A. Reckhow. Dent Diagrams: A Unified Approach to Polygon Covering Problems. Technical report, University of Alberta, 1987. TR 87–14.

7 Joshua J. Daymude, Zahra Derakhshandeh, Robert Gmyr, Alexandra Porter, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. On the Runtime of Universal Coating for Programmable Matter. *Natural Computing*, 17(1):81–96, 2016. doi:10.1007/s11047-017-9658-6.

8 Joshua J. Daymude, Robert Gmyr, Kristian Hinnenthal, Irina Kostitsyna, Christian Scheideler, and Andréa W. Richa. Convex Hull Formation for Programmable Matter. In *Proc. 21st International Conference on Distributed Computing and Networking*, pages 1–10, 2020. doi:10.1145/3369740.3372916.

9 Joshua J. Daymude, Andréa W. Richa, and Christian Scheideler. Local Mutual Exclusion for Dynamic, Anonymous, Bounded Memory Message Passing Systems, November 2021. URL: http://arxiv.org/abs/2111.09449.

10 Joshua J. Daymude, Andréa W. Richa, and Christian Scheideler. The Canonical Amoebot Model: Algorithms and Concurrency Control. In *35th International Symposium on Distributed Computing (DISC)*, volume 209 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 20:1–20:19, 2021. doi:10.4230/LIPIcs.DISC.2021.20.

11 Hooman R. Dehkordi, Fabrizio Frati, and Joachim Gudmundsson. Increasing-Chord Graphs On Point Sets. In *Proc. International Symposium on Graph Drawing (GD)*, LNCS 8871, pages 464–475, 2014. doi:10.1007/978-3-662-45803-7_39.

12 Erik D. Demaine, Jacob Hendricks, Meagan Olsen, Matthew J. Patitz, Trent A. Rogers, Nicolas Schabanel, Shinnosuke Seki, and Hadley Thomas. Know When to Fold 'Em: Self-assembly of Shapes by Folding in Oritatami. In *DNA Computing and Molecular Programming*, pages 19–36, 2018. doi:10.1007/978-3-030-00030-1_2.

13 Zahra Derakhshandeh, Shlomi Dolev, Robert Gmyr, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. Brief announcement: Amoebot—A New Model for Programmable Matter. In *Proc. 26th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 220–222, 2014. doi:10.1145/2612669.2612712.

14 Zahra Derakhshandeh, Robert Gmyr, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. Universal Shape Formation for Programmable Matter. In *Proc. 28th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 289–299, 2016. doi:10.1145/2935764.2935784.

15 Zahra Derakhshandeh, Robert Gmyr, Thim Strothmann, Rida Bazzi, Andréa W. Richa, and Christian Scheideler. Leader Election and Shape Formation with Self-organizing Programmable Matter. In *Proc. International Workshop on DNA-Based Computing (DNA)*, LNCS 9211, pages 117–132, 2015. doi:10.1007/978-3-319-21999-8_8.

16 Giuseppe A. Di Luna, Paola Flocchini, Nicola Santoro, Giovanni Viglietta, and Yukiko Yamauchi. Shape formation by programmable particles. *Distributed Computing*, 33:69–101, 2020. doi:10.1007/s00446-019-00350-6.

17 Fabien Dufoulon, Shay Kutten, and William K. Moses Jr. Efficient Deterministic Leader Election for Programmable Matter. In *Proc. 2021 ACM Symposium on Principles of Distributed Computing*, pages 103–113, 2021. doi:10.1145/3465084.3467900.

18 Cody Geary, Paul W. K. Rothemund, and Ebbe S. Andersen. A single-stranded architecture for cotranscriptional folding of RNA nanostructures. *Science*, 345(6198):799–804, 2014. doi:10.1126/science.1253920.

19 Subir Kumar Ghosh. *Visibility Algorithms in the Plane*. Cambridge University Press, 2007. doi:10.1017/CBO9780511543340.

20 Robert Gmyr, Kristian Hinnenthal, Irina Kostitsyna, Fabian Kuhn, Dorian Rudolph, Christian Scheideler, and Thim Strothmann. Forming Tile Shapes with Simple Robots. In *Proc. International Conference on DNA Computing and Molecular Programming (DNA)*, pages 122–138, 2018. doi:10.1007/978-3-030-00030-1_8.

21 Irina Kostitsyna, Tom Peters, and Bettina Speckmann. Brief announcement: An effective geometric communication structure for programmable matter. In *36th International Symposium on Distributed Computing (DISC 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

22 Irina Kostitsyna, Tom Peters, and Bettina Speckmann. Fast reconfiguration for programmable matter. *arxiv*, August 2023. URL: `https://arxiv.org/abs/2202.11663`.

23 Joseph S. B. Mitchell. A new algorithm for shortest paths among obstacles in the plane. *Annals of Mathematics and Artificial Intelligence*, 3(1):83–105, 1991. `doi:10.1007/BF01530888`.

24 Andre Naz, Benoit Piranda, Julien Bourgeois, and Seth Copen Goldstein. A distributed self-reconfiguration algorithm for cylindrical lattice-based modular robots. In *Proc. 2016 IEEE 15th International Symposium on Network Computing and Applications (NCA)*, pages 254–263, 2016. `doi:10.1109/NCA.2016.7778628`.

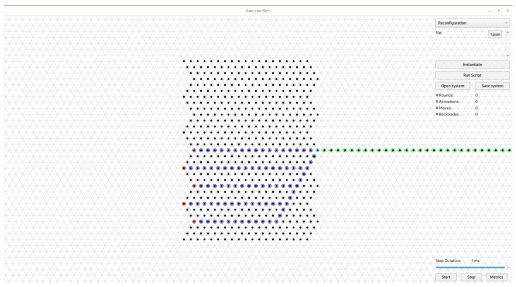
25 Matthew J. Patitz. An introduction to tile-based self-assembly and a survey of recent results. *Natural Computing*, 13(2):195–224, 2014. `doi:10.1007/s11047-013-9379-4`.

26 Benoit Piranda and Julien Bourgeois. Designing a quasi-spherical module for a huge modular robot to create programmable matter. *Autonomous Robots*, 42(8):1619–1633, 2018. `doi:10.1007/s10514-018-9710-0`.

27 Alexandra Porter and Andrea Richa. Collaborative Computation in Self-organizing Particle Systems. In *Proc. International Conference on Unconventional Computation and Natural Computation (UCNC)*, LNCS 10867, pages 188–203, 2018. `doi:10.1007/978-3-319-92435-9_14`.

28 Damien Woods, Ho-Lin Chen, Scott Goodfriend, Nadine Dabby, Erik Winfree, and Peng Yin. Active self-assembly of algorithmic shapes and patterns in polylogarithmic time. In *Proc. 4th Conference on Innovations in Theoretical Computer Science (ITCS)*, pages 353–354, 2013. `doi:10.1145/2422436.2422476`.

A Simulation figures



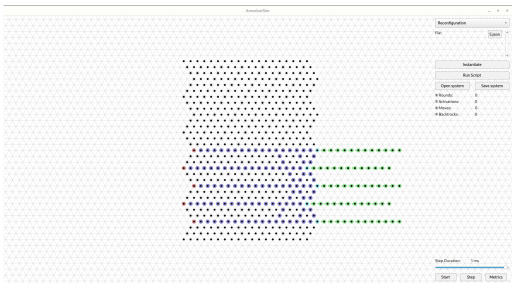
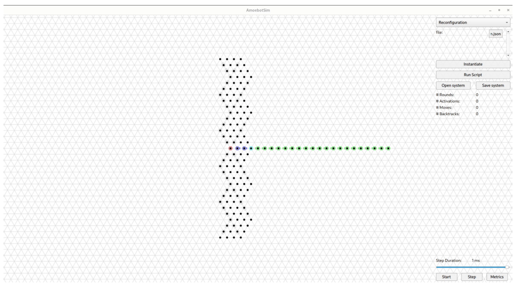


Figure 8 One supply and five demands vs. five supplies and five demands.



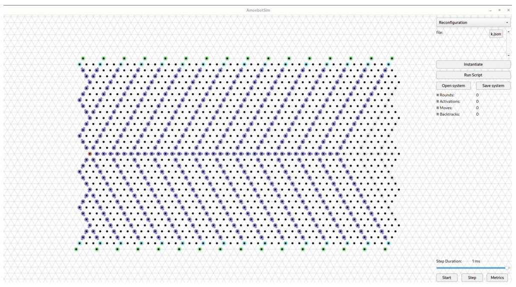


Figure 9 Left: Particles move directly between supply and demand. Right: Worst case configuration for back-tracking.

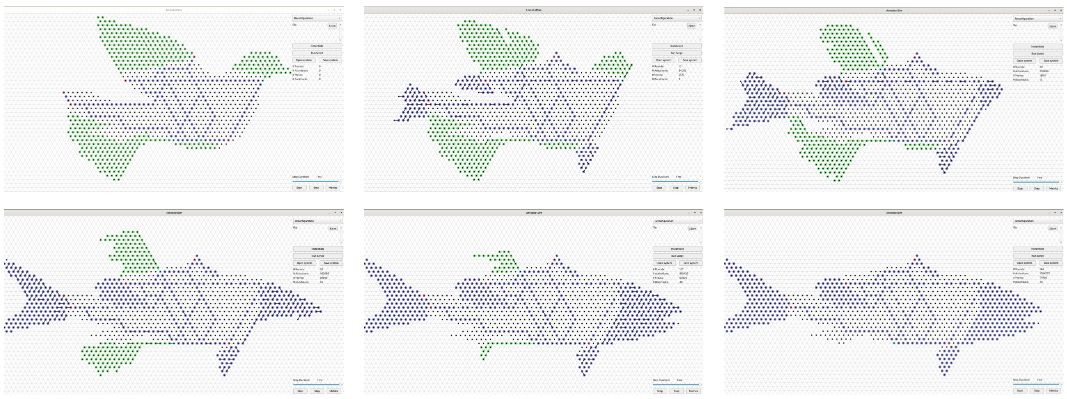


Figure 10 Dove to fish, full video at `https://github.com/PetersTom/AmoebotVideos`.

B Coarse grid

Particles can only make progress if they have a contracted successor. If there are crossing paths in S, these paths might interfere. To ensure that flows of particles along different feather paths can cross, we introduce a coarsened grid, and devise a special crossing procedure.

Rather than constructing supply graph S on the triangular grid G, we now do so using a grid G_L that is coarsened by a factor of three and is overlaid over the core $I \cap T$ (see Figure 2). Then, among the nodes of G we distinguish between those that are also *grid nodes* of G_L, *edge nodes* of G_L, and those that are neither. By our assumptions on the input, the graph G_L is connected. Note, that then, every node of the particle system is either a part of G_L (is a grid or an edge node), or is adjacent to a node of G_L. To ensure that all particles agree on the location of G_L, we assume that we are given a leader particle ℓ in the core $I \cap T$, that initiates the construction of G_L (see Figure 2 in Section 1).

Note that for two adjacent grid nodes v_1 and v_2 of G_L, the edge (v_1, v_2) does not have to be in G_L, if one or both corresponding edge nodes are not part of the core $I \cap T$. We say a grid node v in G_L is a boundary node if there exists a grid cell with corners v, v_1, v_2 where at least one of its edges is missing. Grid node v is a *direct boundary* node if this missing edge is incident to v, and an *indirect boundary* node if the missing edge is (v_1, v_2). All other grid nodes in G_L are inner nodes.

Now, feather trees only grow over the particles in G_L. Demand roots initiate the construction of their feather trees from a closest grid node in G_L. The supply roots organize their respective supply components in SP-trees as before (in the original grid), and connect to all adjacent particles in the supply graph S ($S \subseteq G_L$).

The growth of a feather tree in G_L is very similar to that in G. On direct boundary nodes, the cones propagate according to the same rules as before. On indirect boundary nodes, the creation of new shafts and branches is outlined in Figure 11. The resulting feather tree may now have angles of 60°. To still be able to navigate the supply graph, bubbles are allowed to make a 60° bend on indirect boundary nodes, resetting $\beta = 0$, only when both the start and end vertex of that bend are also boundary nodes.

A useful property of G_L that helps us ensure smooth crossings of the bubble flows is that there are at least two edge nodes in between any two grid nodes. We can now restrict the expanded particles carrying bubbles to mainly use the edge nodes of S, and only cross the grid nodes if there is enough room for them to fully cross.

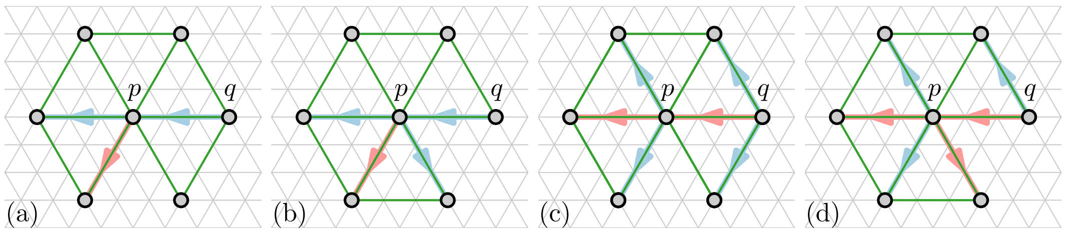


Figure 11 Construction of feather trees at indirect boundary nodes. G_L is shown in green, only the particles on nodes of G_L are shown. A branch ((a)–(b), in blue) and a shaft ((c)–(d), in red) of a feather tree grows from q to p. (a) A new shaft is emanated from p; (b) a new shaft and a new branch are emanated from p; (c) p behaves as an internal node; (d) a new shaft is emanated from p.

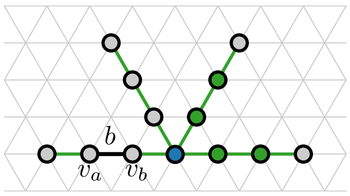


Figure 12 A bubble b on two edge nodes of G_L. Only particles on G_L are drawn. The blue particle is on $c(b)$, the nodes occupied by the two pairs of green particles together form $N(b)$.

With $p(x)$, we denote the particle occupying node x. For a directed edge from u to v in S, we say that u is the predecessor of v, and v is the successor of u, and denote this by $u \to v$. Consider a bubble b moving forward in S (with $\delta = \text{s}$), held by an expanded particle p occupying two edge nodes v_a and v_b, with $v_a \to v_b$ (see Figure 12). Let $c(b)$ be the grid node of S adjacent to v_b, such that $v_b \to c(b)$; $c(b)$ is the grid node that b needs to cross next. Let $\mathcal{D}(b)$ denote the set of valid directions for b in S from the position of $c(b)$. Let $N(b) = \{(r_1^i, r_2^i) \mid i \in \mathcal{D}(b)\}$ be the set of pairs of edge nodes lying in these valid directions for b. Specifically, for each $i \in \mathcal{D}(b)$, let $c(b) \to r_1^i$ in the direction i, and $r_1^i \to r_2^i$.

Crossings in G_L. The grid nodes of G_L that are part of the supply graph act as traffic conductors. We thus use terms grid nodes and junctions interchangeably. For bubble b, its particle p is only allowed to pull on the particle at $c(b)$, and thus initiate the crossing of $c(b)$, if there is a pair of nodes $(r_1, r_2) \in N(b)$ that are occupied by contracted particles. In this case, after at most three activation rounds, b will completely cross $c(b)$, the expanded particle now carrying it will occupy the edge nodes r_1 and r_2, and the junction $c(b)$ will be ready to send another bubble through itself. Assume for now that S has all edges oriented in one direction (there are no parallel edges in opposite directions). Below we discuss how to lift this assumption. The procedure followed by the junctions is the following. If an expanded particle p wants to pull on a particle at a junction, it first requests permission to do so by sending a *request* token containing the direction it wants to go after $c(b)$. Every junction node stores a queue of these requests. A request token arriving from the port i is only added to the queue if there are no requests from i in the queue yet. As every direction is stored only once, this queue is at most of size six. When particle $p(r_1)$ occupying an edge node r_1, with some grid node $c \to r_1$, activates, it checks if itself and the particle $p(r_2)$ at $r_1 \to r_2$ are contracted. If so, $p(r_1)$ sends an *availability* token to $p(c)$. Similarly, junction nodes need to store at most six availability tokens at once.

When the particle at a junction activates, and it is ready to transfer the next bubble, it grants the first pull request with a matching availability token by sending the acknowledgment token to the particle holding the corresponding bubble. The request and availability tokens are then consumed. Only particles with granted pull requests are allowed to pull and move their bubbles onto junctions. Junction queues are associated with the grid nodes of S, and not particles. Thus, if an expanded particle p occupying a grid node c pulls another particle q to c, the queues are transformed into the coordinate system of q and sent to q.

▶ Remark 16. Above, for simplicity of presentation, we assume that there are no bidirectional edges in S. These edges, however, can be treated as follows. During the construction of the supply graph one of the two opposite directions is chosen as a dominating one. For a corresponding edge of G_L, one of its two edge nodes that receives the supply found token first, reserves the direction of the corresponding feather path for this edge. If the other edge node eventually receives the supply found token from the opposite direction, it stores the information about this edge as being inactive. While the branch of the dominating direction is alive, the dominated branch is marked as unavailable. As soon as, and if, supply runs out for the dominating branch, and it is marked dead, the dominated branch is activated, and can now be used. This slows down the reconfiguration process by a number of rounds at most linear in the size of the dominating branch.

Quorum Subsumption for Heterogeneous Quorum Systems

Xiao Li
University of California, Riverside, CA, USA

Eric Chan
University of California, Riverside, CA, USA

Mohsen Lesani
University of California, Riverside, CA, USA

Abstract

Byzantine quorum systems provide higher throughput than proof-of-work and incur modest energy consumption. Further, their modern incarnations incorporate personalized and heterogeneous trust. Thus, they are emerging as an appealing candidate for global financial infrastructure. However, since their quorums are not uniform across processes anymore, the properties that they should maintain to support abstractions such as reliable broadcast and consensus are not well-understood. It has been shown that the two properties quorum intersection and availability are necessary. In this paper, we prove that they are not sufficient. We then define the notion of quorum subsumption, and show that the three conditions together are sufficient: we present reliable broadcast and consensus protocols, and prove their correctness for quorum systems that provide the three properties.

2012 ACM Subject Classification Computing methodologies → Distributed algorithms; Computer systems organization → Availability; Computer systems organization → Reliability

Keywords and phrases Distributed Systems, Impossibility Results, Byzantine fault tolerance

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.28

Acknowledgements We would like to thank DISC '23 reviewers for detailed and constructive reviewers. Further, we would like to specially thank Giuliano Losa for his insightful comments.

1 Introduction

Bitcoin [42] had the promise to democratize the global finance. Globally scattered servers validate and process transactions, and maintain a consistent replication of a ledger. However, the nature of the proof-of-work consensus exhibited disadvantages such as high energy consumption, and low throughput. In contrast, Byzantine replication have always had modest energy consumption. Further, since its advent as PBFT [18], many recent extensions [47, 39, 48, 17, 6, 12, 13] have improved its throughput. However, its basic model of quorums is closed and homogeneous: the set of processes are fixed, and the quorums are assumed to be uniform across processes. Thus, projects such as Ripple [44] and Stellar [38, 33] emerged to bring *heterogeneity* and openness to Byzantine quorum systems. They let every process declare its own set of quorums, or the processes it trusts called slices, from which quorums are calculated.

In this paper, we first consider a basic model of heterogeneous quorum systems where each process has an individual set of quorums. Then, we consider fundamental questions about their properties. Quorum systems are the foundation of common distributed computing abstractions such as reliable broadcast and consensus. We specify the expected safety and liveness properties for these abstractions. *What are the necessary and sufficient properties of heterogeneous quorum systems to support these abstractions?* Previous work [34] noted that quorum intersection and weak availability properties are necessary for the quorum system to implement the consensus abstraction. Quorum intersection requires that every pair of quorums overlap at a well-behaved process. The safety of consensus relies on the quorum intersection property of the underlying quorum system: intuitively, if an operation communicates with a quorum, and a later operation communicates with another quorum, a single well-behaved process in their intersection can make the second quorum aware of the first. A quorum system is weakly available for a process if it has a quorum for that process whose members are all well-behaved. Intuitively, the quorum system is available to that process through that quorum. Since a process needs to communicate with at least one quorum to terminate, the liveness properties are dependent on the availability of the quorum system.

The quorum intersection and availability properties are *necessary.* Are they sufficient as well? In this paper, we prove that they are *not sufficient* conditions to implement reliable broadcast and consensus. For each abstraction, we present execution scenarios, and apply indistinguishability arguments to show that any protocol violates at least one of the safety or liveness properties. What property should be added to make the properties sufficient? A less known property is quorum sharing [34]. Roughly speaking, every quorum should include a quorum for all its members. This is a property that trivially holds for homogeneous quorum systems where every quorum is uniformly a quorum of all its members. However, in general, it does not hold for heterogeneous quorum systems. Previous work showed that it also holds for Stellar quorums if Byzantine processes do not lie about their slices.

Since Byzantine processes' quorums is arbitrary, in practice, quorum sharing is too strong. In order to require inclusion only for the quorums of a well-behaved subset of processes, we consider a weaker notion, called *quorum subsumption.* As we will see, this property lets processes in the included quorum make local decisions while preserving the properties of the including quorum. We precisely capture this property, and show that together with the other two properties, it is sufficient to implement reliable broadcast and consensus abstractions. We present *protocols for both reliable broadcast and consensus*, and prove that if the underlying quorum system has quorum intersection, availability, and subsumption for certain quorums, then the protocols satisfy the required safety and liveness properties.

In summary, this paper makes the following contributions.

- Properties of quorum-based protocols (Section 3) and specifications of reliable broadcast and consensus on heterogeneous quorum systems (Section 4).
- Proof of insufficiency of quorum intersection and availability to solve consensus (Subsection 5.1) and reliable broadcast (Subsection 5.2).
- Sufficiency of quorum intersection, quorum availability and quorum subsumption to solve consensus and reliable broadcast. We present protocols for reliable broadcast (Subsection 6.1) and consensus (Subsection 6.2), and their proofs of correctness.

2 Heterogeneous Quorum Systems

A quorum is a subset of processes that are collectively trusted to perform an operation. However, this trust may not be uniform: while a process may trust a part of a system, another process may not trust that same part. In this section, we adopt a general model of quorum systems [32, 34] and its properties. These basic definitions adapt common properties of quorum systems to the heterogeneous setting, and serve as the foundation for theorems and protocols in the later sections. Since we want the theorems to be as strong as possible, we introduce the weak notion of quorum subsumption in this paper.

2.1 Processes and Quorums

Processes and Failures. A quorum system is hosted on a set of processes $\mathcal{P}$. For every execution, we can partition the set $\mathcal{P}$ into *Byzantine* $\mathcal{B}$ and *well-behaved* $\mathcal{W} = \mathcal{P} \setminus \mathcal{B}$ processes. Well-behaved processes follow the given protocol, while Byzantine processes can deviate from the protocol arbitrarily.

We assume that the network is partially synchronized, *i.e.*, after an unknown global stabilization time (GST), if both the sender and receiver are well-behaved, the message will eventually be delivered with a known bounded delay [20].

Heterogeneous Quorum Systems (HQS). To represent subjective trust, we let each process specify its own quorums. A quorum q of process p is a non-empty subset P of $\mathcal{P}$ that p trusts to get information from if it obtains the same information from each member of P. (In practice, a quorum of p can contain p itself, although the model does not require it.) Each process p stores its own set of quorums that we call individual quorums of p. Any superset of a quorum of p is also a quorum of p; thus, there are minimal quorums: a quorum of p is a minimal quorum of p if none of its strict subsets is a quorum of p. Thus, to avoid redundancy, p can ignore its quorums that are proper supersets of its minimal quorums. Thus, each process stores only its individual minimal quorums.

▶ **Definition 1** (Quorum System). *A heterogeneous quorum system $\mathcal{Q}$ is a mapping from processes to their non-empty set of individual minimal quorums.*

Since the trust assumptions of Byzantine processes can be arbitrary, their quorums can be left unspecified. Figure 1 presents an example quorum system. When obvious from the context, we say quorums of p to refer to the individual minimal quorums of p, and use $\mathcal{Q}$ to refer to the set of all individual minimal quorums of the system, i.e. the co-domain of $\mathcal{Q}$. Additionally, we say quorum systems to refer to heterogeneous quorum systems. A process p is a *follower* of a process p' iff there is a quorum $q \in \mathcal{Q}(p)$ that includes p'.

In dissemination quorum system (DQS) [37] (and the cardinality-based quorum systems as a special case), quorums are uniform for all processes. Processes have the same set of individual minimal quorums. For example, a quorum system that tolerates f Byzantine failures out of $3f + 1$ processes considers any set of $2f + 1$ processes as a quorum for all processes.

2.2 Properties

A quorum system is expected to maintain certain properties in order to provide distributed abstractions such as Byzantine reliable broadcast and consensus. Quorum intersection and quorum availability are well-established requirements for quorum systems. In the following section, we will see their adaption to HQS. Further, we identify a new property we call quorum subsumption that helps achieve the aforementioned abstractions on HQS. Finally, we briefly present a few related quorum systems, and their properties.

Quorum Intersection. Processes store and retrieve information from the quorum system by communicating with its quorums. To ensure that information is properly passed from a quorum to another, the quorum system is expected to maintain a well-behaved process at the intersection of every pair of quorums. For example, in the running example in Figure 1, all the quorums of well-behaved processes intersect at at least one of well-behaved processes in $\{1, 3, 4\}$.

$$\begin{aligned}
&\mathcal{P} = \mathcal{W} \cup \mathcal{B}, \quad \mathcal{W} = \{1,3,4,5\}, \quad \mathcal{B} = \{2\} \\
&\mathcal{Q} = \{1 \mapsto \{\{1,2,3\},\{1,4\}\}, \\
&\qquad 3 \mapsto \{\{3,4\},\{1,3\}\} \\
&\qquad 4 \mapsto \{\{3,4\}\} \\
&\qquad 5 \mapsto \{\{1,2,3,5\}\}\}
\end{aligned}$$

Figure 1 Quorum System Example.

▶ **Definition 2** (Quorum Intersection). *A quorum system $\mathcal{Q}$ has quorum intersection iff every pair of quorums of well-behaved processes in $\mathcal{Q}$ intersect at a well-behaved process, i.e.,* $\forall p, p' \in \mathcal{W}.\ q \in \mathcal{Q}(p).\ q' \in \mathcal{Q}(p').\ q \cap q' \cap \mathcal{W} \neq \emptyset$

Quorum Availability. In order to support progress for a process, the quorum system is expected to have at least one quorum for that process whose members are all well-behaved. We say that the quorum system is weakly available for that process. (In the literature, this notion of availability is often unqualified, but we explicitly contrast the weak notion to the strong notion that we will define.) In classical quorum systems, any quorum is a quorum for all processes. This guarantees that if the quorum system is available for a process, it is available for all processes. However, this is obviously not true in a heterogeneous quorum system where quorums are not uniform. In this setting, we weaken the availability property so that it requires only a subset and not necessarily all well-behaved processes to have a well-behaved quorum. In Figure 1, $\mathcal{Q}$ is available for the set $\{1,3,4\}$: the quorum $\{1,4\}$ of process 1, and the quorum $\{3,4\}$ of processes 3 and 4 make them weakly available. Each process in that subset can always communicate with a quorum independently of Byzantine processes.

▶ **Definition 3** (Weak Availability). *A quorum system is weakly available for a set of processes P iff every process in P has at least one quorum that is a subset of well-behaved processes $\mathcal{W}$. A quorum system is available iff it is available for a non-empty set of processes.*

If a quorum system is weakly available, there is at least one well-behaved process that can communicate with a quorum independently of Byzantine processes.

With quorum availability introduced, we can consider when a quorum system is unavailable. A quorum system is unavailable for a process when that process has no quorum in $\mathcal{W}$, *i.e.*, the Byzantine processes $\mathcal{B}$ can block every one of its quorums. We generalize this idea in the notion of blocking.

▶ **Definition 4** (Blocking Set). *A set of processes P is a blocking set for a process p (or is p-blocking) if P intersects every quorum of p.*

For example, consider cardinality-based quorum systems where the system contains $3f+1$ processes. Any set of size $f+1$ is a blocking set for all well-behaved processes, since a set with $f+1$ processes intersects with any quorum, a set with $2f+1$ processes. In Figure 1, well-behaved process 5 is blocked by $\{2\}$, since its only quorum $\{1,2,3,5\}$ intersect with $\{2\}$

Notice also that the definition does not stipulate that the blocking set is Byzantine, but rather it is more general. The concept of blocking will be useful for designing our protocols in (Section 6). For now, we prove a lemma for blocking sets. In order to state the lemma, we generalize the notion of availability. Given a set of processes P, we generalize availability for P at the complete set of well-behaved processes $\mathcal{W}$ (Definition 3) to availability for P at a subset P' of well-behaved processes. We say that a quorum system is weakly available for a set of processes P at a subset of well-behaved processes P' iff every process in P has at least one quorum that is a subset of P'.

Table 1 Non-termination for Bracha protocol with blocking sets.

sender	1	2	3	4
$\mathit{BCast}(m_1)$				
	$\mathit{Echo}(m_1)$		$\mathit{Echo}(m_1)$	$\mathit{Echo}(m_1)$
		$\mathit{Ready}(m_2)$		
	$\mathit{Ready}(m_1)$		$\mathit{Ready}(m_2)$	$\mathit{Ready}(m_2)$
	blocked forever			$\mathit{Deliver}(m_2)$

▶ **Lemma 5.** *In every quorum system that is weakly available for a set of processes P at P', every blocking set of every process in P intersects P'.*

Proof. Consider a quorum system that is weakly available for P at P', a process p in P, and a set of processes P'' that blocks p. By the definition of available, there is at least one quorum q of p that is a subset of P'. By the definition of blocking set (Definition 4), q intersects with P''. Hence, P' intersects P'' as well. ◀

Quorum subsumption. We now introduce the notion of quorum subsumption.

▶ **Definition 6** (Quorum Subsumption). *A quorum system $\mathcal{Q}$ is quorum subsuming for a quorum q iff every process in q has a quorum that is included in q, i.e., $\forall p \in q.\ \exists q' \in \mathcal{Q}(p).\ q' \subseteq q$. We say that $\mathcal{Q}$ is quorum subsuming for a set of quorums if it is quorum subsuming for each quorum in the set.*

In Figure 1, $\mathcal{Q}$ is quorum subsuming for $\{3, 4\}$: both members in this quorum have the quorum $\{3, 4\}$ that is trivially a subset of itself. However, $\mathcal{Q}$ is not quorum subsuming for process 1's quorum $\{1, 4\}$: process 4's only quorum $\{3, 4\}$ is not a subset of $\{1, 4\}$.

Quorum subsumption is inspired by and weakens the notion of quorum sharing [34]. Quorum sharing requires the above subsumption property for all quorums. Thus, many quorum systems including Ripple and Stellar do not satisfy it (unless Byzantine processes do not lie about their slices [34].) They can maintain the subsumption property only for quorums of a well-behaved subset of processes. In particular, no requirement can be made for quorums of Byzantine processes. Therefore, we define the weaker notion of quorum subsumption for a subset of quorums, and later show that it is sufficient to implement broadcast and consensus.

In order to make progress, protocols (such as Bracha's Byzantine reliable broadcast [9]) require the members of a quorum to be able to communicate with at least one of their own quorums, or communicate with a subset of processes that contains at least one well-behaved process. Let us see intuitively how quorum subsumption can support liveness properties. Consider a quorum system $\mathcal{Q}$ for processes $\mathcal{P} = \{1, 2, 3, 4\}$ where the Byzantine processes are $\{2\}$, and $\mathcal{Q}(1) = \{\{1, 3, 4\}\}$, $\mathcal{Q}(3) = \{\{1, 2, 3\}\}$, and $\mathcal{Q}(4) = \{\{2, 3, 4\}\}$. The quorum system $\mathcal{Q}$ has quorum intersection, and is weakly available for the set $\{1\}$ since there is a well-behaved quorum $\{1, 3, 4\}$ for the process 1. In the classic Bracha protocol, the sender broadcasts $\mathit{Echo}(m)$, a well-behaved broadcasts $\mathit{Echo}(m)$ when it receives it from the sender, it broadcasts $\mathit{Ready}(m)$ after receiving $2f + 1$ $\mathit{Echo}(m)$ or $f + 1$ $\mathit{Ready}(m)$ messages, and finally, delivers m if it receives $2f + 1$ $\mathit{Ready}(m)$ messages. In Stellar [33] and follow-up works [34, 24, 15], the check for receiving $\mathit{Ready}(m)$ messages from $f + 1$ processes is replaced with receiving $\mathit{Ready}(m)$ messages from a blocking set of the current process. Let's consider the example execution presented in Table 1; it gives an intuition of why the quorum system needs stronger conditions than weak availability. Consider a Byzantine sender who sends $\mathit{BCast}(m_1)$ to process $\{1, 3, 4\}$. Well-behaved process $\{1, 3, 4\}$ sends out $\mathit{Echo}(m_1)$ to each other. We let process 1 deliver *Echo*(m_1) messages from process 1, 3, and 4 first; it then sends out *Ready*(m_1) messages. We note that the two processes 3, and 4 cannot broadcast *Ready*(m_1) since they have not received *Echo*(m_1) from a quorum of their own. Then the Byzantine process 2 sends *Ready*(m_2) messages to process $\{3, 4\}$. Since the set $\{2\}$ is blocking for the quorums of both processes 3 and 4, both send out *Ready*(m_2) messages. These broadcast protocols prevent a process that is ready for a value from getting ready for another value. Therefore, although $\{3\}$ and $\{4\}$ are both blocking sets for the process 1, it cannot become ready for m_2. Process 1 never receives enough Thus, *Ready* messages for either m_1 or m_2 to deliver a message, and is blocked forever. If the quorum $\{1, 3, 4\}$ for 1 had the quorum subsumption property, then 3 and 4 could send out *Ready*(m_1) messages, and eventually 1 would make progress.

Complete Quorum. We will later see that quorum availability and quorum subsumption are important together for liveness. We succinctly combine the two properties into the notion of complete quorums.

▶ **Definition 7** (Complete Quorum). *A quorum q in a quorum system $\mathcal{Q}$ is a complete quorum if all its members are well-behaved, and $\mathcal{Q}$ is quorum subsuming for q.*

In our previous running example Figure 1, quorum $\{3, 4\}$ is a complete quorum: both of its members are well-behaved and $\mathcal{Q}$ is quorum subsuming for $\{3, 4\}$.

▶ **Definition 8** (Strong Availability). *A quorum system $\mathcal{Q}$ has strong availability for a subset of processes P iff every process in P has at least one complete quorum. We call P a* strongly available set *for $\mathcal{Q}$, and call a member of P a* strongly available process*. We say that $\mathcal{Q}$ is strongly available if it is strongly available for a non-empty set.*

Intuitively, operations stay available at a strongly available process since its complete quorum can perform operations on his behalf in the face of Byzantine attacks. In Figure 1, $\mathcal{Q}$ is strongly available for $\{3, 4\}$. In contrast, $\mathcal{Q}$ is only weakly available for process 1, since its quorum $\{1, 2, 3\}$ includes 2 that is not well-behaved, and its other quorum $\{1, 4\}$ is well-behaved but not a complete quorum.

By Lemma 5, every blocking set of every strongly available process contains at least one well-behaved process.

3 Protocol Implementation

In the subsequent sections, we will see that it is impossible to construct a protocol for Byzantine reliable broadcast and consensus in an HQS given only quorum intersection and quorum availability. After that, we give a protocol for Byzantine reliable broadcast and consensus for an HQS that has quorum intersection and strong availability. We first need a model of quorum-based protocols, and then the exact specifications of the distributed abstractions we aim to design protocols for. In this section, we consider the former.

We consider a modular design for protocols. A protocol is captured as a component that accepts request events and issues response events. A component uses other components as sub-components: it issues requests to them and accepts responses from them. A component stores a state and defines handlers for incoming requests from the parent component, and incoming responses from children components. Each handler gets the pre-state and the incoming event as input, and outputs the post-state and outgoing events, either as responses to the parent or requests to the children components. The outputs of a handler can be deterministically a function of its inputs, or randomized.

▶ **Definition 9** (Determinism)**.** *A protocol is deterministic iff the outputs of its handlers are a function of the inputs.*

Quorum-based Protocols. A large class of protocols are implemented based on quorum systems. In order to state impossibility results for these protocols, we capture the properties of quorum-based protocols [34, 29] as a few axioms. Our impossibility results concern protocols that adhere to the necessity, sufficiency, and locality axioms.

A process in a quorum-based protocol should process a request only if it can communicate with at least one of its quorums.

▶ **Axiom 1** (Necessity of Quorums [34])**.** *If a well-behaved process p issues a response for a request then there must be a quorum q of p such that p receives at least one message from each member of q.*

In a quorum-based protocol, a process only needs the participation of itself and members of one of its quorums to deliver a message.

▶ **Axiom 2** (Sufficiency of Quorums)**.** *For every execution where a well-behaved process p issues a response, there exists an execution where only p and a quorum of p take steps, and p eventually issues the same response.*

We add a remark for Byzantine reliable broadcast (BRB) which has a designated sender process. We will use a slight variant of the sufficiency axiom for BRB that states that there exists an execution where only *the sender*, p and a quorum of p take steps.

A process's local state is only affected by the information that it receives from the members of it's quorums.

▶ **Axiom 3** (Locality)**.** *The state of a well-behaved process changes upon receiving a message only if the sender is a member of one of its quorums.*

For BRB, we will use a slight variant of the locality axiom that allows processes change state upon receiving messages from *the sender* in addition to members of quorums.

4 Protocol Specification

We now define the specification of reliable broadcast and consensus for HQS. The liveness properties are weaker than classical notions since in an HQS, availability might be maintained only for a subset P of well-behaved processes.

Reliable Broadcast. We now define the specification of the reliable broadcast abstraction. The abstraction accepts a single broadcast request from a designated sender (either in the system or a process that is separate from the other processes in system), and issues delivery responses.

▶ **Definition 10** (Specification of Reliable Broadcast)**.**

- *(Validity for a set of well-behaved processes P). If a well-behaved process p broadcasts a message m, then every process in P eventually delivers m.*
- *(Integrity). If a well-behaved process delivers a message m from a well-behaved sender p, then m was previously broadcast by p.*
- *(Totality for a set of well-behaved processes P). If a message is delivered by a well-behaved process, then every process in P eventually delivers a message.*
- *(Consistency). No two well-behaved processes deliver different messages.*
- *(No duplication). Every well-behaved process delivers at most one message.*

We also consider a variant of reliable broadcast called federated voting. Similar to reliable broadcast, the abstraction accepts a broadcast request from processes, and issues delivery responses. In contrast to reliable broadcast where there is a dedicated sender, in federated voting, every process can broadcast a message. The specification of federated voting is similar to that of reliable broadcast except for validity. The messages that well-behaved processes broadcast may not be the same. Therefore, the validity property provides guarantees only when the messages are the same or there is only one sender. The validity property for a set of well-behaved processes P guarantees that if all well-behaved processes broadcast a message m, or only one well-behaved process broadcasts a message m, then every process in P eventually delivers m.

Consensus. We now consider the specification of the consensus abstraction. It accepts propose requests from processes in the system, and issues decision responses.

▶ **Definition 11** (Specification of Consensus).

- *(Validity). If all processes are well-behaved, and some process decides a value, then that value was proposed by some process.*
- *(Agreement). No two well-behaved processes decide differently.*
- *(Termination for a set of well-behaved processes P). Every process in P eventually decides.*

5 Impossibility

We now present the impossibility results for consensus and Byzantine Reliable Broadcast (BRB). It is known that quorum intersection and quorum availability are necessary conditions [34] to implement consensus and BRB protocols. In this section, we show that while these two conditions are necessary, they are not sufficient.

We consider the information-theoretic settings (Fault axiom [21]), where byzantine processes have unlimited computational power, and can show arbitrary behavior. However, processes communicate only over secure channels so that the recipient knows the identity of the sender. A Byzantine process is unable to impersonate a well-behaved process. This is similar to the classic unauthenticated Byzantine general problem [30], and is necessary for open decentralized blockchains and HQS, where the trusted authorities including public key infrastructures may not be available.

The two proofs will take a similar approach. First, we assume there does exist a protocol for our distributed abstraction that satisfies all the desired specifications. We then present a quorum system $\mathcal{Q}$ and consider its executions that have quorum intersection and availability in the face of Byzantine attacks. We then show through a series of indistinguishable executions that the protocol cannot satisfy all the desired specifications, leading to a contradiction. The high-level idea is that in the information-theoretic setting, a well-behaved process is not able to distinguish between an execution where the sender is Byzantine and sends misleading messages, and an execution where the relaying process is Byzantine and forwards misleading messages. For example, let p_1, p_2 and p_3 be three processes in the system. When p_3 receives conflicting messages from p_1 through p_2, it does not know whether p_1 or p_2 is Byzantine. This eventually leads to violation of the agreement or validity property of the abstraction.

We consider binary proposals for consensus, and binary values (from the sender) for reliable broadcast. For the consensus abstraction, we succinctly present the values that processes propose as as a vector of values that we call a configuration. If the initial value of a process is $\perp$ in the configuration, that process is considered Byzantine. Otherwise, the process is well-behaved. For example, a configuration $C = \langle 0, 0, \perp \rangle$ denotes the first and second process proposing zero and the third process being Byzantine.

5.1 Consensus

We first consider consensus protocols in HQS.

▶ **Theorem 12.** *Quorum intersection and weak availability are not sufficient for deterministic quorum-based consensus protocols to provide validity, agreement and termination for weakly available processes.*

Proof. We suppose there is a quorum-based consensus protocol that guarantees validity, agreement, and termination for every quorum system $\mathcal{Q}$ with quorum intersection and weak availability, towards contradiction. Consider a quorum system $\mathcal{Q}$ for processes $\mathcal{P} = \{a, b, c\}$ with the following quorums: $\mathcal{Q}(a) = \{\{a, c\}\}$, $\mathcal{Q}(b) = \{\{a, b\}\}$, $\mathcal{Q}(c) = \{\{b, c\}\}$.

We make the following observations: (1) if all processes are well-behaved, then $\mathcal{Q}$ has quorum intersection and weak availability for $\{a, b, c\}$, (2) if only process a is Byzantine, then $\mathcal{Q}$ preserves quorum intersection, and weak availability for $\{c\}$, (3) if only process c is Byzantine, then $\mathcal{Q}$ preserves quorum intersection, and weak availability for $\{b\}$. Going forward, we implicitly assume termination for weakly available processes.

Now consider the following four configurations as shown in Figure 2: $C_0 = \langle 0, 0, 0 \rangle$, $C_1 = \langle 1, 1, 1 \rangle$, $C_2 = \langle 0, 1, \perp \rangle$, and $C_3 = \langle \perp, 1, 1 \rangle$. The goal is now to show a series of executions over the configurations so that at least one property of the protocol is violated.

- We begin with execution E_0 (shown in red) with the initial configuration C_0. All the messages between a and c are delivered. By termination for weakly available processes and validity, process a decides 0. Additionally, by quorum sufficiency, a can reach this decision with only processes $\{a, c\}$ taking steps.
- Next, we have execution E_1 (shown in blue) with initial configuration C_1. All the messages between b and c are delivered. Again, by termination for weakly available processes and validity, process c decides 1. By quorum sufficiency, c can reach this decision with only processes $\{b, c\}$ taking steps.
- Next, we have execution E_2 as a sequence of E_1 and E_0, with initial configuration C_2. Suppose messages between well-behaved processes a and b are delayed. Byzantine process c first replays E_1 with process b, then replays E_0 with process a. This cause process a to decide 0. Now let Byzantine process c stay silent, and messages between processes a and b be delivered. By termination for b, agreement and quorum sufficiency, process a makes b decide 0 as well (shown in green).
- Lastly, we have execution E_3 with initial configuration C_3. Suppose messages between b to c are delivered in the beginning. We let processes $\{b, c\}$ replay E_1; thus, c decides 1. Then, Byzantine process a sends messages to b as if it were at the end of E_2. In turn, b decides 0. Thus, agreement is violated as two well-behaved processes decided differently. ◀

Indistinguishably. We provide some intuition for the proof construction. Ultimately, the problem lies in process b not being able to distinguish whether process a or process c is the Byzantine process. More specifically, both E_2 and E_3 begin with execution E_1. Since process b cannot distinguish between the two executions, it does not know which value to decide. If process b believes E_2 is the actual execution, then b should decide 0 to agree with the

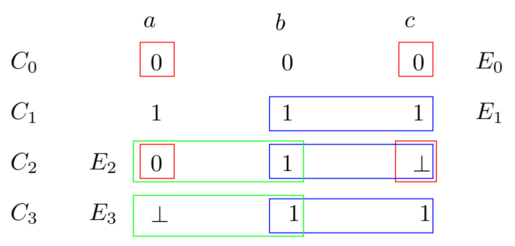


Figure 2 Indistinguishable Executions.

decision of well-behaved process a. However, if E_3 is the actual execution, then agreement is violated as process c decided 1. Conversely, if process b believes E_3 is the actual execution, then b should decide 1 to agree with the decision of well-behaved process c. Then, if E_2 is the actual execution, agreement is violated as the well-behaved process a decided 0.

We note that this proof could not be constructed if there was quorum subsumption. For example, if the process b adds the quorum $\{a, b, c\}$, then $\mathcal{Q}$ will have quorum subsumption for the quorum $\{a, b, c\}$ of b. However, then by quorum subsumption, there will be no Byzantine process, and the executions E_2 and E_3 cannot be constructed. If the process a adds the quorum $\{a, b\}$, then it will have quorum subsumption. However, then the process a cannot Byzantine process anymore, and the executions E_3 cannot be constructed. Similarly, if the process b adds the quorum $\{b, c\}$, the executions E_2 cannot be constructed.

5.2 Byzantine Reliable Broadcast

Now, we prove the insufficiency of quorum intersection and quorum availability for Byzantine reliable broadcast.

For the reliable broadcast abstraction, we represent the initial configuration as an array of values received by the processes from the sender. The sender is a fixed and external process in the executions, and is only used to assign input values for processes in the system, which are captured as the initial configurations. The sender does not take steps in the executions, and processes are not able to distinguish executions based on the sender.

▶ **Theorem 13.** *Quorum intersection and weak availability are not sufficient for deterministic quorum-based reliable broadcast protocols to provide validity and totality for weakly available processes, and consistency.*

Proof. The proof is similar to the proof for consensus. In fact, we will reuse the construction. There are differences between reliable broadcast and consensus specifications in (1) their validity properties, and (2) their totality and termination properties respectively. The proof can be adjusted for these differences. For reliable broadcast, we need a sender process s who broadcasts a message. In executions that we want a well-behaved process to deliver the message m, we either (1) keep the sender s well-behaved and have it send m, and then apply validity, or (2) have a process deliver m, then apply totality and consistency. The initial configuration represents values received by each process from the sender.

Executions follow those in the previous proof. Message delivery and delays mirror the previous executions. In execution E_0 for configuration C_0, the well-behaved sender s broadcasts 0, and messages between processes a and c are delivered. By validity for weakly available processes, process a delivers 0, and by quorum sufficiency, only processes $\{a, c\}$ need to take steps. In execution E_1 for configuration C_1, the well-behaved sender s broadcasts 1, and messages between processes b and c are delivered. By validity for weakly available

Algorithm 1 Byzantine Reliable Broadcast (BRB).

```
Implements: ReliableBroadcast
    request : $broadcast(v)$
    response : $deliver(v)$
Vars:
    $Q$                                  ▷ Minimal quorums of self
    $\mathcal{F}$ : Set[$\mathcal{P}$]                  ▷ The followers of self
    $echoed, readied, delivered$ : Boolean ← false
    $E, R : V \mapsto$ Set[$\mathcal{P}$] ← $\emptyset$
                         ▷ Set of echoed and readied processes
Uses:
    $ptp$ : PointToPointLink
upon request $broadcast(v)$ from sender
    $ptp$ request $send(p, BCast(v))$ for each $p \in \mathcal{P}$
upon $ptp$ response $deliver(p', BCast(v))$
    if $\neg echoed$ then
        $echoed$ ← true
        $ptp$ request $send(p, Echo(v))$ for each $p \in \mathcal{F}$
upon $ptp$ response $deliver(p', Echo(v))$
    $E(v) \leftarrow E(v) \cup \{p'\}$
    if $\neg readied \wedge \exists q \in Q.\ q \subseteq E(v)$ then
        $readied$ ← true
        $ptp$ request $send(p, Ready(v))$ for each $p \in \mathcal{F}$
upon $ptp$ response $deliver(p', Ready(v))$
    $R(v) \leftarrow R(v) \cup \{p'\}$
    if $\neg readied \wedge$ $R$ is a blocking set of self then
        $readied$ ← true
        $ptp$ request $send(p, Ready(v))$ for each $p \in \mathcal{F}$
    if $\neg delivered \wedge \exists q \in Q.\ q \subseteq R(v)$ then
        $delivered$ ← true
        response $deliver(v)$
```

processes, and quorum sufficiency, process c delivers 1, only with $\{b, c\}$ taking steps. In configurations C_2 and C_3, the sender s is Byzantine. The messages between processes a and b are delayed in the beginning. In execution E_2 for configuration C_2, the Byzantine sender s and Byzantine process c replay E_1 with process b, then replay E_0 with process a. Then Byzantine process c stays silent, and messages between processes a and b are delivered. By totality for weakly available processes, since process a delivers 0, then process b will also deliver a value. By consistency, process b delivers 0 as well. In the last execution E_3 for configuration C_3, we let the Byzantine process a stay silent in the beginning, and processes b and c replay E_1. Thus, process c delivers 1. Afterwards, messages between process b and c are delayed, and the Byzantine process a replays E_2. Again, process b cannot distinguish between the two executions E_2 and E_3. Since process a sends the exact same messages to process b as the end of E_2, process b will deliver 0. Thus, consistency between c and b is violated. ◀

6 Protocols

We just showed that quorum intersection and availability are not sufficient to implement our desired distributed abstractions. Now, we show that quorum intersection and strong availability, our newly introduced property are sufficient to implement both Byzantine reliable broadcast and consensus.

6.1 Reliable Broadcast Protocol

In Algorithm 1, we adapt the Bracha protocol [9] to show that quorum intersection and strong availability together are sufficient for Byzantine reliable broadcast. The parts that are different from the classical protocol are highlighted in blue.

Each process stores the set of its individual minimal quorums Q, and its set of followers $\mathcal{F}$. It also stores the boolean flags *echoed*, *readied*, and *delivered* which record actions the process has taken to avoid duplicate actions. It further uses point-to-point links *ptp* to each of its followers. Upon receiving a request to broadcast a value v (at L. 11), the sender broadcasts the value v to all processes (at L. 12). Upon receiving the message from the sender (at L. 13), a well-behaved process echoes the message among its followers (at L. 16) only if it has not already *echoed*. When a well-behaved process receives a quorum of consistent echo messages (at L. 17), it sends ready messages to all its followers (at L. 21). A well-behaved process can also send a ready message when it receives consistent ready messages from a blocking set (at L. 24). When a well-behaved process receives a quorum of consistent ready messages for v (at L. 27), it delivers v (at L. 29). The implementation of the federated voting abstraction is similar. The only difference is that there can be multiple senders (at L. 11).

We prove that this protocol implements Byzantine reliable broadcast when the quorum system satisfies quorum intersection, and strong availability. We remember that strong availability requires both weak availability and quorum subsumption. More precisely, it requires a well-behaved quorum q for a process p, and quorum subsumption for q.

▶ **Theorem 14.** *Quorum intersection and strong availability are sufficient to implement Byzantine reliable broadcast.*

This theorem follows from five lemmas in the appendix [31] that prove the protocol satisfies the specification of Byzantine reliable broadcast that we defined in Definition 10. Consider a quorum system with quorum intersection, and strong availability for P. Here, we state and prove only the validity property.

▶ **Lemma 15.** *The BRB protocol guarantees validity for P.*

Proof. Consider a well-behaved sender that broadcasts a message m. We show that every process in P eventually delivers m. By availability, every process $p \in P$ has a complete quorum q. Consider a process $p' \in q$. By quorum subsumption, p' has a quorum $q' \subseteq q$. By availability, all members of q (including q') are well-behaved. Thus, when they receive m from the sender, they all echo it to their followers. The processes in q' have p' as a follower. Thus, p' receives consistent echo messages for m from one of its quorums q'. Thus, p' sends out ready messages for m to its followers. Thus, all processes in q send out ready messages for m to their followers. The processes in q have p as a follower. Therefore, p receives a quorum of consistent ready messages for m from one of its quorums q, and delivers m. ◀

6.2 Byzantine Consensus Protocol

In this section, we show that quorum intersection and strong availability are sufficient to implement Byzantine consensus. We first present the consensus protocol for heterogeneous quorum systems, and then prove its correctness.

At a high level, the protocol proceeds in rounds with assigned leaders for each. Ballots that carry proposal values are totally ordered. A leader tries to commit its own candidate ballot only after aborting any lower ballot in the system. Leaders use the federated voting abstraction (that we saw in Section 4) to abort or commit ballots. There may be multiple leaders or Byzantine leaders before GST, and they may broadcast contradicting abort and commit messages for the same ballot. However, by the consistency property of federated voting, processes agree on aborting or committing ballots.

A ballot b is a pair $\langle r, v\rangle$ of a round number r and a proposed value v. Ballots are totally ordered by first their round numbers, and then their values: a ballot $\langle r, v\rangle$ is below another $\langle r', v'\rangle$, written as $\langle r, v\rangle < \langle r', v'\rangle$, if $r < r'$ or $r = r' \wedge v < v'$. Two ballots $b = \langle r, v\rangle$ and $b' = \langle r', v'\rangle$ are compatible, $b \sim b'$, if they have the same value, *i.e.*, $v = v'$; otherwise, they are incompatible, $b \not\sim b'$. We say that a ballot is below and incompatible with another,

Algorithm 2 Byzantine Consensus.

```
1  Implements: Consensus
2    request : $propose(v)$
3    response : $decice(v)$
4  Vars:
5    $round : \mathbb{N}^+ \leftarrow 0$   ▷ Current round number
6    $candidate, prepared : \langle \mathbb{N}^+, V \rangle \leftarrow \langle 0, \bot \rangle$
7    $leader : \mathcal{P} \leftarrow p_0$   ▷ current leader
8  Uses:
9    $fv : B \mapsto$ ByzantineReliableBroadcast
10   $le$ : EventualLeaderElection
11 upon request $propose(v)$
12     $candidate \leftarrow \langle 1, v \rangle$
13     if self $= leader$ then
14         $fv(b')$ request $broadcast(\mathbb{A})$ for all $b' \lnsim candidate$
15 upon $fv(b')$ response $deliver(p, \mathbb{A})$ for all $b' \lnsim b$ where $prepared < b$
16     $prepared \leftarrow b$
17     if self $= leader \wedge prepared = candidate$ then
18         $fv(candidate)$ request $broadcast(\mathbb{C})$
19 upon $fv(b)$ response $deliver(p, \mathbb{C})$ where $b = prepared \wedge p = leader$
20     response $decice(b.v)$
21 upon $timeout$ triggered
22     $le$ request $Complain(round)$
23 upon $le$ response $new\text{-}leader(p)$
24     $leader \leftarrow p$
25     $round \leftarrow round + 1$
26     if self $= leader$ then
27         Delay for time $\Delta$
28     start-timer$(round)$
29     if $prepared = \langle 0, \bot \rangle$ then
30         $candidate \leftarrow \langle round, candidate.v \rangle$
31     else
32         $candidate \leftarrow \langle round, prepared.v \rangle$
33     if self $= leader$ then
34         $fv(b')$ request $broadcast(\mathbb{A})$ for all $b' \lnsim candidate$
```

$b \lnsim b'$, if $b < b'$ and $b \not\sim b'$. For message passing communication, we assume batched network semantics (BNS), where messages issued in an event are sent as a batch, and the receiving process delivers and processes the batch of messages together. (In particular, as we will see later in the correctness proofs, if prepare messages that are sent together are not processed together the validity property can be violated.)

The protocol is similar to SCP [38, 25] in structure; the important difference is that this protocol uses leaders [34] and guarantees termination. Our protocol guarantees termination regardless of Byzantine processes. On the other hand, the SCP protocol guarantees a liveness property called *non-blocking* which requires Byzantine processes to stop. (More precisely, if a process p in the intact set [38, 24] has not yet decided in some execution, then for every continuation of that execution in which all the Byzantine processes stop, the process p eventually decides.)

Each process stores four local variables: *round* is the current round number, *candidate* is the ballot that the process tries to commit, *prepared* is the ballot that the process is safe to discard any ballots lower and incompatible with, and *leader* is the current leader. Each process uses an instance of federated voting for each ballot, and an eventual leader election module. The latter issues *new-leader* events, and eventually elects a well-behaved process as the leader. (Previous work [34] presented a probabilistic leader election module.)

Upon receiving a proposal request (at L. 11), a well-behaved process initializes its candidate ballot to the pair of the first round and its own proposal (at L. 12). If the current process **self** is the leader, it tries to prepare its *candidate* by broadcasting abort $\mathbb{A}$ messages for all ballots with *candidate* (at L. 14). When a well-behaved process delivers $\mathbb{A}$ messages from the leader for all ballots below and incompatible with some ballot b, and its current *prepared* ballot is below b (at L. 15), it sets *prepared* to b (at L. 16). If the current process **self** is the leader, and the *prepared* ballot is equal to the *candidate* ballot, then it broadcasts

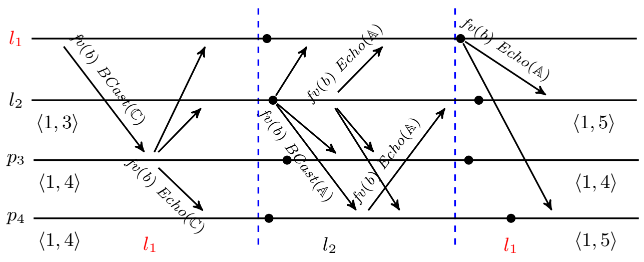

l_1
l_2
p_3
p_4
$\langle 1,3\rangle$
$\langle 1,4\rangle$
$\langle 1,4\rangle$
$\langle 1,5\rangle$
$\langle 1,4\rangle$
$\langle 1,5\rangle$
l_1
l_2
l_1
$fv(b)$ $BCast(\mathbb{C})$
$fv(b)$ $Echo(\mathbb{C})$
$fv(b)$ $Echo(\mathbb{A})$
$fv(b)$ $BCast(\mathbb{A})$
$fv(b)$ $Echo(\mathbb{A})$
$fv(b)$ $Echo(\mathbb{A})$

Figure 3 Last Minute Attack. $b = \langle 1, 4\rangle$. The *candidate* of well-behaved leader l_2 is $b' = \langle 2, 3\rangle$. The votes $\mathbb{C}$ and $\mathbb{A}$ are abbreviated as C and A. The new leader events are triggered at the black dots at each process. Prepared ballots are shown below the time line for each process.

a commit $\mathbb{C}$ message for its *candidate* ballot (at L. 18). When a well-behaved process delivers a $\mathbb{C}$ message for a ballot b from the leader, and it has already prepared the same ballot (at L. 19), it decides the value of that ballot (at L. 20).

To ensure liveness, a well-behaved process triggers a timeout if no value is decided after a predefined time elapses in each round. The process then complains to the leader election module (at L. 22). When the leader election module issues a new leader (at L. 23), a well-behaved process updates its *leader* variable, and increments the *round* number (at L. 25). The leader itself then waits for a time Δ (at L. 27) which we will further explain below. The process also resets the timer with a doubled timeout for the next round (at L. 28). It then updates the *candidate* ballot: if no value is prepared before, the *candidate* ballot is updated to the new round number and the value of the current *candidate* (at L. 30); otherwise, it is updated to the new round number and the value of the *prepared* ballot (at L. 32). Then, the leader tries to prepare the *candidate* by aborting below and incompatible ballots similar to the steps above (at L. 34).

Let us now explain why delay Δ is needed for termination. Without this delay, a Byzantine leader can perform a last minute attack that we illustrate in Figure 3. Consider that we have four processes, one of them is Byzantine, and any set of three processes is a quorum. Let the Byzantine process be the leader l_1, and let the ballot b be prepared. The leader l_1 sends a commit for ballot b to one well-behaved process p_3. Then, p_3 echos commit for b. Then, the timeout for l_1 happens, and the next well-behaved leader l_2 comes up. Without the delay, l_2 may have not prepared b yet (although other well-behaved processes p_3 and p_4 prepared it). Therefore, the ballot b' that l_2 updates its candidate to (at L. 32) is not b, and may not be compatible with b. In order to prepare b', the leader l_2 tries to abort b (at L. 34) but b cannot be aborted: in order to abort b, a quorum of processes should echo it. However, the well-behaved process p_3 has already echoed commit, and if the Byzantine process l_1 remains silent, the remaining two well-behaved processes l_2 and p_4 are not a quorum, and cannot abort b. Therefore, l_2 cannot succeed, and the timeout is triggered. Further, if the next leader is the Byzantine process l_1 again, it can repeat the above scenario: it can abort b to prepare a higher ballot b_2, and make a well-behaved process echo commit for b_2, before passing the leadership. The attack can continue infinitely, and delay termination. If the delay Δ is larger than the bounded communication delay after GST, it makes the leader l_2 observe the highest prepared ballot b, and adopt its value as the value of its candidate b'_2 (at L. 32). When it tries to commit b'_2, since it is compatible with b, it does not need abort it. Therefore, it can prepare and commit b'_2, and decide. We also note that instead of the delay Δ, the above attack can be avoided if the leader election can provide two successive well-behaved leaders.

▶ **Theorem 16.** *Quorum intersection and strong availability are sufficient to implement consensus.*

This theorem follows from three lemmas in the appendix [31] that prove that the protocol satisfies the specification of Byzantine consensus that we defined in Definition 11. An example execution of the protocol is described in the appendix [31].

7 Related Works

Quorum Systems with Heterogeneous Trust. Ripple [44] and Cobalt [35] pioneered decentralized trust. They let each node specify a list, called the unique node list (UNL), of processes that it trusts. However, they do not consider quorum availability or subsumption.

Stellar [38, 33] presents federated Byzantine quorum systems (FBQS) [24, 25] where quorums are iteratively calculated from quorums slices. Stellar also presents a federated voting and consensus protocol. In comparison, the assumptions of the protocols presented in this paper are weaker, and their guarantees are stronger. The stellar consensus protocol (SCP) guarantees termination when Byzantine processes stop. In contrast, the consensus protocol in this paper guarantees termination regardless of Byzantine processes. Further, abstract SCP [24] provides agreement only for intact processes. The intact set for an FBQS is a subset of processes that have strong availability. On the other hand, the consensus protocol in this paper provides agreement for all well-behaved processes. In FBQS, the intersections of quorums should have a process in the intact set; however, in HQS, they only need to have a well-behaved process. The validity and totality properties for the reliable broadcast for FBQS are restricted to the intact set. On the other hand, the reliable broadcast protocol in this paper provides totality for all processes that have weak availability, and validity for all processes that have strong availability.

Personal Byzantine quorum systems (PBQS) [34] capture the quorum systems that FBQSs derive form slices, and propose a responsiveness consensus protocol [48, 1, 43, 3]. They define a notion called quorum sharing which requires quorum subsumption for every quorum. Stellar quorums have quorum sharing if and only if processes do not lie about their slices. (The appendix [31] presents examples.) In this paper, we relax quorum sharing to quorum subsumption, and capture quorums that FBQSs derive even when Byzantine quorums lie about their slices, and show that even if a quorum system does not satisfy quorum sharing, safety can be maintained for all processes, and liveness can be maintained for the set of strongly available processes.

Asymmetric Byzantine quorum systems (ABQS) [15, 16, 4] allow each process to define a subjective dissemination quorum system (DQS), in a globally known system. The followup model [14] lets each process specify a subjective DQS for processes that it knows, transitively relying on the assumptions of other processes. In contrast, HQS lets each process specify its own set of quorums without knowing the quorums of other processes. Further, it does not require the specification of a set of possible Byzantine sets. Further, there are systems where a strongly available set (from HQS) exists but no guild set (from ABQS) exists. (The appendix [31] presents examples.) Therefore, HQS can provide safety and liveness for those executions but ABQS cannot. ABQS presents shared memory and broadcast protocols, and further, rules to compose two ABQSs. On the other hand, this paper proves impossibility results, and presents protocols for reliable broadcast and consensus abstractions. HQS provides strictly stronger guarantees with weaker assumptions. In ABQS, the properties of reliable broadcast are stated for wise processes and the guild. However, this paper states these four properties for well-behaved processes and the strongly available set. Well-behaved processes are a superset of wise processes, and as noted above, in certain executions, the strongly available set is a superset of the guild.

Flexible BFT [36] allows different failure thresholds between learners. Heterogeneous Paxos [45, 46] further generalizes the separation between learners and acceptors with different trust assumptions; it specifies quorums as sets rather than number of processes. These two projects introduce a consensus protocol that guarantees safety or liveness for learners with correct trust assumptions. However, they require the knowledge of all processes in the system. In contrast, HQS only requires partial knowledge of the system, and captures the properties of quorum systems where reliable broadcast and consensus protocols are impossible or possible. Multi-threshold reliable broadcast and consensus [27] and MT-BFT [40] elaborate Bracha [9] to have different fault thresholds for different properties, and different synchrony assumptions. However, they have cardinality-based or uniform quorums across processes. In contrast, HQS supports heterogeneous quorums.

K-consistent reliable broadcast (K-CRB) [7] introduces a relaxed reliable broadcast abstraction where the correct processes can define their own quorum systems. Given a quorum system, it focuses on delivering the smallest number k of different values. In contrast, we propose the weakest condition to solve classical reliable broadcast and consensus. Moreover, K-CRB's relaxed liveness guarantee (accountability) requires public key infrastructure. In contrast, all the results in this paper are for information-theoretic setting.

Our consensus protocol uses eventual leader election. Several other works present view synchronization and eventual leader election for Byzantine replicated systems [11, 10], and dynamic networks [41, 28]. It is interesting to see if their leader election modules can be generalized to the heterogeneous setting, and support responsiveness [48, 5] for our consensus protocol.

Impossibility Results. There are two categories of assumptions about the computational power of Byzantine processes. In the information-theoretic setting, Byzantine process have unlimited computational resources. While in the computational setting, Byzantine processes can not break a polynomial-time bound [23]. In this work, our impossibility results for reliable broadcast and consensus fall in the information-theoretic category. Whether the same results hold in the computational setting is an interesting open question.

FLP [22] proved that consensus is not solvable in asynchronous networks even with one crash failure. Many following works [26, 19, 2, 21, 30, 8] considered solvability, and necessary and sufficient conditions for consensus and reliable broadcast to tolerate f Byzantine failures in partially synchronous networks. The number of processes should be more than $3f$ and the connectivity of the communication graph should be more than $2f$. However, these results apply for cardinality-based quorums, which is a special instance of HQS. We generalize the reliable broadcast and consensus abstractions to HQS which supports non-uniform quorums, and prove impossibility results for them.

8 Conclusion

This paper presented a general model of heterogeneous quorum systems where each process defines its own set of quorums, and captured their properties. Through indistinguishably arguments, it proved that no deterministic quorum-based protocol can implement the consensus and Byzantine reliable broadcast abstractions on a heterogeneous quorum system that provides only quorum intersection and availability. It introduced the quorum subsumption property, and showed that the three conditions together are sufficient to implement the two abstractions. It presented Byzantine broadcast and consensus protocols for heterogeneous quorum systems, and proved their correctness when the underlying quorum system maintain the three properties.

References

1 Ittai Abraham and Gilad Stern. Information theoretic hotstuff. *arXiv preprint arXiv:2009.12828*, 2020.

2 Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Consensus with byzantine failures and little system synchrony. In *International Conference on Dependable Systems and Networks (DSN'06)*, pages 147–155. IEEE, 2006.

3 Dan Alistarh, James Aspnes, Faith Ellen, Rati Gelashvili, and Leqi Zhu. Why extension-based proofs fail. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, pages 986–996, 2019.

4 Orestis Alpos, Christian Cachin, and Luca Zanolini. How to trust strangers: Composition of byzantine quorum systems. In *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*, pages 120–131. IEEE, 2021.

5 Hagit Attiya, Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Bounds on the time to reach agreement in the presence of timing uncertainty. *Journal of the ACM (JACM)*, 41(1):122–152, 1994.

6 Mathieu Baudet, Avery Ching, Andrey Chursin, George Danezis, Francois Garillot, Zekun Li, Dahlia Malkhi, Oded Naor, Dmitri Perelman, and Alberto Sonnino. State machine replication in the libra blockchain. *The Libra Assn., Tech. Rep*, 7, 2019.

7 João Paulo Bezerra, Petr Kuznetsov, and Alice Koroleva. Relaxed reliable broadcast for decentralized trust. In *Networked Systems: 10th International Conference, NETYS 2022, Virtual Event, May 17–19, 2022, Proceedings*, pages 104–118. Springer, 2022.

8 Malte Borcherding. Levels of authentication in distributed agreement. In *International Workshop on Distributed Algorithms*, pages 40–55. Springer, 1996.

9 Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, 32(4):824–840, 1985.

10 Manuel Bravo, Gregory Chockler, and Alexey Gotsman. Liveness and latency of byzantine state-machine replication. In *36th International Symposium on Distributed Computing (DISC 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

11 Manuel Bravo, Gregory Chockler, and Alexey Gotsman. Making byzantine consensus live. *Distributed Computing*, 35(6):503–532, 2022.

12 Ethan Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains*. PhD thesis, University of Guelph, 2016.

13 Ethan Buchman, Rachid Guerraoui, Jovan Komatovic, Zarko Milosevic, Dragos-Adrian Seredinschi, and Josef Widder. Revisiting tendermint: Design tradeoffs, accountability, and practical use. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S)*, pages 11–14. IEEE, 2022.

14 Christian Cachin, Giuliano Losa, and Luca Zanolini. Quorum systems in permissionless network. *arXiv preprint arXiv:2211.05630*, 2022.

15 Christian Cachin and Björn Tackmann. Asymmetric distributed trust. In *23rd International Conference on Principles of Distributed Systems (OPODIS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

16 Christian Cachin and Luca Zanolini. From symmetric to asymmetric asynchronous byzantine consensus. *arXiv preprint arXiv:2005.08795*, 2020.

17 Harold Carr, Christa Jenkins, Mark Moir, Victor Cacciari Miraldo, and Lisandra Silva. Towards formal verification of hotstuff-based byzantine fault tolerant consensus in agda. In *NASA Formal Methods: 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24–27, 2022, Proceedings*, pages 616–635. Springer, 2022.

18 Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

19 Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, Vassos Hadzilacos, Petr Kouznetsov, and Sam Toueg. The weakest failure detectors to solve certain fundamental problems in distributed computing. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 338–346, 2004.

20 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

21 Michael J Fischer, Nancy A Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1(1):26–39, 1986.

22 Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

23 Juan Garay and Aggelos Kiayias. Sok: A consensus taxonomy in the blockchain era. In *Cryptographers' track at the RSA conference*, pages 284–318. Springer, 2020.

24 Álvaro García-Pérez and Alexey Gotsman. Federated byzantine quorum systems. In *22nd International Conference on Principles of Distributed Systems (OPODIS 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

25 Álvaro García-Pérez and Maria A Schett. Deconstructing stellar consensus (extended version). *arXiv preprint arXiv:1911.05145*, 2019.

26 Guy Goren, Yoram Moses, and Alexander Spiegelman. Probabilistic indistinguishability and the quality of validity in byzantine agreement. *arXiv preprint arXiv:2011.04719*, 2020.

27 Martin Hirt, Ard Kastrati, and Chen-Da Liu-Zhang. Multi-threshold asynchronous reliable broadcast and consensus. *Cryptology ePrint Archive*, 2020.

28 Rebecca Ingram, Patrick Shields, Jennifer E Walter, and Jennifer L Welch. An asynchronous leader election algorithm for dynamic networks. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12. IEEE, 2009.

29 Leslie Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19:104–125, 2006.

30 Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, pages 382–401, 1982.

31 Xiao Li, Eric Chan, and Mohsen Lesani. Quorum subsumption for heterogeneous quorum systems. technical report. In *International Symposium on Distributed Computing (DISC 2023)*, 2023.

32 Xiao Li and Mohsen Lesani. Open heterogeneous quorum systems, 2023. `arXiv:2304.02156`.

33 Marta Lokhava, Giuliano Losa, David Mazières, Graydon Hoare, Nicolas Barry, Eli Gafni, Jonathan Jove, Rafał Malinowsky, and Jed McCaleb. Fast and secure global payments with stellar. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 80–96, 2019.

34 Giuliano Losa, Eli Gafni, and David Mazières. Stellar consensus by instantiation. In *33rd International Symposium on Distributed Computing (DISC 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

35 Ethan MacBrough. Cobalt: Bft governance in open networks. *arXiv preprint arXiv:1802.07240*, 2018.

36 Dahlia Malkhi, Kartik Nayak, and Ling Ren. Flexible byzantine fault tolerance. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, pages 1041–1053, 2019.

37 Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed computing*, 11(4):203–213, 1998.

38 David Mazieres. The stellar consensus protocol: A federated model for internet-level consensus. *Stellar Development Foundation*, 32:1–45, 2015.

39 Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 31–42, 2016.

40 Atsuki Momose and Ling Ren. Multi-threshold byzantine fault tolerance. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1686–1699, 2021.

41 Achour Mostefaoui, Michel Raynal, Corentin Travers, Stacy Patterson, Divyakant Agrawal, and Amr EL Abbadi. From static distributed systems to dynamic systems. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, pages 109–118. IEEE, 2005.

42 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *White paper*, 2008.

43 Rafael Pass and Elaine Shi. Thunderella: Blockchains with optimistic instant confirmation. In *Advances in Cryptology–EUROCRYPT 2018: 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29-May 3, 2018 Proceedings, Part II 37*, pages 3–33. Springer, 2018.

44 David Schwartz, Noah Youngs, and Arthur Britto. The ripple protocol consensus algorithm. *Ripple Labs Inc White Paper*, 5(8):151, 2014.

45 Isaac Sheff, Xinwen Wang, Robbert van Renesse, and Andrew C Myers. Heterogeneous paxos. In *OPODIS: International Conference on Principles of Distributed Systems*, number 2020 in OPODIS, 2021.

46 Isaac C Sheff, Robbert van Renesse, and Andrew C Myers. Distributed protocols and heterogeneous trust: Technical report. *arXiv preprint arXiv:1412.3136*, 2014.

47 Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. Efficient byzantine fault-tolerance. *IEEE Transactions on Computers*, 62(1):16–30, 2011.

48 Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.

Fast Deterministic Rendezvous in Labeled Lines

Avery Miller
University of Manitoba, Winnipeg, Canada

Andrzej Pelc
Université du Québec en Outaouais, Canada

Abstract

Two mobile agents, starting from different nodes of a network modeled as a graph, and woken up at possibly different times, have to meet at the same node. This problem is known as *rendezvous*. Agents move in synchronous rounds. In each round, an agent can either stay idle or move to an adjacent node. We consider deterministic rendezvous in the infinite line, i.e., the infinite graph with all nodes of degree 2. Each node has a distinct label which is a positive integer. An agent currently located at a node can see its label and both ports 0 and 1 at the node. The time of rendezvous is the number of rounds until meeting, counted from the starting round of the earlier agent.

We consider three scenarios. In the first scenario, each agent knows its position in the line, i.e., each of them knows its initial distance from the smallest-labeled node, on which side of this node it is located, and the direction towards it. For this scenario, we design a rendezvous algorithm working in time $O(D)$, where D is the initial distance between the agents. This complexity is clearly optimal. In the second scenario, each agent knows *a priori* only the label of its starting node and the initial distance D between them. In this scenario, we design a rendezvous algorithm working in time $O(D \log^* \ell)$, where ℓ is the larger label of the starting nodes. We also prove a matching lower bound $\Omega(D \log^* \ell)$. Finally, in the most general scenario, where each agent knows *a priori* only the label of its starting node, we design a rendezvous algorithm working in time $O(D^2(\log^* \ell)^3)$, which is thus at most cubic in the lower bound. All our results remain valid (with small changes) for arbitrary finite lines and for cycles. Our algorithms are drastically better than approaches that use graph exploration, which have running times that depend on the size or diameter of the graph.

Our main methodological tool, and the main novelty of the paper, is a two way reduction: from fast colouring of the infinite labeled line using a constant number of colours in the $\mathcal{LOCAL}$ model to fast rendezvous in this line, and vice-versa. In one direction we use fast node colouring to quickly break symmetry between the identical agents. In the other direction, a lower bound on colouring time implies a lower bound on the time of breaking symmetry between the agents, and hence a lower bound on their meeting time.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases rendezvous, deterministic algorithm, mobile agent, labeled line, graph colouring

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.29

Funding *Avery Miller*: Supported by NSERC Discovery Grant 2017-05936.
Andrzej Pelc: Partially supported by NSERC Discovery Grant 2018-03899 and by the Research Chair in Distributed Computing at the Université du Québec en Outaouais.

1 Introduction

Background. Two mobile agents, starting from different nodes of a network modeled as a graph, and woken up at possibly different times, have to meet at the same node. This problem is known as *rendezvous*. The autonomous mobile entities (agents) may be natural, such as people who want to meet in a city whose streets form a network. They may also represent human-made objects, such as software agents in computer networks or mobile robots navigating in a network of corridors in a mine or a building. Agents might want to meet to share previously collected data or to coordinate future network maintenance tasks.

Model and problem description. We consider deterministic rendezvous in the infinite line, i.e., the infinite graph with all nodes of degree 2. Each node has a distinct label which is a positive integer. An agent currently located at a node can see its label and both ports 0 and 1 at the node. Technically, agents are anonymous, but each agent could adopt the label of its starting node as its label. Agents have synchronized clocks ticking at the same rate, measuring *rounds*. Agents are woken up by an adversary in possibly different rounds. The clock of each agent starts in its wakeup round. In each round, an agent can either stay idle or move to an adjacent node. After moving, an agent knows on which port number it arrived at its current node. When agents cross, i.e., traverse an edge simultaneously in different directions, they do not notice this fact. No limitation is imposed on the memory of the agents. Computationally, they are modeled as Turing Machines. The time of rendezvous is the number of rounds until meeting, counted from the starting round of the earlier agent.

In most of the literature concerning rendezvous, nodes are assumed to be anonymous. The usual justification of this weak assumption is that in labeled graphs (i.e., graphs whose nodes are assigned distinct positive integers) each agent can explore the entire graph and then meet the other one at the node with the smallest label. Thus, the rendezvous problem in labeled graphs can be reduced to exploration. While this simple strategy is correct for finite graphs, it requires time at least equal to the size of the graph, and hence very inefficient for large finite graphs, even if the agents start at a very small initial distance. In infinite graphs, this strategy is incorrect: indeed, if label 1 is not used in some labeling, an agent may never learn what is the smallest used label. For large environments that would be impractical to exhaustively search, it is desirable and natural to relate rendezvous time to the initial distance between the agents, rather than to the size of the graph or the value of the largest node label. This motivates our choice to study the infinite labeled line: we must avoid algorithms that depend on exhaustive search, and there cannot be an efficient algorithm that is dependent on a global network property such as size, largest node label, etc. In particular, the initial distance between the agents and the labels of encountered nodes should be the only parameters available to measure the efficiency of algorithms. Results for the infinite line can then be applied to understand what's possible in large path-like environments.

We consider three scenarios, depending on the amount of knowledge available *a priori* to the agents. In the first scenario, each agent knows its position in the line, i.e., each of them knows its distance from the smallest-labeled node, on which side of this node it is located, and the direction towards it. This scenario is equivalent to assuming that the labeling is very particular, such that both above pieces of information can be deduced by the agent by inspecting the label of its starting node and port numbers at it. One example of such a labeling is $...8, 6, 4, 2, 1, 3, 5, 7, ...$ (recall that labels must be positive integers), with the following port numbering. For nodes with odd labels, port 1 always points to the neighbour with larger label and port 0 points to the neighbour with smaller label, while for nodes with even label, port 1 always points to the neighbour with smaller label and port 0 points to the neighbour with larger label. We will call the line with the above node labeling and port numbering the *canonical line*, and use, for any node, the term "right of the node" for the direction corresponding to port 1 and the term "left of the node" for the direction corresponding to port 0. In the second scenario that we consider, the labeling and port numbering are arbitrary and each agent knows *a priori* only the label of its starting node and the initial distance D between the agents. Finally, in the third scenario, the labeling and port numbering are arbitrary and each agent knows *a priori* only the label of its starting node.

Our results. We start with the scenario of the canonical line. This scenario was previously considered in [13], where the authors give a rendezvous algorithm with optimal complexity $O(D)$, where D is the (arbitrary) initial distance between the agents. However, they use the strong assumption that both agents are woken up in the same round. In our first result we get rid of this assumption: we design a rendezvous algorithm with complexity $O(D)$, regardless of the delay between the wakeup of the agents. This complexity is clearly optimal.

The remaining results are expressed in terms of the *base-2 iterated logarithm* function, denoted by $\log^*$. The value of $\log^*(n)$ is 0 if $n \leq 1$, and otherwise its value is $1 + \log^*(\log_2 n)$. This function grows extremely slowly with respect to n, e.g., $\log^*(2^{65536}) = 5$.

For the second scenario (arbitrary labeling with known initial distance D) we design a rendezvous algorithm working in time $O(D \log^* \ell)$, where ℓ is the larger label of the two starting nodes. We also prove a matching lower bound $\Omega(D \log^* \ell)$, by constructing an infinite labeled line in which this time is needed, even if D is known and if agents start simultaneously. As a corollary, we get the following impossibility result: for any deterministic algorithm and for any finite time T, there exists an infinite labeled line such that two agents starting from some pair of adjacent nodes cannot meet in time T.

Finally, for the most general scenario, where each agent knows *a priori* only the label of its starting node, we design a rendezvous algorithm working in time $O(D^2(\log^* \ell)^3)$, for arbitrary unknown D. This complexity is thus at most cubic in the above lower bound that holds even if D is known.

It should be stressed that the complexities of our algorithms in the second and third scenarios depend on D and on the larger label of the two starting nodes. No algorithm whose complexity depends on D and on the maximum label in even a small vicinity of the starting nodes would be satisfactory, since neighbouring nodes can have labels differing in a dramatic way, e.g., consider two agents that start at adjacent nodes, and these nodes are labeled with small integers like 2 and 3, but other node labels in their neighbourhoods are extremely large. Our approach demonstrates that this is an instance that can be solved very quickly, but an approach whose running time depends on the labels of other nodes in the neighbourhood can result in arbitrarily large running times (which are very far away from the lower bound).

An alternative way of looking at the above three scenarios is the following. In the first scenario, the agent is given *a priori* the entire (ordered) labeling of the line. Of course, since this is an infinite object, the labeling cannot be given to the agent as a whole, but the agent can *a priori* get the answer to any question about it, in our case answers to two simple questions: how far is the smallest-labeled node and in which direction. In the second and third scenarios, the agent gets information about the label of a given node only when visiting it. It is instructive to consider another, even weaker, scenario: the agent learns the label of its initial node but the other labels are never revealed to it. This weak scenario is equivalent to the scenario of labeled agents operating in an anonymous line: the labels of the agents are guaranteed to be different integers (they are the labels of the starting nodes) but all other nodes look the same. It follows from [16] that in this scenario the optimal time of rendezvous is $\Theta(D \log \ell)$, where ℓ is, as usual, the larger label of the starting nodes.[1] Hence, at least for known D, we get strict separations between optimal rendezvous complexities, according to three ways of getting knowledge about the labeling of the line: $\Theta(D)$, if all knowledge is given at once, $\Theta(D \log^* \ell)$ if knowledge about the labeling is given as the agents visit new nodes, and $\Theta(D \log \ell)$, if no knowledge about the labeling is ever given to the agents, apart

[1] In [16] cycles are considered instead of the infinite line, and the model and result are slightly different but obtaining, as a corollary, this complexity in our model is straightforward.

from the label of the starting node. Of course, in the ultimately weak scenario where even the label of the starting node is hidden from the agents, deterministic rendezvous would become impossible: there is no deterministic algorithm guaranteeing rendezvous of anonymous agents in an anonymous infinite line, even if they start simultaneously from adjacent nodes.

It should be mentioned that the $O(D \log \ell)$-time rendezvous algorithm from [16] is also valid in our scenario of arbitrary labeled lines, with labels of nodes visible to the agents at their visit and with unknown D: the labels of the visited nodes can be simply ignored by the algorithm from [16]. This complexity is incomparable with our complexity $O(D^2(\log^* \ell)^3)$. For bounded D our algorithm is much faster but for large D compared to ℓ it is less efficient.

As usual in models where agents cannot detect crossing on the same edge when moving in opposite directions, we guarantee rendezvous by creating time intervals in which one agent is idle at its starting node and the other searches a sufficiently large neighbourhood to include this node. Since the adversary can choose the starting rounds of the agents, it is difficult to organize these time intervals to satisfy the above requirement. Our main methodological tool, and the main novelty of the paper is a two way reduction: from fast colouring of the infinite labeled line using a constant number of colours in the $\mathcal{LOCAL}$ model to fast rendezvous in this line, and vice-versa. In one direction we use fast node colouring to quickly break symmetry between the identical agents. In the other direction, a lower bound on colouring time implies a lower bound on the time of breaking symmetry between the agents, and hence a lower bound on their meeting time.

As part of our approach to solve rendezvous using node colouring, we provide a result (based on the idea of Cole and Vishkin [12]) that might be of independent interest: for the $\mathcal{LOCAL}$ model, we give a deterministic distributed algorithm EarlyStopCV that properly 3-colours any infinite labeled line such that the execution of EarlyStopCV at any node x with initial label ID_x terminates in time $O(\log^*(\mathrm{ID}_x))$, and the algorithm does not require that the nodes have any initial knowledge about the network other than their own label.

All our results remain valid for finite lines and cycles: in the first scenario without any change, and in the two other scenarios with small changes. As previously mentioned, it is always possible to meet in a labeled line or cycle of size n in time $O(n)$ by exploring the entire graph and going to the smallest-labeled node. Thus, in the second scenario the upper and lower bounds on complexity change to $O(\min(n, D \log^* \ell))$ and $\Omega(\min(n, D \log^* \ell))$, respectively, and in the third, most general scenario, the complexity of (a slightly modified version of) our algorithm changes to $O(\min(n, D^2(\log^* \ell)^3))$.

Related work. Rendezvous has been studied both under randomized and deterministic scenarios. A survey of randomized rendezvous under various models can be found in [3], cf. also [1, 2, 4, 7]. Deterministic rendezvous in networks has been surveyed in [27, 28]. Many authors considered geometric scenarios (rendezvous in the real line, e.g., [7, 8], or in the plane, e.g., [5, 6, 11, 14]). Gathering more than two agents was studied, e.g., in [18, 24, 31].

In the deterministic setting, many authors studied the feasibility and time complexity of synchronous rendezvous in networks. For example, deterministic rendezvous of agents equipped with tokens used to mark nodes was considered, e.g., in [22]. In most of the papers concerning rendezvous in networks, nodes of the network are assumed to be unlabeled and agents are not allowed to mark the nodes in any way. In this case, the symmetry is usually broken by assigning the agents distinct labels and assuming that each agent knows its own label but not the label of the other agent. Deterministic rendezvous of labeled agents in rings was investigated, e.g., in [16, 20] and in arbitrary graphs in [16, 20, 30]. In [16], the authors present a rendezvous algorithm whose running time is polynomial in the size of the graph, in the length of the shorter label and in the delay between the starting times of the agents. In [20, 30], rendezvous time is polynomial in the first two of these parameters and independent of the delay. In [9, 10] rendezvous was considered, respectively, in unlabeled infinite trees, and in arbitrary connected unlabeled infinite graphs, but the complexities depended, among others, on the logarithm of the size of the label space. Gathering many anonymous agents in unlabeled networks was investigated in [17]. In this weak scenario, not all initial configurations of agents are possible to gather, and the authors of [17] characterized all such configurations and provided universal gathering algorithms for them. On the other hand, time of rendezvous in labeled networks was studied, e.g., in [26], in the context of algorithms with advice. In [13], the authors studied rendezvous under a very strong assumption that each agent has a map of the network and knows its position in it.

Memory required by the agents to achieve deterministic rendezvous was studied in [19] for trees and in [15] for general graphs. Memory needed for randomized rendezvous in the ring was discussed, e.g., in [21].

Roadmap. In Section 2, we state two results concerning fast colouring of labeled lines. In Section 3, we present our optimal rendezvous algorithm for the canonical line. In Section 4, we present our optimal rendezvous algorithm for arbitrary labeled lines with known initial distance between the agents, we analyze the algorithm, and we sketch the proof of the matching lower bound. In Section 5, we give our rendezvous algorithm for arbitrary labeled lines with unknown initial distance between the agents. In Section 6, we conclude the paper and present some open problems. In the appendix, we describe a fast colouring algorithm for the infinite line, as announced in Section 2.

Due to lack of space, the analysis of the algorithms for the canonical line and for arbitrary labeled lines with unknown initial distance between the agents, as well as detailed proofs of all results concerning the lower bound from Section 4, are omitted and will appear in the journal version of the paper.

2 Tools

We will use two results concerning distributed node-colouring of lines and cycles in the $\mathcal{LOCAL}$ communication model [29]. In this model, communication proceeds in synchronous rounds and all nodes start simultaneously. In each round, each node can exchange arbitrary messages with all of its neighbours and perform arbitrary local computations. Recall that, in the problem of k-colouring of a graph, we start with a graph whose nodes are labeled with distinct labels, each node knows only its own label, and at the end each node has to adopt one of k colours in such a way that adjacent nodes have different colours.

The first result is based on the 3-colouring algorithm of Cole and Vishkin [12], but improves on their result in two ways. First, our algorithm does not require that the nodes know the size of the network or the largest label in the network. Second, the running time of our algorithm at any node x with initial label ID_x is $O(\log^*(\mathrm{ID}_x))$. The running-time guarantee is vital for later results in this paper, and it is not provided by the original $O(\log^*(n))$ algorithm of Cole and Vishkin since the correctness of their algorithm depends on the fact that all nodes execute the algorithm for the same number of rounds. Our algorithm from Proposition 1, called EarlyStopCV, is described in the appendix.

▶ **Proposition 1.** *There exists an integer constant $\kappa > 1$ and a deterministic distributed algorithm EarlyStopCV such that, for any infinite line G with nodes labeled with distinct integers greater than 1, EarlyStopCV 3-colours G and the execution at any node x with initial label ID_x terminates in time at most $\kappa \log^*(\mathrm{ID}_x)$.*

The second result is a lower bound due to Linial [23, 25]. Note that Linial's original result was formulated for cycles labeled with integers in the range $1, \ldots, n$, but the simplified proof in [23] can be adapted to hold in our formulation below.

▶ **Proposition 2.** *Fix any positive integer n and any set $\mathcal{I}$ of n integers. For any deterministic algorithm $\mathcal{A}$ that 3-colours any path on n nodes with distinct labels from $\mathcal{I}$, there is such a path and at least one node for which algorithm $\mathcal{A}$'s execution takes time at least $\frac{1}{2}\log^*(n) - 1$.*

Finally, we highlight an important connection between the agent-based computational model considered in this paper (where algorithm executions may start in different rounds) and the node-based $\mathcal{LOCAL}$ model (where all algorithm executions start in the same round). Consider a fixed network G in which the nodes are labeled with fixed distinct integers. From the communication constraints imposed by the $\mathcal{LOCAL}$ model, for any deterministic algorithm $\mathcal{A}$, each node v's behaviour in the first i rounds is completely determined by the labeled $(i-1)$-neighbourhood of v in G (i.e., the induced labeled subgraph of G consisting of all nodes within distance $i-1$ from v). By Proposition 1, we know that each node x executing EarlyStopCV in G determines its final colour within $\kappa \log^*(\mathrm{ID}_x)$ rounds, so its colour is completely determined by its labeled $(\kappa \log^*(\mathrm{ID}_x) - 1)$-neighbourhood in G. So, in the agent-based computational model, an agent operating in the same labeled network G with knowledge of EarlyStopCV and the value of κ from Proposition 1 could, starting in any round: visit all nodes within a distance $\kappa \log^*(\mathrm{ID}_x) - 1$ from x, then simulate in its local memory (in one round) the behaviour of x in the first $\kappa \log^*(\mathrm{ID}_x)$ rounds of EarlyStopCV in the $\mathcal{LOCAL}$ model, and thus determine the colour that would be chosen by node x as if all nodes in G had executed EarlyStopCV in the $\mathcal{LOCAL}$ model starting in round 1.

▶ **Proposition 3.** *Consider a fixed infinite line G such that the nodes are labeled with distinct integers greater than 1. Consider any two nodes labeled v_α and v_β in G. Suppose that all nodes in G execute algorithm EarlyStopCV in the $\mathcal{LOCAL}$ model starting in round 1, and let c_α and c_β be the colours that nodes v_α and v_β, respectively, output at the end of their executions. Next, consider two agents α and β that start their executions at nodes labeled v_α and v_β in G, respectively (and possibly in different rounds). Suppose that there is a round r_α in which agent α knows the $(\kappa \log^*(v_\alpha))$-neighbourhood of node v_α in G, and suppose that there is a round r_β in which agent β knows the $(\kappa \log^*(v_\beta))$-neighbourhood of node v_β in G (and note that we may have $r_\alpha \neq r_\beta$). Then, agent α can compute c_α in round r_α, and, agent β can compute c_β in round r_β.*

3 The canonical line

In this section, we describe an algorithm $\mathcal{A}_{\text{rv-canon}}$ that solves rendezvous on the canonical line in time $O(D)$ when two agents start at arbitrary positions and the delay between the rounds in which they start executing the algorithm is arbitrary. The agents do not know the initial distance D between them, and do not know the delay between the starting rounds.

Denote by α and β the two agents. Denote by v_α and v_β the starting nodes of α and β, respectively. Denote by $\mathcal{O}$ the node on the canonical line with the smallest label. For $a \in \{\alpha, \beta\}$, we will write $d(v_a, \mathcal{O})$ to denote the distance between v_a and $\mathcal{O}$.

Algorithm $\mathcal{A}_{\text{rv-canon}}$ proceeds in phases, numbered starting at 0. Each phase's description has two main components. The first component is a colouring of all nodes on the line. At a high level, in each phase $i \geq 0$, the line is partitioned into segments consisting of 2^i consecutive nodes each, and the set of segments is properly coloured, i.e., all nodes in the same segment get the same colour, and two neighbouring nodes in different segments get different colours. The second component describes how an agent behaves when executing the phase. At a high level, the phase consists of equal-sized blocks of rounds, and in each block, an agent either stays idle at its starting node for all rounds in the block, or, it spends the block performing a search of nearby nodes in an attempt to find the other agent (and if not successful, returns back to its starting node). Whether an agent idles or searches in a particular block of the phase depends on the colour of its starting node. The overall idea is: there exists a phase in which the starting nodes of the agents will be coloured differently, and this will result in one agent idling while the other searches, which will result in rendezvous.

The above intuition overlooks two main difficulties. The first difficulty is that the agents do not know the initial distance between them, so they do not know how far they have to search when trying to find the other agent. To deal with this issue, the agents will use larger and larger "guesses" in each subsequent phase of the algorithm, and eventually the radius of their search will be large enough. The second difficulty is that the agents do not necessarily start the algorithm in the same round, so the agents' executions of the algorithm (i.e., the phases and blocks) can misalign in arbitrary ways. This makes it difficult to ensure that there is a large enough set of contiguous rounds during which one agent remains idle while the other agent searches. To deal with this issue, we carefully choose the sizes of blocks and phases, as well as the type of behaviour (idle vs. search) carried out in each block.

We now give the full details of an arbitrary phase $i \geq 0$ in the algorithm's execution.

Colouring. When an agent starts executing phase i, it first determines the colour of its starting node. From a global perspective, the idea is to assign colours to nodes on the infinite line in the following way:

1. Partition the nodes into segments consisting of 2^i nodes each, such that node $\mathcal{O}$ is the leftmost node of its segment. Denote the segment containing $\mathcal{O}$ by S_0, denote each subsequent segment to its right using increasing indices (i.e., $S_1, S_2, \ldots$) and denote each subsequent segment to its left using decreasing indices (i.e., $S_{-1}, S_{-2}, \ldots$).
2. For each segment with even index, colour all nodes in the segment with "red". For each segment with odd index, colour all nodes in the segment with "blue". As a result, neighbouring segments are always assigned different colours.

However, the agent executing phase i does not compute this colouring for the entire line, it need only determine the colour of its starting node. To do so, it uses its knowledge about the distance and direction from its starting node to node $\mathcal{O}$. In particular,

- if the agent's starting node s is $\mathcal{O}$ or to the right of $\mathcal{O}$, it computes the index of the segment in which s is contained, i.e., $\mathit{index} = \lfloor d(s, \mathcal{O})/2^i \rfloor$. If index is even, then s has colour red, and otherwise s has colour blue.
- if the agent's starting node s is to the left of $\mathcal{O}$, it computes the index of the segment in which s is contained, i.e., $\mathit{index} = -\lceil (d(s, \mathcal{O}))/2^i \rceil$. If index is even, then s has colour red, and otherwise s has colour blue.

Behaviour. Phase i consists of $44 \cdot 2^{i+1}$ rounds, partitioned into 11 equal-sized blocks of $4 \cdot 2^{i+1}$ rounds each. The number 2^{i+1} has a special significance: it is the search radius used by an agent during phase i whenever it is searching for the other agent. We use the notation $\mathit{SR}(i)$ to represent the value 2^{i+1} in the remainder of the algorithm's description and analysis. In each of the 11 blocks of the phase, the agent behaves in one of two ways: if a block is designated as a *waiting block*, the agent stays at its starting node v for all $4 \cdot \mathit{SR}(i)$ rounds of the block; otherwise, a block is designated as a *searching block*, in which the agent moves right $\mathit{SR}(i)$ times, then left $2 \cdot \mathit{SR}(i)$ times, then right $\mathit{SR}(i)$ times. In other words, during a searching block, the agent explores all nodes within its immediate $SR(i)$-neighbourhood, and ends up back at its starting node. Whether a block is designated as "waiting" or "searching" depends on the agent's starting node colour in phase i. In particular, if its starting node is red, then blocks 1,8,9 are searching blocks and all others are waiting blocks; otherwise, if its starting node is blue, then blocks 1,10,11 are searching blocks and all others are waiting blocks. This concludes the description of $\mathcal{A}_{\text{rv-canon}}$.

4 Arbitrary lines with known initial distance between agents

4.1 The algorithm

In this section, we describe an algorithm called $\mathcal{A}_{\text{rv-D}}$ that solves rendezvous on lines with arbitrary node labelings when two agents start at arbitrary positions and when the delay between the rounds in which they start executing the algorithm is arbitrary. The algorithm works in time $O(D \log^* \ell)$, where D is the initial distance between the agents, and ℓ is the larger label of the two starting nodes. The agents know D, but they do not know the delay between the starting rounds. Also, we note that the agents have no global sense of direction, but each agent can locally choose port 0 from its starting node to represent "right" and port 1 from its starting node to represent "left". Further, using knowledge of the port number of the edge on which it arrived at a node, an agent is able to choose whether its next move will continue in the same direction or if it will switch directions. Without loss of generality, we may assume that all node labels are strictly greater than one, since the algorithm could be re-written to add one to each label value in its own memory before carrying out any computations involving the labels. This assumption ensures that, for any node label v, the value of $\log^*(v)$ is strictly greater than 0.

Algorithm $\mathcal{A}_{\text{rv-D}}$ proceeds in two stages. In the first stage, the agents assign colours to their starting nodes according to a proper colouring of the nodes that are integer multiples of D away. They each accomplish this by determining the node labels within a sufficiently large neighbourhood and then executing the 3-colouring algorithm EarlyStopCV from Proposition 1 on nodes that are multiples of D away from their starting node. The second stage consists of repeated periods, with each period consisting of equal-sized blocks of rounds. In each block, an agent either stays idle at its starting node, or, it spends the block performing a search of nearby nodes in an attempt to find the other agent (and if not successful, returns back to its starting node). Whether or not an agent idles or searches in a particular block of the period depends on the colour it picked for its starting node. The overall idea behind the algorithm's correctness is: the agents are guaranteed to pick different colours for their starting node in the first stage, and so, when both agents are executing the second stage, one agent will search while the other idles, which will result in rendezvous.

Stage 1: Colouring. Let v_x be the starting node of agent x. We identify the node with its label. Let $r = D \cdot \kappa \log^*(v_x)$, where $\kappa > 1$ is the constant defined in the running time of the algorithm EarlyStopCV from Proposition 1. Denote by $\mathcal{B}_r$ the r-neighbourhood of v_x. First, agent x determines $\mathcal{B}_r$ (including all node labels) by moving right r times, then left $2r$ times, then right r times, ending back at its starting node v_x. Let V be the subset of nodes in $\mathcal{B}_r$ whose distance from v_x is an integer multiple of D. In its local memory, agent x creates a path graph G_x consisting of the nodes in V, with two nodes connected by an edge if and only if their distance in $\mathcal{B}_r$ is exactly D. This forms a path graph centered at v_x with $\kappa \log^*(v_x)$ nodes in each direction. Next, the agent simulates an execution of the algorithm EarlyStopCV by the nodes of G_x to assign a colour $c_x \in \{0, 1, 2\}$ to its starting node v_x.

Stage 2: Search. The agent repeatedly executes periods consisting of $8D$ rounds each, partitioned into two equal-sized blocks of $4D$ rounds each. In each of the two blocks, the agent behaves in one of two ways: if a block is designated as a *waiting block*, then the agent stays at its starting node for all $4D$ rounds; otherwise, a block is designated as a *searching block*, in which the agent moves right D times, then left $2D$ times, then right D times. Whether a block is designated as a "waiting" or "searching" block depends on the agent's starting node colour that was determined in the first stage. In particular, if $c_x = 0$, then both blocks are waiting blocks; if $c_x = 1$, then block 1 is a searching block and block 2 is a waiting block; and, if $c_x = 2$, then both blocks are searching blocks.

4.2 Analysis of the algorithm

In this section, we prove that Algorithm $\mathcal{A}_{\text{rv-D}}$ solves rendezvous within $O(D \log^* \ell)$ rounds, where ℓ is the larger of the labels of the two starting nodes of the agents.

Consider an arbitrary instance on some line L with an arbitrary labeling of the nodes with positive integers. Suppose that two agents α and β execute the algorithm. For each $x \in \{\alpha, \beta\}$, we denote by v_x the label of agent x's starting node, and we denote by c_x the colour assigned to node v_x at the end of Stage 1 in x's execution of the algorithm. To help with the wording of the analysis only, fix a global orientation for L so that v_α appears to the "left" of v_β (and recall that the agents have no access to this information).

First, we argue that after both agents have finished Stage 1 of their executions, they have assigned different colours to their starting nodes.

▶ **Lemma 4.** *In any execution of $\mathcal{A}_{rv\text{-}D}$ by agents α and β, in every round after both agents finish their execution of Stage 1, we have $c_\alpha \neq c_\beta$.*

Proof. Without loss of generality, assume that $v_\alpha > v_\beta$. Let y_0 be the node in L to the left of v_α at distance exactly $D \cdot \kappa \log^*(v_\alpha)$. For each $i \in \{1, \ldots, 2\kappa \log^*(v_\alpha) + 1\}$, let y_i be the node in L at distance $i \cdot D$ to the right of y_0. Note that $v_\alpha = y_{\kappa \log^*(v_\alpha)}$ and $v_\beta = y_{\kappa \log^*(v_\alpha)+1}$.

Create a path graph P consisting of $2\kappa \log^*(v_\alpha) + 2$ nodes. Label the leftmost node in P with y_0, and label the node at distance i from y_0 in P using the label y_i.

By the definition of P, note that $y_{\kappa \log^*(v_\alpha)}$ and $y_{\kappa \log^*(v_\alpha)+1}$ are neighbours in P, which implies that v_α and v_β are neighbours in P. This is important because it implies that, if the nodes of P run the algorithm EarlyStopCV from Proposition 1, the nodes labeled v_α and v_β will choose different colours from the set $\{0, 1, 2\}$.

The rest of the proof shows that, for $x \in \{\alpha, \beta\}$, the graph G_x built in Stage 1 by agent x is an induced subgraph of P. This is sufficient since it implies that having each agent x simulate the algorithm EarlyStopCV on their local G_x results in the same colour assignment to the node labeled v_x as an execution of EarlyStopCV on the nodes of P, so v_α and v_β will be assigned different colours at the end of Stage 1 of Algorithm $\mathcal{A}_{\text{rv-D}}$. Since the colour assignment is not changed in any round after Stage 1, the result follows.

First, consider v_α. Let w_0 be the label of the leftmost node of G_α. By the definition of G_α, the node labeled w_0 is at distance exactly $\kappa \log^*(v_\alpha)$ to the left of v_α in G_α, so the node labeled w_0 is at distance exactly $D \cdot \kappa \log^*(v_\alpha)$ to the left of v_α in L. This proves that $w_0 = y_0 \in P$. For each $j \in \{0, \ldots, 2\kappa \log^*(v_\alpha)\}$, define w_j to be the label of the node at distance j to the right of the node labeled w_0 in G_α. By induction on j, we prove that $w_j = y_j \in P$ for all $j \in \{0, \ldots, 2\kappa \log^*(v_\alpha)\}$. The base case $w_0 = y_0 \in P$ was proved above. As induction hypothesis, assume that $w_{j-1} = y_{j-1} \in P$ for some $j \in \{1, \ldots, 2\kappa \log^*(v_\alpha)\}$. Consider w_j, which by definition of G_α is the neighbour to the right of w_{j-1} in G_α, and, moreover, is located distance exactly D to the right of w_{j-1} in L. By the induction hypothesis, we know that

$w_{j-1} = y_{j-1}$, so w_j is located distance exactly D to the right of y_{j-1} in L, and so $w_j = y_j$ by the definition of y_j. To confirm that $y_j \in P$, we note that $j \leq 2\kappa \log^*(v_\alpha) < 2\kappa \log^*(v_\alpha) + 1$, and that the rightmost node in P is $y_{2\kappa \log^*(v_\alpha)+1}$. This concludes the inductive step, and the proof that G_α is an induced subgraph of P.

Next, consider v_β. Let u_0 be the label of the leftmost node of G_β. By the definition of G_β, the node labeled u_0 in G_β is at distance exactly $\kappa \log^*(v_\beta)$ to the left of v_β, so, in L, the node labeled u_0 is at distance exactly $D \cdot \kappa \log^*(v_\beta)$ to the left of v_β. However, since v_α is located at distance exactly D to the left of v_β in L, and y_0 is located at distance exactly $D \cdot \kappa \log^*(v_\alpha)$ to the left of v_α in L, it follows that y_0 is located at distance exactly $D \cdot (1 + \kappa \log^*(v_\alpha))$ to the left of v_β in L. Since $v_\alpha > v_\beta$, we conclude that $d(y_0, v_\beta) = D \cdot (1 + \kappa \log^*(v_\alpha)) > D \cdot \kappa \log^*(v_\beta) = d(u_0, v_\beta)$ in L, so y_0 must be to the left of u_0 in L. Further, it means that $d(u_0, y_0) = D \cdot (1+\kappa \log^*(v_\alpha)) - D \cdot \kappa \log^*(v_\beta) = D \cdot [1+\kappa \log^*(v_\alpha) - \kappa \log^*(v_\beta)]$ in L. This proves that $u_0 = y_{1+\kappa \log^*(v_\alpha) - \kappa \log^*(v_\beta)}$. We confirm that $y_{1+\kappa \log^*(v_\alpha) - \kappa \log^*(v_\beta)} \in P$ by noticing that the subscript $1 + \kappa \log^*(v_\alpha) - \kappa \log^*(v_\beta)$ lies in the set $\{1, \ldots, 2\kappa \log^*(v_\alpha) + 1\}$ since $v_\alpha > v_\beta$. Next, define $h = 1 + \kappa \log^*(v_\alpha) - \kappa \log^*(v_\beta)$, and, for each $j \in \{0, \ldots, 2\kappa \log^*(v_\beta)\}$, define u_j to be the label of the node at distance j to the right of the node labeled u_0 in G_β. By induction on j, we prove that $u_j = y_{j+h} \in P$ for all $j \in \{0, \ldots, 2\kappa \log^*(v_\beta)\}$. The base case $u_0 = y_h \in P$ was proved above. As induction hypothesis, assume that $u_{j-1} = y_{j-1+h} \in P$ for some $j \in \{1, \ldots, 2\kappa \log^*(v_\beta)\}$. Consider u_j, which by definition of G_β is the neighbour to the right of u_{j-1} in G_β, and, moreover, is located distance exactly D to the right of u_{j-1} in L. By the induction hypothesis, we know that $u_{j-1} = y_{j-1+h}$, so u_j is located distance exactly D to the right of y_{j-1+h} in L, and so $u_j = y_{j+h}$ by the definition of y_h. To confirm that $y_{j+h} \in P$, we note that $j \leq 2\kappa \log^*(v_\beta)$ and $h = 1 + \kappa \log^*(v_\alpha) - \kappa \log^*(v_\beta)$, so $j + h \leq 1 + \kappa \log^*(v_\alpha) + \kappa \log^*(v_\beta) \leq 2\kappa \log^*(v_\alpha) + 1$, where the last inequality is due to $v_\alpha > v_\beta$. As the rightmost node of P is $y_{2\kappa \log^*(v_\alpha)+1}$, it follows that y_{j+h} is at or to the left of the rightmost node in P, so $y_{j+h} \in P$. This concludes the inductive step, and the proof that G_α is an induced subgraph of P. ◀

To prove that the algorithm correctly solves rendezvous within $O(D \log^* \ell)$ rounds for arbitrary delay between starting rounds, there are two main cases to consider. If the delay is large, then the late agent is idling for the early agent's entire execution of Stage 1, and rendezvous will occur while the early agent is exploring its $(D\kappa \log^* \ell)$-neighbourhood. Otherwise, the delay is relatively small, so both agents reach Stage 2 quickly, and the block structure of the repeated periods ensures that one agent will search while the other waits, so rendezvous will occur. These arguments are formalized in the following result.

▶ **Theorem 5.** *Algorithm $\mathcal{A}_{rv\text{-}D}$ solves rendezvous in $O(D \log^* \ell)$ rounds on lines with arbitrary node labelings when two agents start at arbitrary positions at known distance D, and when the delay between the rounds in which they start executing the algorithm is arbitrary.*

Proof. The agent that starts executing the algorithm first is called the *early agent*, and the other agent is called the *late agent*. If both agents start executing the algorithm in the same round, then arbitrarily call one of them early and the other one late. Without loss of generality, we assume that α is the early agent. The number of rounds that elapse between the two starting rounds is denoted by $delay(\alpha, \beta)$.

First, we address the case where the delay between the starting rounds of the agents is large. This situation is relatively easy to analyze, since agent β remains idle during agent α's entire execution up until they meet.

▷ Claim 6. If $delay(\alpha, \beta) > 4D\kappa \log^* v_\alpha$, then rendezvous occurs within $4D\kappa \log^* v_\alpha$ rounds in agent α's execution.

To prove the claim, we note that the total number of rounds that elapse in agent α's execution of Stage 1 is $4D\kappa \log^* v_\alpha$. By the assumption about $\mathit{delay}(\alpha, \beta)$, agent β is idle at its starting node for α's entire execution of Stage 1. Finally, note that α explores its entire $(D\kappa \log^* \alpha)$-neighbourhood during its execution of Stage 1, where $D\kappa \log^* \alpha \geq D$, so it follows that agent α will be located at agent β's starting node at some round in its execution of Stage 1, which completes the proof of the claim.

So, in the remainder of the proof, we assume that $\mathit{delay}(\alpha, \beta) \leq 4D\kappa \log^* \alpha$. We prove that rendezvous occurs during Stage 2 by considering the round τ in which agent β starts Stage 2 of its execution. By Lemma 4, we have $c_\alpha \neq c_\beta$ in every round from this time onward. We separately consider cases based on the values of c_α and c_β.

- **Case 1: $\mathbf{c}_\alpha = \mathbf{0}$ or $\mathbf{c}_\beta = \mathbf{0}$.** Let x be the agent with $c_x = 0$, and let y be the other agent. Since $c_x \neq c_y$, we know that c_y is either 1 or 2. From the algorithm's description, agent y will start a searching block within $8D$ rounds after round τ. Moreover, agent x remains idle for the entirety of Stage 2 of its execution. Since y explores its entire D-neighbourhood during a searching block, it follows that y will be located at x's starting node within $12D$ rounds after τ.
- **Case 2: $\mathbf{c}_\alpha = \mathbf{1}$ or $\mathbf{c}_\beta = \mathbf{1}$.** Let x be the agent with $c_x = 1$, and let y be the other agent. Since $c_x \neq c_y$, we know that c_y is either 0 or 2. As the case $c_y = 0$ is covered by Case 1 above, we proceed with $c_y = 2$. From the algorithm's description, agent x has a waiting block that begins within $8D$ rounds after τ. Suppose that this waiting block starts in some round t of x's execution, then we know that x stays idle at its starting node for the $4D$ rounds after t. But round t corresponds to the i'th round of some searching block in agent y's execution of Stage 2 (since y only performs searching blocks). In the first $4D - i$ rounds, agent y completes its searching block, and then performs the first i rounds of the next searching block. Since the searching blocks are performed using the same movements each time, it follows that y explores its entire D-neighbourhood in the $4D$ rounds after t, so will meet x at x's starting node within those rounds. Altogether, since t occurs within $8D$ rounds after τ, and y's searching block takes another $4D$ rounds, it follows that rendezvous occurs within $12D$ rounds after τ.

In all cases, we proved that rendezvous occurs within $12D$ rounds after τ. But τ is the round in which agent β starts Stage 2, so $\tau \leq 4D\kappa \log^* v_\beta$ in β's execution. By our assumption on the delay, β's execution starts at most $4D\kappa \log^* v_\alpha$ rounds after the beginning of α's execution. Altogether, this means that rendezvous occurs within $4D\kappa \log^* v_\alpha + 4D\kappa \log^* v_\beta + 12D$ rounds from the start of α's execution. Setting $\ell = \max\{v_\alpha, v_\beta\}$, we get that rendezvous occurs within time $8D\kappa \log^* \ell + 12D \in O(D \log^* \ell)$, as desired. ◀

4.3 Lower bound

In this section we prove a $\Omega(D \log^* \ell)$ lower bound for rendezvous time on the infinite line, where ℓ is the larger label of the two starting nodes, even assuming that agents start simultaneously, they know the initial distance D between them, and they have a global sense of direction. We summarize the argument here, and omit the detailed proofs of the results. We start with some terminology about rendezvous executions.

▶ **Definition 7.** *Consider any labeled infinite line L, and any two nodes labeled v, w on L that are at fixed distance D. Consider any rendezvous algorithm $\mathcal{A}$, and suppose that two agents start executing $\mathcal{A}$ in the same round: one agent α_v starting at node v, and the other agent α_w starting at node w. Denote by $\gamma(\mathcal{A}, L, v, w)$ the resulting execution until α_v and*

α_w meet. When $\mathcal{A}$ and L are clear from the context, we will simply write $\gamma(v,w)$ to denote the execution. Denote by $|\gamma(\mathcal{A},L,v,w)|$ the number of rounds that have elapsed before α_v and α_w meet in the execution.

The following definition formalizes the notion of "behaviour sequence": an integer sequence that encodes the movements made by an agent in each round of an algorithm's execution.

▶ **Definition 8.** *Consider any execution $\gamma(\mathcal{A},L,v,w)$ by an agent α_v starting at node v and an agent α_w starting at node w, both agents starting simultaneously. Define the behaviour sequence $\mathcal{B}_v(\mathcal{A},L,v,w)$ as follows: for each $t \in \{1,\ldots,|\gamma(\mathcal{A},L,v,w)|\}$, set the t'th element to 0 if α_v moves left in round t of the execution, to 1 if α_v stays at its current node in round t of the execution, and to 2 if α_v moves right in round t of the execution. Similarly, define the sequence $\mathcal{B}_w(\mathcal{A},L,v,w)$ using the moves by agent α_w. When $\mathcal{A}$ and L are clear from the context, we will simply write $\mathcal{B}_v(v,w)$ and $\mathcal{B}_w(v,w)$ to denote the two behaviour sequences of α_v and α_w, respectively.*

As the agents are anonymous and we only consider deterministic algorithms, note that for any fixed $\mathcal{A},L,v,w$, we have $\gamma(v,w)=\gamma(w,v)$ and $\mathcal{B}_v(v,w)=\mathcal{B}_v(w,v)$ and $\mathcal{B}_w(v,w)=\mathcal{B}_w(w,v)$. Moreover, for a fixed starting node v on a fixed line L, the behaviour of an agent running an algorithm $\mathcal{A}$ does not depend on the starting node (or behaviour) of the other agent, until the two agents meet. This implies the following result, i.e., if we look at two executions of $\mathcal{A}$ where one agent α_v starts at the same node v in both executions, then α_v's behaviour in both executions is exactly the same up until rendezvous occurs in the shorter execution.

▶ **Proposition 9.** *Consider any labeled infinite line L, and any fixed rendezvous algorithm $\mathcal{A}$. Consider any fixed node v in L, and let w_1 and w_2 be two nodes other than v. Let $p=\min\{|\gamma(v,w_1)|,|\gamma(v,w_2)|\}$. Then $\mathcal{B}_v(v,w_1)$ and $\mathcal{B}_v(v,w_2)$ have equal prefixes of length p.*

The following proposition states that two agents running a rendezvous algorithm starting at two different nodes cannot have the same behaviour sequence. This follows from the fact that the distance between two agents cannot decrease if they perform the same action in each round (i.e., both move left, both move right, or both don't move).

▶ **Proposition 10.** *Consider any labeled infinite line L, and any fixed rendezvous algorithm $\mathcal{A}$. For any two nodes x and y in L, in the execution $\gamma(x,y)$ we have $\mathcal{B}_x(x,y) \neq \mathcal{B}_y(x,y)$.*

The remainder of this section is dedicated to proving the $\Omega(D\log^* \ell)$ lower bound for rendezvous on the infinite line when the two agents start at a known distance D apart. We proceed in two steps: first, we prove an $\Omega(\log^* \ell)$ lower bound in the case where $D=1$, and then we prove the general $\Omega(D\log^* \ell)$ lower bound using a reduction from the $D=1$ case. Throughout this section, we will refer to the constant κ that was defined in Proposition 1 in order to state the running time bound of the algorithm EarlyStopCV.

4.3.1 The $D = 1$ case

We prove a $\Omega(\log^* \ell)$ lower bound for rendezvous on the infinite line, where ℓ is the larger label of the two starting nodes, in the special case where the two agents start at adjacent nodes. This lower bound applies even to algorithms that start simultaneously and know that the initial distance between the two agents is 1.

The overall idea is to assume that there exists a very fast rendezvous algorithm (i.e., an algorithm that always terminates within $\frac{1}{16\kappa}\log^*(\ell)$ rounds) and prove that this implies the existence of a distributed 3-colouring algorithm for the $\mathcal{LOCAL}$ model whose running time is faster than the lower bound proven by Linial (see Proposition 2). This contradiction proves that any rendezvous algorithm must have running time $\Omega(\log^*(\ell))$.

The first step is to reduce distributed colouring in the $\mathcal{LOCAL}$ model to rendezvous. The following result describes how to use the rendezvous algorithm to create the distributed colouring algorithm. The idea is to record the agent's behaviour sequence in the execution of the rendezvous algorithm, and convert the sequence to an integer colour. Proposition 10 guarantees that the assigned colours are different.

▶ **Lemma 11.** *Consider any rendezvous algorithm $\mathcal{A}_{rv}$ that always terminates within $\frac{1}{16\kappa}\log^*(\ell)$ rounds, where ℓ is the larger label of the two starting nodes. Then there exists a distributed colouring algorithm $\mathcal{A}_{col}$ such that, for any labeled infinite line L, and for any finite subline P of L consisting of nodes whose labels are bounded above by some integer Y, algorithm $\mathcal{A}_{col}$ uses $\lfloor \frac{1}{16\kappa}\log^*(Y)\rfloor + 1$ rounds of communication and assigns to each node in P an integer colour from the range $1, \dots, 4^{2\lfloor \frac{1}{16\kappa}\log^*(Y)\rfloor+1}$.*

The second step is to take the algorithm $\mathcal{A}_{\text{col}}$ from Lemma 11 and turn it into a 3-colouring algorithm $\mathcal{A}_{3\text{col}}$ using very few additional rounds, by using the algorithm EarlyStopCV from Proposition 1 to quickly reduce the number of colours down to 3. Combined with the previous lemma, we get the following result that shows how to obtain a very fast distributed 3-colouring algorithm under the assumption that we have a very fast rendezvous algorithm.

▶ **Lemma 12.** *Consider any rendezvous algorithm $\mathcal{A}_{rv}$ that always terminates within $\frac{1}{16\kappa}\log^*(\ell)$ rounds, where ℓ is the larger label of the two starting nodes. Then there exists a distributed 3-colouring algorithm $\mathcal{A}_{3col}$ such that, for any labeled infinite line L, and for any finite subline P of L consisting of nodes whose labels are bounded above by some integer Y, algorithm $\mathcal{A}_{3col}$ uses at most $\left(\frac{1}{4}+\frac{1}{16\kappa}\right)\log^*(Y)+1+3\kappa$ rounds of communication to 3-colour the nodes of P.*

Finally, we demonstrate how to use the above result to prove the desired $\Omega(\log^* \ell)$ lower bound for rendezvous. The idea is to construct an infinite line that contains an infinite sequence of finite sublines, each of which is a worst-case instance (according to Linial's lower bound), and obtaining the desired contradiction by observing that the upper bound on the running time of $\mathcal{A}_{3\text{col}}$ violates the lower bound guaranteed by Linial's result.

▶ **Lemma 13.** *Any algorithm that solves the rendezvous task on all labeled infinite lines, where the two agents start at adjacent nodes, uses $\Omega(\log^* \ell)$ rounds in the worst case, where ℓ is the larger label of the two starting nodes.*

4.3.2 The $D > 1$ case

We prove a $\Omega(D\log^* \ell)$ lower bound for rendezvous on the infinite line, where ℓ is the larger label of the two starting nodes, in the case where the two agents start at nodes that are distance $D > 1$ apart. This lower bound applies even to algorithms that start simultaneously and know that the initial distance between the two agents is D. Hence it shows that the running time of Algorithm $\mathcal{A}_{\text{rv-D}}$ has optimal order of magnitude among rendezvous algorithms knowing the initial distance between the agents.

The overall idea is to assume that there exists a very fast rendezvous algorithm called $\mathcal{A}_{\text{rv}}$ (that always terminates within $\frac{1}{224\kappa}D\log^* \ell$ rounds) and prove that this implies the existence of a rendezvous algorithm $\mathcal{A}_{\text{rv-adj}}$ for the $D = 1$ case that always terminates within $\frac{1}{16\kappa}\log^* \ell$ rounds, which we already proved is impossible in Section 4.3.1. This contradiction proves that any rendezvous algorithm for the $D > 1$ case must have running time $\Omega(D\log^* \ell)$.

The proof can be summarized as follows: take any instance where the agents start at adjacent nodes, and "blow it up" by a factor of D, i.e., all node labels are multiplied by a factor of D, and $D-1$ "dummy nodes" are inserted between each pair of nodes. Each node from the original instance, together with the $D-1$ nodes to its right, are called a *segment* in the blown-up instance. Then, the algorithm $\mathcal{A}_{\text{rv}}$ is locally simulated on the blown-up instance in *stages* consisting of D rounds each, and each simulated stage corresponds to 1 round of algorithm $\mathcal{A}_{\text{rv-adj}}$ in the original instance. At the end of every simulated stage, the simulated agent is in some segment that has leftmost node v, so the real agent situates itself at node v for the corresponding round in the original instance. Roughly speaking, since $\mathcal{A}_{\text{rv}}$ guarantees rendezvous in the blown-up instance, the two simulated agents will end up in the same segment (with the same leftmost node v), so the two real agents will end up at node v in the original instance. Further, since each stage of D simulated rounds corresponds to one round in the original instance, the number of rounds used by $\mathcal{A}_{\text{rv-adj}}$ is a factor of D less than the running time of the simulated algorithm $\mathcal{A}_{\text{rv}}$. This gives us a contradiction, as the resulting running time for $\mathcal{A}_{\text{rv-adj}}$ is smaller than the lower bound proven in Lemma 13.

However, the above summary overlooks some complications.

1. **Assigning labels to the dummy nodes in the blown-up instance:** the agent in the original instance needs to assign labels to nearby dummy nodes in the blown-up instance in a way that is consistent with the original instance. However, the agent initially only knows the label at its own node, which is insufficient. To address this issue, before each simulated stage of $\mathcal{A}_{\text{rv}}$, the agent uses 4 rounds to visit its neighbouring nodes in the original instance so that it can learn about neighbouring labels, which it then uses to accurately assign labels to nearby dummy nodes in the blown-up instance.
2. **Guaranteeing rendezvous:** two agents running $\mathcal{A}_{\text{rv-adj}}$ are each independently simulating $\mathcal{A}_{\text{rv}}$ locally in their own memory, so they cannot detect if the simulated agents meet at a node in the blown up instance. So, although $\mathcal{A}_{\text{rv}}$ guarantees rendezvous in the blown up instance, it might be the case that there is never a stage of D simulated rounds after which the two simulated agents end up in the same segment, since the simulated agents might continue moving after the undetected rendezvous and end up in different segments at the end of the simulated stage. To address this issue, we carefully choose the segment length and stage length appropriately to guarantee that, at the end of the stage containing the undetected rendezvous, the two simulated agents are either in the same segment or in neighbouring segments in the blown up instance. After each simulated stage is done, the real agents do a 3-round "dance" in the original instance in such a way that rendezvous will occur after the undetected simulated rendezvous.

▶ **Theorem 14.** *Any algorithm that solves the rendezvous task on all labeled infinite lines, where the two agents start at known distance $D > 1$ apart, uses $\Omega(D \log^* \ell)$ rounds in the worst case, where ℓ is the larger label of the two starting nodes.*

5 Arbitrary lines with unknown initial distance between agents

In this section, we describe an algorithm called $\mathcal{A}_{\text{rv-noD}}$ that solves rendezvous on lines with arbitrary node labelings in time $O(D^2(\log^* \ell)^3)$ (where D is the initial distance between the agents and ℓ is the larger label of the two starting nodes) when two agents start at arbitrary positions and when the delay between the rounds in which they start executing the algorithm is arbitrary. The agents do not know the initial distance D between them, and they do not know the delay between the starting rounds. Also, we note that the agents have no global sense of direction, but each agent can locally choose port 0 from its starting node to represent "right" and port 1 from its starting node to represent "left". Further, using knowledge of the port number of the edge on which it arrived at a node, an agent is able to choose whether its next move will continue in the same direction or if it will switch directions. Without loss of generality, we may assume that all node labels are strictly greater than one, since the algorithm could be re-written to add one to each label value in its own memory before carrying out any computations involving the labels. This assumption ensures that, for any node label v, the value of $\log^*(v)$ is strictly greater than 0.

As seen in Section 4.1, if the initial distance D between the agent is known, then we have an algorithm $\mathcal{A}_{\text{rv-D}}$ that will solve rendezvous in $O(D\log^*\ell)$ rounds. We wish to extend that algorithm for the case of unknown distance by repeatedly running $\mathcal{A}_{\text{rv-D}}$ with guessed values for D. To get an optimal algorithm, i.e., with running time $O(D\log^*\ell)$, a natural attempt would be to proceed by doubling the guess until it exceeds D, so that the searching range of an agent includes the starting node of the other agent. However, this approach will not work in our case, because our algorithm $\mathcal{A}_{\text{rv-D}}$ requires the exact value of D to guarantee rendezvous. More specifically, the colouring stage using guess g only guarantees that nodes at distance exactly g are assigned different colours. So, instead, our algorithm $\mathcal{A}_{\text{rv-noD}}$ increments the guessed value by 1 so that the guess is guaranteed to eventually be equal to D, which results in a running time quadratic in D instead of linear in D.

At a high level, our algorithm $\mathcal{A}_{\text{rv-noD}}$ consists of phases, where each phase is an attempt to solve rendezvous using a value g which is a guess for the value D. The first phase sets $g = 1$. Each phase has three stages. In the first stage, the agent waits at its starting node for a fixed number of rounds. In the second and third stages, the agent executes a modified version of the algorithm $\mathcal{A}_{\text{rv-D}}$ from Section 4.1 using the value g instead of D. At the end of each phase, if rendezvous has not yet occurred, the agent increments its guess g and proceeds to the next phase. A major complication is that an adversary can choose the wake-up times of the agents so that the phases do not align well, e.g., the agents are using the same guess g but are at different parts of the phase, or, they are in different phases and not using the same guess g. This means we have to very carefully design the phases and algorithm analysis to account for arbitrary delays between the wake-up times.

The detailed description of the algorithm is as follows. Consider an agent x whose execution starts at a node labeled v_x. We now describe an arbitrary phase in the algorithm's execution. Let $g \geq 1$, let $d = 1 + \lfloor \log_2 g \rfloor$, and recall that $\kappa > 1$ is an integer constant defined in the running time of the algorithm EarlyStopCV from Proposition 1. The g-th phase executed by agent x, denoted by $\mathcal{P}_g^x$, consists of executing the following three stages.

Stage 0: Wait. Stay at the node v_x for $36 \cdot 2^d \cdot \kappa \log^*(v_x)$ rounds.

Stage 1: Colouring. Let $r = g \cdot \kappa \log^*(v_x)$. Denote by $\mathcal{B}_r$ the r-neighbourhood of v_x, and let V_g be the subset of nodes in $\mathcal{B}_r$ whose distance from v_x is an integer multiple of g. First, agent x determines $\mathcal{B}_r$ (including all node labels) by moving right r times, then left $2r$ times, then right r times, ending back at its starting node v_x. Then, in its local memory, agent x creates a path graph G_x consisting of the nodes in V_g, with two nodes connected by an edge if and only if their distance in $\mathcal{B}_r$ is exactly g. This forms a path graph centered at v_x with $\kappa \log^*(v_x)$ nodes in each direction. The agent simulates an execution of the algorithm EarlyStopCV from Proposition 1 by the nodes of G_x to obtain a colour $c_x \in \{0, 1, 2\}$. Let CV_x be the 2-bit binary representation of c_x. Transform CV_x into an 8-bit binary string CV'_x by replacing each 0 in CV_x with 0011, and replacing each 1 in CV_x with 1100. Finally, create a 9-bit string S by appending a 1 to CV'_x.

Stage 2: Search. This stage consists of performing $\kappa \log^*(v_x)$ periods of $|S| = 9$ blocks each. Block i of a period is designated as a *waiting* block if the i'th bit of S is 0, else it is designated as a *searching* block. A waiting block consists of $4 \cdot (2^d)$ consecutive rounds during which the agent stays at its starting node. A searching block consists of $4 \cdot (2^d)$ consecutive rounds: the agent first moves right 2^d times, then left $2 \cdot 2^d$ times, then right 2^d times.

6 Conclusion

We presented rendezvous algorithms for three scenarios: the scenario of the canonical line, the scenario of arbitrary labeling with known initial distance D, and the scenario where each agent knows *a priori* only the label of its starting node. While for the first two scenarios the complexity of our algorithms is optimal (respectively $O(D)$ and $O(D \log^* \ell)$, where ℓ is the larger label of the two starting nodes), for the most general scenario, where each agent knows *a priori* only the label of its starting node, the complexity of our algorithm is $O(D^2(\log^* \ell)^3)$, for arbitrary unknown D, while the best known lower bound, valid also in this scenario, is $\Omega(D \log^* \ell)$.

The natural open problem is the optimal complexity of rendezvous in the most general scenario (with arbitrary labeling and unknown D), both for the infinite labeled line and for the finite labeled lines and cycles. This open problem can be generalized to the class of arbitrary trees or even arbitrary graphs.

References

1 Steve Alpern. The rendezvous search problem. *SIAM Journal on Control and Optimization*, 33(3):673–683, 1995. doi:10.1137/S0363012993249195.

2 Steve Alpern. Rendezvous search on labeled networks. *Naval Research Logistics (NRL)*, 49(3):256–274, 2002. doi:10.1002/nav.10011.

3 Steve Alpern and Shmuel Gal. *The theory of search games and rendezvous*, volume 55 of *International series in operations research and management science*. Kluwer, 2003.

4 E. J. Anderson and R. R. Weber. The rendezvous problem on discrete locations. *Journal of Applied Probability*, 27(4):839–851, 1990. URL: http://www.jstor.org/stable/3214827.

5 Edward J. Anderson and Sándor P. Fekete. Asymmetric rendezvous on the plane. In Ravi Janardan, editor, *Proceedings of the Fourteenth Annual Symposium on Computational Geometry, Minneapolis, Minnesota, USA, June 7-10, 1998*, pages 365–373. ACM, 1998. doi:10.1145/276884.276925.

6 Edward J. Anderson and Sándor P. Fekete. Two dimensional rendezvous search. *Oper. Res.*, 49(1):107–118, 2001. doi:10.1287/opre.49.1.107.11191.

7 Vic Baston and Shmuel Gal. Rendezvous on the line when the players' initial distance is given by an unknown probability distribution. *SIAM Journal on Control and Optimization*, 36(6):1880–1889, 1998. doi:10.1137/S0363012996314130.

8 Vic Baston and Shmuel Gal. Rendezvous search when marks are left at the starting points. *Naval Research Logistics*, 48(8):722–731, December 2001. doi:10.1002/nav.1044.

9 Subhash Bhagat and Andrzej Pelc. Deterministic rendezvous in infinite trees. *CoRR*, abs/2203.05160, 2022. doi:10.48550/arXiv.2203.05160.

10 Subhash Bhagat and Andrzej Pelc. How to meet at a node of any connected graph. In Christian Scheideler, editor, *36th International Symposium on Distributed Computing, DISC 2022, October 25-27, 2022, Augusta, Georgia, USA*, volume 246 of *LIPIcs*, pages 11:1–11:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.DISC.2022.11.

11 Sébastien Bouchard, Yoann Dieudonné, Andrzej Pelc, and Franck Petit. Almost universal anonymous rendezvous in the plane. In Christian Scheideler and Michael Spear, editors, *SPAA '20: 32nd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, July 15-17, 2020*, pages 117–127. ACM, 2020. doi:10.1145/3350755.3400283.

12 Richard Cole and Uzi Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Inf. Control.*, 70(1):32–53, 1986. doi:10.1016/S0019-9958(86)80023-7.

13 Andrew Collins, Jurek Czyzowicz, Leszek Gasieniec, Adrian Kosowski, and Russell A. Martin. Synchronous rendezvous for location-aware agents. In David Peleg, editor, *Distributed Computing - 25th International Symposium, DISC 2011, Rome, Italy, September 20-22, 2011. Proceedings*, volume 6950 of *Lecture Notes in Computer Science*, pages 447–459. Springer, 2011. doi:10.1007/978-3-642-24100-0_42.

14 Jurek Czyzowicz, Leszek Gasieniec, Ryan Killick, and Evangelos Kranakis. Symmetry breaking in the plane: Rendezvous by robots with unknown attributes. In Peter Robinson and Faith Ellen, editors, *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 4–13. ACM, 2019. doi:10.1145/3293611.3331608.

15 Jurek Czyzowicz, Adrian Kosowski, and Andrzej Pelc. How to meet when you forget: log-space rendezvous in arbitrary graphs. *Distributed Comput.*, 25(2):165–178, 2012. doi:10.1007/s00446-011-0141-9.

16 Anders Dessmark, Pierre Fraigniaud, Dariusz R. Kowalski, and Andrzej Pelc. Deterministic rendezvous in graphs. *Algorithmica*, 46(1):69–96, 2006. doi:10.1007/s00453-006-0074-2.

17 Yoann Dieudonné and Andrzej Pelc. Anonymous meeting in networks. *Algorithmica*, 74(2):908–946, 2016. doi:10.1007/s00453-015-9982-0.

18 Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Peter Widmayer. Gathering of asynchronous robots with limited visibility. *Theor. Comput. Sci.*, 337(1-3):147–168, 2005. doi:10.1016/j.tcs.2005.01.001.

19 Pierre Fraigniaud and Andrzej Pelc. Delays induce an exponential memory gap for rendezvous in trees. *ACM Trans. Algorithms*, 9(2):17:1–17:24, 2013. doi:10.1145/2438645.2438649.

20 Dariusz R. Kowalski and Adam Malinowski. How to meet in anonymous network. *Theor. Comput. Sci.*, 399(1-2):141–156, 2008. doi:10.1016/j.tcs.2008.02.010.

21 Evangelos Kranakis, Danny Krizanc, and Pat Morin. Randomized rendezvous with limited memory. *ACM Trans. Algorithms*, 7(3):34:1–34:12, 2011. doi:10.1145/1978782.1978789.

22 Evangelos Kranakis, Nicola Santoro, Cindy Sawchuk, and Danny Krizanc. Mobile agent rendezvous in a ring. In *23rd International Conference on Distributed Computing Systems (ICDCS 2003), 19-22 May 2003, Providence, RI, USA*, pages 592–599. IEEE Computer Society, 2003. doi:10.1109/ICDCS.2003.1203510.

23 Juhana Laurinharju and Jukka Suomela. Linial's lower bound made easy. *CoRR*, abs/1402.2552, 2014. URL: http://arxiv.org/abs/1402.2552, arXiv:1402.2552.

24 Wei Shi Lim and Steve Alpern. Minimax rendezvous on the line. *SIAM Journal on Control and Optimization*, 34(5):1650–1665, 1996. doi:10.1137/S036301299427816X.

25 Nathan Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992. doi:10.1137/0221015.

26 Avery Miller and Andrzej Pelc. Tradeoffs between cost and information for rendezvous and treasure hunt. *J. Parallel Distributed Comput.*, 83:159–167, 2015. doi:10.1016/j.jpdc.2015.06.004.

27 Andrzej Pelc. Deterministic rendezvous in networks: A comprehensive survey. *Networks*, 59(3):331–347, 2012. doi:10.1002/net.21453.

28 Andrzej Pelc. Deterministic rendezvous algorithms. In Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro, editors, *Distributed Computing by Mobile Entities, Current Research in Moving and Computing*, volume 11340 of *Lecture Notes in Computer Science*, pages 423–454. Springer, 2019. doi:10.1007/978-3-030-11072-7_17.

29 David Peleg. *Distributed Computing: A Locality-Sensitive Approach.* Society for Industrial and Applied Mathematics, 2000. doi:10.1137/1.9780898719772.

30 Amnon Ta-Shma and Uri Zwick. Deterministic rendezvous, treasure hunts, and strongly universal exploration sequences. *ACM Trans. Algorithms*, 10(3):12:1–12:15, 2014. doi:10.1145/2601068.

31 L. C. Thomas. Finding your kids when they are lost. *Journal of the Operational Research Society*, 43(6):637–639, 1992. doi:10.1057/jors.1992.89.

32 Roger Wattenhofer. Principles of distributed computing, 2023. Accessed on 2023-04-16. URL: https://disco.ethz.ch/courses/fs23/podc/.

A The Line Colouring Algorithm

As announced in Proposition 1 in Section 2, we design a deterministic distributed algorithm EarlyStopCV that, in the $\mathcal{LOCAL}$ model, properly 3-colours any infinite labeled line such that the execution of this algorithm at any node x with initial label ID_x terminates in time $O(\log^*(\text{ID}_x))$. Our algorithm builds upon a solution to a problem posed in Homework Exercises 1 from the Principles of Distributed Computing course at ETH Zürich [32].

At a very high level, we want to execute the Cole and Vishkin algorithm [12], but with some modifications. We introduce four special colours, denoted by $\alpha, \beta, \mathit{Pdone}, \mathit{Cdone}$, that are defined in such a way that they are not equal to any colour that could be chosen using the Cole and Vishkin strategy (e.g., they could be defined as negative integers). These colours will be chosen by a node in certain situations where it needs to choose a colour that is guaranteed to be different from its neighbour's, but using the Cole and Vishkin strategy might not work. Another significant modification is that each node will actually choose two colours: its "final colour" that it will output upon terminating the algorithm, but also an intermediate "Phase 1 colour". These roughly correspond to the two parts of the Cole and Vishkin strategy: in Phase 1, each node picks a colour that is guaranteed to be different from each of its neighbours' chosen colour (but selects it from a "large" range of colours), and then in Phase 2, each node picks a final colour from the set $\{0, 1, 2\}$. However, since we want to allow different nodes to execute different phases of the algorithm at the same time (since we want to allow them to terminate at different times), each node must maintain and advertise its "Phase 1" and "final" colours separately, so that another node that is still performing Phase 1 is basing its decisions on its neighbours' Phase 1 colours (and not their final colours). Finally, we note that the original Cole and Vishkin strategy is described for a directed tree, i.e., where each node has at most one parent and perhaps some children, whereas our algorithm must work in an undirected infinite line, so we introduce a pre-processing step (Phase 0) to set up parent/child relationships.

The first part of the algorithm partitions the undirected infinite line into directed sub-lines that will perform the rest of the algorithm in parallel. In round 1, each node shares its unique ID with both neighbours. In round 2 (also referred to as Phase 0), each node compares its own ID with those that it received from its neighbours in round 1. If a node determines that it is a *local minimum* (i.e., its ID is less than the ID's of its neighbours), then it picks the special colour α as its Phase 1 colour and proceeds directly to Phase 2 without performing Phase 1. If a node determines that it is a *local maximum* (i.e., its ID is greater than the ID's of its neighbours), then it picks the special colour β as its Phase 1 colour and proceeds directly to Phase 2 without performing Phase 1. All other nodes, i.e., those that are not a local minimum or a local maximum, pick their own ID as their Phase 1 colour and continue to perform Phase 1. Further, each such node also chooses one *parent* neighbour (the neighbour with smaller ID) and one *child* neighbour (the neighbour with larger ID). In particular, the nodes that perform Phase 1 are each part of a directed sub-line that is bordered by local maxima/minima, so the sub-lines can all perform Phase 1 in parallel without worrying that the chosen colours will conflict with colours chosen in other sub-lines.

The second part of the algorithm, referred to as Phase 1, essentially implements the Cole and Vishkin strategy within each directed sub-line. The idea is that each node follows the Cole and Vishkin algorithm using its own Phase 1 colour and the Phase 1 colour of its parent until it has selected a Phase 1 colour in the range $\{0, \ldots, 51\}$ that is different from its parent's Phase 1 colour, and then the node will proceed to Phase 2 where it will pick a final colour in the range $\{0, 1, 2\}$. However, this idea must be modified to avoid three potential pitfalls:

1. Suppose that, in a round t, a node v has a parent or child that stopped performing Phase 1 in an earlier round (and possibly has finished Phase 2 and has stopped executing the algorithm). If node v still has a large colour in round t, i.e., not in the range $\{0, \ldots, 51\}$, but continues to use the Cole and Vishkin strategy, then there is no guarantee that the new colour it chooses will still be different than those of its neighbours. This is not an issue for the original Cole and Vishkin algorithm, since it was designed in such a way that all nodes execute the algorithm the exact same number of times (using *a priori* knowledge of the network size). To deal with this scenario, node v first looks at the Phase 1 colours that were most recently advertised by its parent and child, and sees if either of them is in the range $\{0, \ldots, 51\}$. If node v sees that its parent's Phase 1 colour is in the range $\{0, \ldots, 51\}$, then it knows that its parent is not performing Phase 1 in this round, so instead of following the Cole and Vishkin strategy, it immediately adopts the special colour *Pdone* as its Phase 1 colour and moves on to Phase 2 of the algorithm. On the other hand, if node v sees that its child's colour is in the range $\{0, \ldots, 51\}$, then it knows that its child is not performing Phase 1 in this round, so instead of following the Cole and Vishkin strategy, it immediately adopts the special colour *Cdone* as its Phase 1 colour and moves on to Phase 2 of the algorithm. By adopting the special colours in this way, node v's Phase 1 colour is guaranteed to be different than any non-negative integer colour that was previously adopted by its neighbours.
2. Suppose that a node v's parent has a Phase 1 colour equal to a special colour (i.e., one of $\alpha, \beta, \mathit{Pdone}, \mathit{Cdone}$). The Cole and Vishkin strategy is not designed to work with such special colours, so, a node v with such a parent will pretend that its parent has colour 0 instead. By doing this, it will choose some non-negative integer as dictated by the Cole and Vishkin strategy, and this integer is guaranteed to be different from all special colours, so v's chosen Phase 1 colour will be different from its parent's.
3. Suppose that a node v has a parent with an extremely large integer colour. The Cole and Vishkin strategy will make sure that v chooses a new Phase 1 colour that is different than the Phase 1 colour chosen by its parent, however, it is not guaranteed that this new colour is significantly smaller than the colour v started with. In particular, we want to guarantee that node v terminates Phase 1 within $\log^*(\mathrm{ID}_v)$ rounds, so we want node v's newly-chosen Phase 1 colour to be bounded above by a logarithmic function of its *own* colour in every round (and never depend on its parent's much larger colour). To ensure this, we apply a suffix-free encoding to the binary representation of each node's colour **before** applying the Cole and Vishkin strategy. Doing this guarantees that the smallest index where two binary representations of colours differ is bounded above by the length of the binary representation of the *smaller* colour.

When a node is ready to proceed to Phase 2, it has chosen a Phase 1 colour from the set $\{0, \ldots, 51\} \cup \{\alpha, \beta\} \cup \{\mathit{Pdone}, \mathit{Cdone}\}$, and its chosen Phase 1 colour is guaranteed to be different from the Phase 1 colours of its two neighbours.

The third part of the algorithm, referred to as Phase 2, uses a round-robin strategy over the 56 possible Phase 1 colours, which guarantees that any two neighbouring nodes pick their final colour in different rounds. In particular, when executing each round of Phase 2, a node calculates the current *token* value, which is defined as the current round number modulo 56. We assume that all nodes start the algorithm at the same time, so the current token value is the same at all nodes in each round. In each round, each node that is performing Phase 2 compares the token value to its own Phase 1 colour. If a node v's Phase 1 colour is in $\{0, \ldots, 51\}$ and is equal to the current token value, then it proceeds to choose its final colour (as described below). Otherwise, if a node v's Phase 1 colour is one of the special colours, it waits until the current token value is equal to a value that is dedicated to that special colour (i.e., 52 for α, 53 for β, 54 for *Pdone*, 55 for *Cdone*), then chooses its final colour in that round. To choose its final colour, node v chooses the smallest colour from $\{0, 1, 2\}$ that was never previously advertised as a final colour by its neighbours. Then, v immediately sends out a message to its neighbours to advertise the final colour that it chose, then v terminates. Since two neighbouring nodes are guaranteed to have different Phase 1 colours, they will choose their final colour in different rounds, so the later of the two nodes always avoids the colour chosen by the earlier node, and there is always a colour from $\{0, 1, 2\}$ available since each node only has two neighbours.

Algorithm pseudocode

For any two binary strings S_1, S_2, denote by $S_1 \cdot S_2$ the concatenation of string S_1 followed by string S_2. For any positive integer i, the function $\textsc{BinaryRep}(i)$ returns the binary string consisting of the base-2 representation of i. Conversely, for any binary string S, the function $\textsc{IntVal}(S)$ returns the integer value when S is interpreted as a base-2 integer. For any binary string S of length $\ell \geq 1$, the string is a concatenation of bits, i.e., $S = s_{\ell-1} \cdots s_0$, and we will write $S[i]$ to denote the bit s_i. The notation $|S|$ denotes the length of S.

For any binary string S of length $\ell \geq 1$, we define a function $\textsc{EncodeSF}(S)$ that returns the binary string obtained by replacing each 0 in S with 01, replacing each 1 in S with 10, then prepending 00 to the result. More formally, $\textsc{EncodeSF}(S)$ returns a string S' of length $2\ell + 2$ such that $S'[2\ell + 1] = S'[2\ell] = 0$, and, for each $i \in \{0, \ldots, \ell - 1\}$, $S'[2i + 1] = S[i]$ and $S'[2i] = 1 - S[i]$. For example, $\textsc{EncodeSF}(101) = 00100110$. The function $\textsc{EncodeSF}$ is an encoding method with two important properties: an encoded string is uniquely decodable, and, no encoded string is a suffix of another encoded string.

We define four special colours $\alpha, \beta, Pdone, Cdone$ that are not positive integers (i.e., they cannot be confused with any node's ID, and they cannot be confused with any non-negative integer colour chosen by a node during the algorithm's execution). Practically speaking, one possible implementation is to use $\alpha = -4$, $\beta = -3$, $Pdone = -2$, and $Cdone = -1$.

Algorithm 1 provides the pseudocode for the algorithm's execution at a node. The node's two neighbours are referred to as A and B. Algorithms 2 and 3 are the subroutines that a node uses to compute its new Phase 1 colour in each round.

Algorithm 1 EarlyStopCV.

```
%% clockVal is assumed to contain the current local round number, starting at 1.
%% myID is assumed to contain the node's initial identifier.
%% Initially, Acol.P1 = Bcol.P1 = Acol.final = Bcol.final = $null$
if clockVal == 1 then                                           ▷ Round 1: get IDs of neighbours
    Send myID to both neighbours
    Receive $ID_A$ from $A$ and receive $ID_B$ from $B$
else if clockVal == 2 then ▷ Phase 0: detect if local max or local min, otherwise assign parent and child
    if myID $<$ $ID_A$ and myID $<$ $ID_B$ then                     ▷ I'm a local minimum
        myPhase1Col $\leftarrow \alpha$
    else if myID $>$ $ID_A$ and myID $>$ $ID_B$ then                ▷ I'm a local maximum
        myPhase1Col $\leftarrow \beta$
    else if $ID_A$ $<$ myID and myID $<$ $ID_B$ then    ▷ neighbourhood IDs increase towards B
        myPhase1Col $\leftarrow$ myID
        parent $\leftarrow A$
        child $\leftarrow B$
    else                                               ▷ neighbourhood IDs increase towards A
        myPhase1Col $\leftarrow$ myID
        parent $\leftarrow B$
        child $\leftarrow A$
    end if
    Send myPhase1Col to both neighbours
    Receive $msg_A$ from $A$ and receive $msg_B$ from $B$
    Acol.P1 $\leftarrow msg_A$
    Bcol.P1 $\leftarrow msg_B$
    doPhase1 $\leftarrow$ (myPhase1Col $\notin \{0, \ldots, 51, \alpha, \beta\}$)
else if doPhase1 == true then ▷ Phase 1: detect if one of my neighbours is settled, otherwise perform CV
    if parent == $A$ then
        myPhase1Col $\leftarrow$ ChooseNewPhase1Colour(myPhase1Col, Acol.P1, Bcol.P1)
    else
        myPhase1Col $\leftarrow$ ChooseNewPhase1Colour(myPhase1Col, Bcol.P1, Acol.P1)
    end if
    Send ("P1", myPhase1Col) to both neighbours
    if received a message of the form ($key$, $val$) from $A$ then: Acol.$key \leftarrow val$
    if received a message of the form ($key$, $val$) from $B$ then: Bcol.$key \leftarrow val$
    doPhase1 $\leftarrow$ (myPhase1Col $\notin \{0, \ldots, 51, Pdone, Cdone\}$)
else                                                   ▷ Phase 2: colour reduction down to $\{0, 1, 2\}$
    token $\leftarrow$ clockVal mod 56
    if (myPhase1Col == token) or
       ((myPhase1Col == $\alpha$) $\wedge$ (token == 52)) or
       ((myPhase1Col == $\beta$) $\wedge$ (token == 53)) or
       ((myPhase1Col == $Pdone$) $\wedge$ (token == 54)) or
       ((myPhase1Col == $Cdone$) $\wedge$ (token == 55)) then
            myFinalCol $\leftarrow$ smallest element in $\{0, 1, 2\} \setminus \{$Acol.final, Bcol.final$\}$
            Send ("final", myFinalCol) to both neighbours
            terminate()
    end if
    if received a message of the form ($key$, $val$) from $A$ then: Acol.$key \leftarrow val$
    if received a message of the form ($key$, $val$) from $B$ then: Bcol.$key \leftarrow val$
end if
```

Algorithm 2 $\textsc{ChooseNewPhase1Colour}(\texttt{myPhase1Col}, \texttt{ParentPhase1Col}, \texttt{ChildPhase1Col})$.

```
1: if ParentPhase1Col $\in \{0, \ldots, 51\}$ then
2:     newColour $\leftarrow$ $Pdone$
3: else if ChildPhase1Col $\in \{0, \ldots, 51\}$ then
4:     newColour $\leftarrow$ $Cdone$
5: else if ParentPhase1Col $\in \{Pdone, Cdone, \alpha, \beta\}$ then
6:     newColour $\leftarrow$ $\textsc{CVChoice}(\texttt{myPhase1Col}, 0)$
7: else
8:     newColour $\leftarrow$ $\textsc{CVChoice}(\texttt{myPhase1Col}, \texttt{ParentPhase1Col})$
9: end if
10: return newColour
```

Algorithm 3 $\textsc{CVChoice}(\texttt{MyCol}, \texttt{OtherCol})$.

```
1: MyString $\leftarrow$ $\textsc{EncodeSF}(\textsc{BinaryRep}(\texttt{MyCol}))$
2: OtherString $\leftarrow$ $\textsc{EncodeSF}(\textsc{BinaryRep}(\texttt{OtherCol}))$
3: i $\leftarrow$ smallest $x \geq 0$ such that $\texttt{MyString}[x] \neq \texttt{OtherString}[x]$
4: newString $\leftarrow$ $\textsc{BinaryRep}(\texttt{i}) \cdot \texttt{myString}[\texttt{i}]$
5: return $\textsc{IntVal}(\texttt{newString})$
```

Null Messages, Information and Coordination

Raïssa Nataf
Technion, Haifa, Israel

Guy Goren
Protocol Labs, Haifa, Israel

Yoram Moses
Technion, Haifa, Israel

Abstract

This paper investigates the role that null messages play in synchronous systems with and without failures, and provides necessary and sufficient conditions on the structure of protocols for information transfer and coordination there. We start by introducing a new and more refined definition of null messages. A generalization of message chains that allow these null messages is provided, and is shown to be necessary and sufficient for information transfer in reliable systems. Coping with crash failures requires a much richer structure, since not receiving a message may be the result of the sender's failure. We introduce a class of communication patterns called *resilient message blocks*, which impose a stricter condition on protocols than the *silent choirs* of Goren and Moses (2020). Such blocks are shown to be necessary for information transfer in crash-prone systems. Moreover, they are sufficient in several cases of interest, in which silent choirs are not. Finally, a particular combination of resilient message blocks is shown to be necessary and sufficient for solving the Ordered Response coordination problem.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Computing methodologies → Reasoning about belief and knowledge

Keywords and phrases null messages, fault tolerance, coordination, information flow, knowledge analysis

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.30

Related Version *Full Version*: `https://arxiv.org/abs/2208.10866` [17]

Funding *Yoram Moses*: Yoram Moses is the Israel Pollak academic chair at the Technion. Both his work and that of Raïssa Nataf were supported in part by the Israel Science Foundation under grant 2061/19.

1 Introduction

Communication and coordination in distributed systems depend crucially on properties of the model at hand. In synchronous systems in which processes have clocks and message transmission times are bounded, sending explicit messages is not the only way to transmit information. Suppose that a sender s needs to transmit its (binary) initial value v_s to a destination process d, in a system in which messages are delivered in 1 time step. If s follows a protocol by which it sends d a message at time 0 in case $v_s = 0$ and does not send anything if $v_s = 1$, then d can learn that $v_s = 1$ at time 1 without receiving any messages. Lamport called this "*sending a message by not sending a message*" in [13], and he referred to not sending a message over a communication channel at a given time t as sending a "*null message*." In this paper we provide a new and more precise definition of null messages, and investigate the general role that null messages play in information transfer and in coordination in synchronous systems with and without failures.

In particular, our results extend and generalize those of Goren and Moses in [7], who were the first to explicitly consider how silence can be used in systems with crash failures.

The possibility of failures makes information transfer a rather subtle issue. Denote by f an *a priori* upper bound on the number of failures per execution. Consider the following protocol, which we denote P_1: In the first round process s sends a message to p reporting whether its initial value v_s is 1 or 0. In round 2, if process p has received a message stating that $v_s = 1$ it keeps silent, and it sends an actual message to d otherwise. A run r_1 of P_1 in which $v_s = 1$ and no process fails is depicted in Figure 1. (In our figures, solid arrows represent actual messages and dashed arrows represent null messages.) Note that if $f = 0$, then process p receives the first round message from s in every run of P_1. Consequently, if $v_s = 1$ then following the second round, d learns that $v_s = 1$ since it did not hear from p. Now assume that one process may crash ($f = 1$). In this case r_1, where none fails, is a legal execution of P_1 but d is not informed that $v_s = 1$ in r_1. This is because d cannot distinguish r_1 from a run in which $v_s = 0$ and p crashes before sending its message to d.

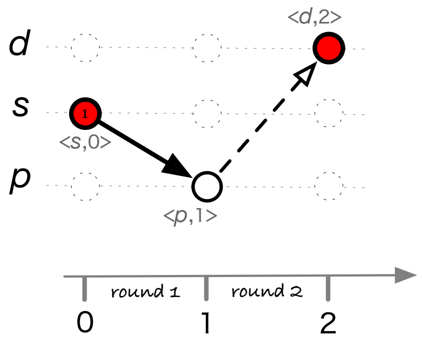


Figure 1 The run r_1 of P_1 in which d is informed that $v_s = 1$ when $f = 0$ but **not** when $f = 1$.

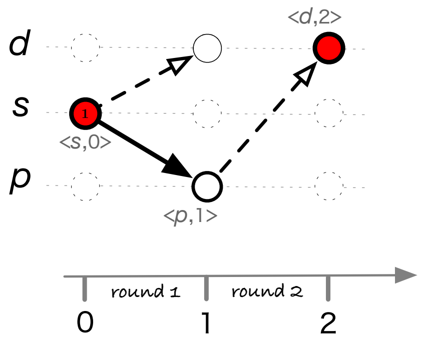


Figure 2 The run r_2 of P_2 in which a silent choir informs d that $v_s = 1$ when $f = 1$.

Consider now a protocol P_2 that differs from P_1 only in that according to P_2 process s should send d a message at time 0 if and only if $v_s = 0$. (Process s remains silent if $v_s = 1$.) Figure 2 depicts the run r_2 of P_2 where $v_s = 1$ and no failures occur. Observe that in case $f = 1$, process d is informed in r_2 that $v_s = 1$. As before, d cannot observe in r_2 whether p has crashed. However, since $f = 1$, at most one of the missing messages to d can be explained by a process crash. The other missing message must be caused by the fact that $v_s = 1$. Note that exactly the same messages are sent in r_1 and r_2. Process d obtains different information in the two cases because the protocols are different: In particular, s keeps silent toward d only under certain conditions of interest according to P_2 while it always keeps silent according to P_1. This is what provides d genuine information.

The above discussion motivates a new and more refined definition of null messages. While [13] considers not receiving a message as the receipt of a null message, we define a null message to be sent by a process i to its neighbor j at time t in a given execution if process i does not send an actual message at time t, and there is at least one execution of the protocol in which i *does* send j an actual message at time t. (A formal definition appears in Section 2.) With such a definition, a null message is guaranteed to carry some nontrivial information.

Goren and Moses showed in [7] that information can be transmitted in silence even when crashes may occur. Their *Silent Choir* Theorem states a necessary condition for d to learn the initial value of s in a crash-prone system without a message chain from s. For failure-free executions, their necessary condition becomes the following: If d knows the value of v_s at time $m > 0$ without an actual message chain (i.e., a message chain exclusively composed of actual messages) from s having reached it, there are at least $f + 1$ processes that receive an actual message chain from s by time $m - 1$ and send no message to d at time $m - 1$. These processes are called a *silent choir*. In r_2, process d learns the value of s at time 2, and we can see in Figure 2 that the set $\{s, p\}$ constitutes a silent choir. However, as we shall see, while being necessary for information transfer in crash-prone synchronous systems, the Silent Choir Theorem's conditions are not sufficient, even for failure-free executions.

Consider again the run r_1 of Figure 1 and assume $f = 1$. The set $\{s, p\}$ forms a silent choir, i.e., the conditions of the Silent Choir Theorem hold. However, as we described above, d does not learn that $v_s = 1$ in r_1. Process s, who belongs to the silent choir here, *never* sends d a message under P_1, and so its silence does not form a null message according to our new definition. Naturally, strengthening the Silent Choir condition by requiring that the processes of the silent choir actually send null messages at time $m - 1$, i.e., one time unit before d gains the knowledge that $v_s = 1$, could make it sufficient. However, the condition would then **not** be necessary. For example, as depicted in Figure 2, s does not send d a null message in the second round of r_2 (which is at time $m - 1$ in the language of the Silent Choir Theorem). Notice that, in addition, members of the silent choir do not necessarily send null messages to d. Indeed, they do not even need to be neighbors of d.

Beyond the fundamental value of studying null messages to understand the differences between synchronous and asynchronous models of distributed systems, judicious use of null messages can lead to considerable savings. For a concrete example, consider a network structured as depicted in Figure 3, in which there are different costs for sending over different channels. This can arise, for example, from the three intermediate processes residing at the same site or belonging to the same organization as s, while d is across the ocean, or just connected via expensive channels. Assume, in addition, that d needs to know the value of v_s, where normally $v_s = 1$ and only very rarely $v_s \neq 1$. Finally, we wish to be able to overcome up to $\boldsymbol{f = 2}$ process crashes. While sending an actual message chain from s to d would cost \$1001, null messages can be used to inform d that $v_s = 1$ at a cost of \$2 if $v_s = 1$ and no failures occur (see Figure 4). With such a solution, if $v_s = 0$ then the cost may in the worst case be as high as \$3003. But if the latter is rare and the former is very common, such use of null messages can provide a clear advantage. E.g., if for every 100 runs in which $v_s = 1$ and no failures occur we expect to see 2 runs where this is not the case, then using null messages for the 102 runs will cost at most \$6206, compared to \$102102 spent by a protocol that uses only actual messages in the network of Figure 3.

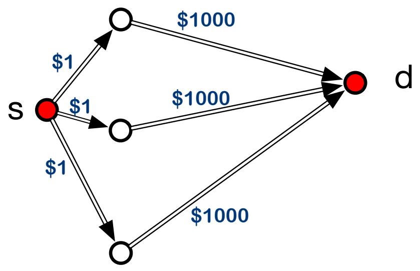


Figure 3 A network with different communication costs. Sending an actual message chain from s to d costs \$1001.

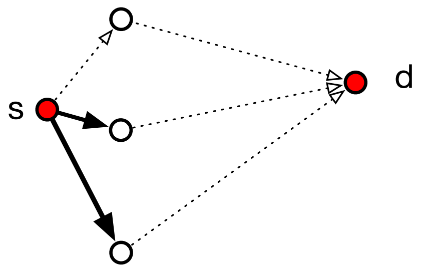


Figure 4 The sender s can inform d that $v_s = 1$ at a cost of \$2 in a failure free run assuming a bound of $f = 2$ failures.

This paper investigates the role of null messages for information flow and coordination in synchronous systems with crash failures. Its main contributions are:

1. We provide a new definition of null messages, whereby not sending a message is considered to be a null message only if it conveys nontrivial information. Moreover, we formalize an essential aspect of the synchronous model, by proving that *enhanced message chains*, which can contain both actual and null messages, are necessary for information transfer in synchronous systems.
2. We strengthen this result by proving that in order for there to be information transfer from a process s to d, process d must **know** that an enhanced message chain from s has reached it. This result plays an important role in the analysis of information transfer in the presence of failures.
3. We identify communication patterns called *resilient message blocks*, which are necessary for information transfer in crash-prone synchronous systems, proving a stronger and more general theorem than the Silent Choir Theorem of [7]. Based on this theorem, we provide necessary and sufficient conditions that characterize protocols for nice-run information transfer, in which the transfer should succeed in failure-free executions, and for Robust information transfer in which it should succeed more generally.
4. Finally, we provide an analysis of communication requirements from protocols solving the Ordered Response coordination problem, based on resilient blocks.

This paper is structured as follows. In Section 2, we define the model and present preliminary definitions and results regarding null messages, knowledge, and communication graphs, which are used throughout the analysis. We show in Section 2.2 that it is impossible to achieve information transfer from a process s to another process d in synchronous systems without constructing an enhanced message chain from s to d. We then prove the stronger fact (in Theorem 8) that in order for d to know the value of v_s, it must *know* that an enhanced message chain from s to d exists. This is followed in Section 3 by an analysis that identifies the communication patterns that must be created by a protocol in runs where d learns the value of v_s Theorem 11. These structures, called *resilient message blocks*, can involve multiple enhanced message chains that must be arranged in a particular manner. The analysis is applied to characterize necessary and sufficient conditions on the communication patterns solving Nice-run Information Transfer in Section 4. Based on Section 3 and the results of application of Section 4, necessary and sufficient conditions for solving the Ordered Response coordination problem are presented in Section 5. Finally, patterns solving Robust Information Transfer are characterized in Section 6. A brief conclusion is presented in Section 7. Due to space restrictions, the proofs for Sections 5 and 6 appear in the paper's Arxiv version [17].

1.1 Related Work

Lamport's seminal paper [12] focuses on the role of message chains in asynchronous message passing systems. Indeed, Chandy and Misra showed in [4] that the only way in which knowledge about the state of the system at remote sites can be gained in asynchronous systems is via the construction of message chains. As mentioned above, in his later paper [13], Lamport points out that in synchronous systems information can also be conveyed using null messages. In a more recent paper [3], Ben Zvi and Moses analyzed knowledge gain and coordination in a model in which processes are reliable (no process ever crashes) and share a global clock, and there are upper bounds (possibly greater than 1) on message transmission times along each of the channels in the network. They extend the notion of a message chain to so-called *syncausal* message chains, which are sequences consisting of a combination of time intervals that correspond to the upper bounds and actual messages. They show that syncausal chains are necessary and sufficient for point-to-point information transfer when

$f = 0$. Moreover, they define a coordination problem called Ordered Response (which we revisit in Section 5) and show that a communication pattern they call a *centipede*, which generalizes message chains for their model, is necessary and sufficient for solving this problem. As mentioned in the Introduction and as will be defined in Section 2, not sending a message does not always count as a null message, even if message delays are bounded. The notion of an enhanced message chain thus refines that of a syncausal message chain: Every enhanced message chain is a syncausal message chain, but the converse is not true.

Our paper extends the work [7]. Their Silent Choir Theorem, discussed in the previous section, gives necessary conditions that are not sufficient even for failure-free executions. In the current paper we take the further step of characterizing necessary and sufficient properties of communication patterns that solve this problem, and investigate the role of silence in the more general coordination problem of Ordered Response coordination problem. None of the previous works (e.g., [13, 3, 7]) requires not sending a message to be informative in order to count as a null message. Making this requirement plays a technically significant role in the analyses performed in the current paper.

In Sections 4–6 we consider the design of protocols that are required to behave in a good way in the common case, which in this case is when the initial values are appropriate and no failures occur. Focusing on the design of protocols that are optimized for the common case has a long tradition in distributed computing (see, e.g., [15, 11, 14]). In our synchronous model Amdur, Weber, Hadzilacos and Halpern use them in order to design efficient protocols for Byzantine agreement [1, 10]. Guerraoui and Wang and others use them for Atomic Commitment [9, 7]. Solutions for Consensus in a synchronous Byzantine model optimized for the common case appeared in [8], also making explicit use of null messages. However, how null messages can be used for information transfer and coordination has not been characterized in a formal way.

2 Model and Preliminary Results

We follow the modeling of [7]. We consider a standard synchronous message-passing model with a set $\mathbb{P}$ of $N > 2$ processes and benign crash failures. For convenience, one of the processes will be denoted by s and called the *source*, while another, d can be considered as the *destination*. Processes are connected via a communication network defined by a directed graph $(\mathbb{P}, \mathsf{ch})$ where an edge from process i to process j is called a *channel*, and denoted by $\mathsf{ch}_{i,j}$. We assume that the receiver of a message detects the channel over which it was delivered, and thus knows the identity of the sender. The model is *synchronous*: All processes share a discrete global clock that starts at time 0 and advances by increments of one. Communication in the system proceeds in a sequence of *rounds*, with round $m+1$ taking place between time m and time $m+1$, for $m \geq 0$. A message sent at time m (i.e., in round $m+1$) from a process i to j will reach j at time $m+1$, i.e., at the end of round $m+1$. In every round, each process performs local computations, sends a set of messages to other processes, and finally receives messages sent to it by other processes during the same round. At any given time $m \geq 0$, a process is in a well-defined ***local state***. We denote by $r_i(m)$ the local state of process i at time m in the run r. For simplicity, we assume that the local state of a process i consists of its initial value v_i, the current time m, and the sequence of the events that i has observed (including the messages it has sent and received) up to that time. In particular, its local state at time 0 has the form $(v_i, 0, \{\})$. We focus on deterministic protocols, so a ***protocol*** Q describes what messages a process should send and what decisions it should take, as a function of its local state.

Processes in our model are prone to crash failures. A faulty process in a given execution fails by *crashing* at a given time. A process that crashes at time t is completely inactive from time $t+1$ on, and so it performs no actions and in particular sends no messages in round $t+2$ and in all later rounds. It behaves correctly up to and including round t. Finally, in round $t+1$ (which takes place between time t and time $t+1$) this process sends a (possibly strict) subset of the messages prescribed by the protocol. For ease of exposition we assume that the process does not perform any additional local actions (e.g., decisions) in round $t+1$.

For ease of exposition, we say that a process that has not failed up to and including time t is *active* at time t. We will consider the design of protocols that are required to tolerate up to f crashes. We denote by γ^f the model described above in which no more than f processes crash in any given run. We assume that a protocol has access to the values of N and f as well as to the communication network $(\mathbb{P}, \mathsf{ch})$. A ***run*** is a description of a (possibly infinite) execution of the system. We call a set of runs a ***system***. We will be interested in systems of the form $R_Q = R(Q, \gamma^f)$ consisting of all runs of a given protocol Q in which no more than f processes fail. A failure pattern determines who fails in the run, and what messages it succeeds in sending when it fails. Formally, we define:

▶ **Definition 1** (Failure patterns). *A* failure pattern *for a model γ^f is a set*

$$FP \triangleq \{\langle q_1, t_1, Bl(q_1)\rangle, \ldots \langle q_k, t_k, Bl(q_k)\rangle\}$$

of $k \le f$ triples, where $q_i \in \mathbb{P}$, $t_i \ge 0$, and $Bl(q_i) \subseteq \mathbb{P}$ for $i = 1, \ldots, k$. We consider a run r to be compatible with *a failure pattern $FP = \{\langle q_1, t_1, Bl(q_1)\rangle, \ldots \langle q_k, t_k, Bl(q_k)\rangle\}$ if*

1. *each process q_i fails in r at time t_i and only the processes $q_1, \ldots, q_k$ fail in r, while*
2. *for every process p to whom q_i should send a message in round $t_i + 1$ according to Q (based on q_i's local state at time t_i in r), a message from q_i to p is sent at time t_i in r iff $p \notin Bl(q_i)$.*

If a process q_i is specified as failing in a pattern FP, then for every $p \in Bl(q_i)$, we consider the channel $\mathsf{ch}_{q_i,p}$ to be *blocked* from round $t_i + 1$ on.

For ease of exposition in this paper we will restrict our attention to the case in which the source s has a binary initial value $v_s \in \{0, 1\}$, while the initial values of all other processes $p \ne s$ are fixed. Thus, there are only two distinct initial global states in a system R_Q. Moreover, the deterministic protocol Q, a given initial global state (in our case the value of v_s) and a failure pattern uniquely determine a run. Relaxing these assumptions would not modify our results in a significant way; it would only make proofs quite a bit more cumbersome.

2.1 Defining Knowledge

Our analysis makes use of a formal theory of knowledge in distributed systems. We sketch the theory here; see [6] for more details and a general introduction to the topic. In general, a process i can be in the same local state in different runs of the same protocol. We shall say that two runs r and r' are *indistinguishable* to process i at time m if $r_i(m) = r'_i(m)$. The current time m is represented in the local state $r_i(m)$, and so, $r_i(m) = r'_i(m')$ can hold only if $m = m'$. Notice that since we assume that processes follow deterministic protocols, if $r_i(m) = r'_i(m)$ then process i is guaranteed to perform the same actions at time m in both r and r' if it is active at time m.

▶ **Definition 2** (Knowledge). *Fix a system R, a run $r \in R$, a process i and a fact φ. We say that $K_i\varphi$ (which we read as "process i **knows** φ") holds at time m in r iff φ is true at time m at all runs $r' \in R$ such that $r_i(m) = r'_i(m)$.*

Definition 2 immediately implies the so-called *Knowledge property*: If $K_i\varphi$ holds at time m in r, then so does φ. The logical notation for "the fact φ holds at time m in the run r with respect to the system R" is $(R, r, m) \vDash \varphi$. Often, the system is clear from context and is not stated explicitly. In this paper, the system will typically consist of all the runs of a given protocol Q in the current model of computation, which we denote by R_Q. Observe that knowledge can change over time. Thus, for example, $K_j(v_i = 1)$ may be false at time m in a run r and true at time $m + 1$, based perhaps on messages that j does or does not receive in round $m + 1$.

An essential connection between knowledge and action in distributed protocols, called the ***knowledge of preconditions principle*** (KoP), is provided in [16]. It states that whatever must be true whenever a particular action is performed by a process i must be known by i when the action is performed. More formally, we say that a fact φ is a ***necessary condition*** for an action α in a system R if for all runs $r \in R$ and times m, if α is performed at time m in r then φ must be true at time m in r. In our model the KoP can be stated as follows:

▶ **Theorem 3** (KoP [16]). *Fix a protocol Q for γ^f and let α be an action of process i in R_Q. If φ is a necessary condition for α in R_Q then $K_i\varphi$ is a necessary condition for α in R_Q.*

As observed in [2], in synchronous systems the passage of time can provide a process information about events at remote sites. (E.g., p can know that q performs an action at some specific time, based purely on the protocol.) In order to focus on genuine flow of information between processes, we make the following definition:

▶ **Definition 4.** Information transfer *(*IT*) between s and d is achieved when $K_d(v_s = b)$ holds, for some value $b \in \{0, 1\}$.*

Since the initial value v_s is independent of the protocol, for d to learn this value requires genuine flow of information from s to d.

2.2 Null Messages and Enhanced Message Chains

As discussed in the Introduction, if no message is ever sent over a given channel at time t under the protocol Q, then the absence of such a message in a given execution is not informative. We now define not sending to be a null message only if it is informative:

▶ **Definition 5.** *Let r be a run of some protocol Q. Process* ***i* sends *j* a null message** *at (r, t) if*

- $\mathsf{ch}_{i,j}$ *is not blocked at (r, t),*
- *i does not send an actual message over $\mathsf{ch}_{i,j}$ at (r, t), and*
- *there is a run r' of Q in which i sends an actual message over $\mathsf{ch}_{i,j}$ at (r', t).*

We can now generalize message chains to allow for null messages as well as actual ones:

▶ **Notation 1.** *We denote by $\theta = \langle p, t\rangle$ the* **process-time** *pair consisting of a process p and time t. Such a pair is used to refer to the point at time t on p's timeline.*

▶ **Definition 6.** *Let r be a run of a protocol Q. We say that there is* **an enhanced message chain** *from $\theta = \langle p, t\rangle$ to $\theta' = \langle q, t'\rangle$ in r, and write $\boldsymbol{\theta \rightsquigarrow_{Q,r} \theta'}$, if there exist processes $p = i_1, i_2 \dots, i_k = q$ and times $t \le t_1 < t_2 < \cdots < t_k = t'$ such that for all $1 \le h < k$ process i_h sends either an actual message or a null message to i_{h+1} at (r, t_h). (We omit the subscript and write simply $\boldsymbol{\theta \rightsquigarrow \theta'}$ when Q and r are clear from the context.)*

Observe that Figure 4 contains three enhanced message chains between process s and d. Two of them contain a single actual message each, and one does not contain any actual message.

We are now ready to show that information transfer in synchronous systems requires the existence of an enhanced message chain.

▶ **Theorem 7.** *Let $f \geq 0$ and let Q be a protocol and $r \in R(Q, \gamma^f)$. Then $K_d(v_s = 1)$ holds at (r, m) only if $\langle s, 0\rangle \rightsquigarrow \langle d, m\rangle$ in r.*

Proof. Assume, by way of contradiction, that $K_d(v_s = 1)$ at (r, m), and that $\langle s, 0\rangle \not\rightsquigarrow \langle d, m\rangle$ in r. By Definition 2 it suffices to show a run $r' \in R(Q, \gamma^f)$ in which $v_s \neq 1$ such that $r_d(m) = r'_d(m)$. Denote

$$T \triangleq \{\theta \in \mathbb{V} \colon \langle s, 0\rangle \not\rightsquigarrow \theta \text{ in } r\}.$$

I.e., T is the set of nodes to which there is no enhanced message chain from $\langle s, 0\rangle$ in the run r. Observe that, by assumption, $\langle d, m\rangle \in T$. We construct a run r' as follows: The initial global state $r'(0)$ differs from $r(0)$ only in the value of the variable v_s (thus, $v_s = 0$ in r'), which appears in s's local state. All other initial local states are the same in $r'(0)$ and in $r(0)$. Finally, all processes have the same failure patterns in both runs. We now prove by induction on time t that $r_i(t) = r'_i(t)$ holds for all nodes $\langle i, t\rangle \in T$.

Base: $t = 0$. Assume that $\langle i, 0\rangle \in T$. By definition of T, it follows that $i \neq s$, and by construction of r' we immediately have that $r_i(0) = r'_i(0)$, as required.

Step. Let $t > 0$ and assume that the claim holds for all nodes $\langle j, t'\rangle$ with $t' < t$. Fix a node $\langle i, t\rangle \in T$. Clearly, $\langle i, t-1\rangle \in T$, and so by the inductive hypothesis $r_i(t-1) = r'_i(t-1)$. To establish our claim regarding $\langle i, t\rangle$, it suffices to show that i receives exactly the same messages at time t in both runs. Recall that the synchrony of the model implies that the only messages that i can receive at time t are ones sent at time $t-1$. Hence, we reason by cases, showing that every process $z \neq i$ sends i the same messages at time $t-1$ in both runs.

- Suppose that $\langle z, t-1\rangle \in T$. Then by the inductive assumption $r_z(t-1) = r'_z(t-1)$, i.e., process z has the same local state at time $t-1$ in both runs. Since Q is deterministic and since the runs r and r' have identical failure patterns, z sends i a message in r at time $t-1$ in r' iff it does so in r. Moreover, if it sends a message, it sends the same message in both cases.
- Suppose that $\langle z, t-1\rangle \notin T$, i.e., there is an enhanced message chain from $\langle s, 0\rangle$ to $\langle z, t-1\rangle$ in r. Since $\langle i, t\rangle \in T$ we have that z does not send a message to i at time $t-1$ in r. Assume by way of contradiction that i receives such a message in r'. In particular, this implies that the channel $\mathsf{ch}_{z,i}$ is not blocked in r', and since r and r' have the same failure pattern, $\mathsf{ch}_{z,i}$ is not blocked in r. Hence, by definition, there is a null message from $\langle z, t-1\rangle$ to $\langle i, t\rangle$ in r. This contradicts the fact that, by assumption, $\langle i, t\rangle \in T$. It follows that, in both r and r', process i does not receive any message from z at time t.

Since $r_i(t-1) = r'_i(t-1)$ process i performs the same actions at time $t-1$ in both runs. Since, in addition, i receives exactly the same messages at (r', t) as it does in (r, t) as we have shown, it follows that $r_i(t) = r'_i(t)$.

The inductive argument above showed that, for all processes i and all times $t \leq m$, if i is has not failed by time t and $\langle i, t\rangle \in T$, then $r_i(t) = r'_i(t)$. Since $\langle d, m\rangle \in T$ by assumption, it follows that, in particular, $r_d(m) = r'_d(m)$. Since $v_s \neq 1$ in r' we obtain that $\neg K_d(v_s = 1)$ at time m in r by Definition 2. This contradicts the assumption that $K_d(v_s = 1)$ holds at time m in r, completing the proof. ◀

Theorem 7 establishes that enhanced message chains are necessary for information transfer for all values of $f \geq 0$. In fact, when $f = 0$, enhanced message chains are also sufficient. (We omit a proof of this particular claim since it will follow from the more general Theorem 16). This demonstrates that enhanced message chains play an analogous role in reliable synchronous settings to the one that standard message chains play in asynchronous systems (cf. [4]).

In reliable systems, null messages are detected as reliably as actual messages are. As discussed in the Introduction, however, this is no longer true in the presence of failures. The knowledge formalism allows us to crisply capture a stronger requirement than the one in Theorem 7, which lies at the heart of the issue.

▶ **Theorem 8.** *Let $f \geq 0$, let r be a run of $R_Q = R(Q, \gamma^f)$ and denote $\theta_s = \langle s, 0\rangle$ and $\theta_d = \langle d, m\rangle$. Then $K_d(v_s = 1)$ holds at (r, m) only if $K_d(\theta_s \rightsquigarrow \theta_d)$ holds at (r, m).*

Proof. Suppose that $(R_Q, r, m) \vDash K_d(v_s = 1)$. Definition 2 implies that $(R_Q, r', m) \vDash K_d(v_s = 1)$ holds for every run r' such that $r'_d(m) = r_d(m)$. By Theorem 7 it follows that $(R_Q, r', m) \vDash (\theta_s \rightsquigarrow \theta_d)$ for every such r', and so by Definition 2 we obtain that $(R_Q, r, m) \vDash K_d(\theta_s \rightsquigarrow \theta_d)$, as claimed. ◀

3 Dealing with Failures

The need to know that an enhanced chain has reached d, established in Theorem 8, is not the same as the mere existence of such a chain. This difference matters when processes may fail, because then silence can be ambiguous, and null messages can be confused with process crashes. How, then, can d come to know that an enhanced chain has reached it, in a setting where $f > 0$ processes can crash? One possibility would be to have the protocol construct $f + 1$ enhanced chains from $\langle s, 0\rangle$ to $\langle d, m\rangle$ whose sets of participating processes are pairwise disjoint.[1] While such an assumption may be needed in a protocol that in a precise sense guarantees information transfer (we revisit this point in Section 6), in many instances the destination process can learn the sender's value even if the protocol does not employ such a scheme.

Our purpose is to investigate the communication patterns under which d can learn the value of v_s. For this purpose, we will find it convenient to associate a "communication graph" with every run, which we define as follows. The nodes of the graph are process-time pairs. Edges correspond to messages sent among processes, to null messages, and a local tick of the clock at a process. More formally:

▶ **Definition 9** (Communication Graphs). *The* communication graph *of a run r of protocol Q is $\mathsf{CG}_Q(r) \triangleq (\mathbb{V}, E)$, with nodes $\mathbb{V} = \mathbb{P} \times \mathbb{N}$ and edges $E = E_{\mathsf{l}} \cup E_{\mathsf{a}}(r) \cup E_{\mathsf{n}}(r)$, where*

- $E_{\mathsf{l}} = \{(\langle i, t\rangle, \langle i, t + 1\rangle) : i \in \mathbb{P}, t \in \mathbb{N}\}$,
- $E_{\mathsf{a}}(r) = \{(\langle i, t\rangle, \langle j, t + 1\rangle) :$ *i sends an actual message to j at time t in r*$\}$,
- $E_{\mathsf{n}}(r) = \{(\langle i, t\rangle, \langle j, t + 1\rangle) :$ *i sends a null message to j at time t in r*$\}$

Notice that both the set of nodes $\mathbb{V}$ and the set E_{l} of local edges are the same in all communication graphs. Observe that the communication graph directly represents enhanced message chains: $\boldsymbol{\theta \rightsquigarrow_{Q,r} \theta'}$ holds if and only if $\mathsf{CG}_Q(r)$ contains a path from θ to θ'.

[1] A similar issue arises in the Byzantine Agreement literature (cf. [5]) where many process-disjoint chains are used to overcome the possibility of failures.

3.1 Resilient Message Blocks

The Silent Choir Theorem of [7] states that a necessary condition for $K_d(v_s = 1)$ to hold at time m without an actual chain from s to d, is for there to be actual message chains to $f + 1$ members of the silent choir, after which they are all silent to d at time $m - 1$. As discussed in the Introduction, however, these members need not send d a *null message* at $m - 1$. Indeed, members of the "choir" need not even be neighbors of d. In this section we will present a strictly stronger condition on the communication pattern than the one in their theorem, called a resilient message block. The new condition will also be more informative as it is explicitly formulated in terms of null messages. Moreover, for the interesting case of optimizing communication for failure-free executions, our resilient message blocks will be both necessary *and* sufficient for information transfer.

Very roughly speaking, a process that knows about failures might detect the existence of an enhanced chain more easily than one who is unaware of failures. E.g., if d has detected all f faulty processes, then it can readily detect null messages sent by correct processes. Correctly coping with such issues requires a somewhat subtle definition and theorem statement.

▶ **Notation 2** (B null free paths)**.** *Fix a protocol Q, let r be a run of Q, and let B be a set of processes. A path π in $\mathsf{CG}_Q(r)$ that does not contain null messages sent by processes in the set B is called B* null free *(we write that "π is $B_{\not\eta}$" for brevity).*

A $B_{\not\eta}$ path can contain null messages, but not ones "sent" by members of B. Roughly speaking, if the members of B crash, this path can remain a legal enhanced message chain. In light of Theorem 8, we are now ready to characterize the properties of communication graphs of protocols that enable information transfer. Using $B_{\not\eta}$ paths we can now define the central communication patterns that will play a role in our analysis:

▶ **Definition 10** ((f/failed)-resilient message block)**.** *Let r be a run of a protocol Q and denote by $\mathbb{F}^r$ the set of processes that fail in r. Let $\theta, \theta' \in \mathbb{P} \times \mathbb{N}$ be two nodes. An* (f/failed)-resilient message block *from θ to θ' in $\mathsf{CG}_Q(r)$ is a set Γ of paths between θ and θ' such that for every set of processes B such that $|B \cup \mathbb{F}^r| \leq f$, there is a $B_{\not\eta}$ path in $\mathsf{CG}_Q(r)$ from θ to θ' in Γ.*

Recall that an actual message chain contains no null messages, and is thus a $B_{\not\eta}$ path for every set B of processes. As a result, an actual message chain between two nodes is, in particular, an f/failed-resilient message block, for all runs and *all* values of $f \geq 0$.

Our next theorem states that in order for a process d to know in some run r at time m that there is an enhanced message chain from some node to itself, there must be a resilient message block between them. Roughly speaking, the claim is proved by way of contradiction. We assume a set B of processes contradicting the assumption and construct a run r' that is indistinguishable to d from r in which nodes involving processes of B cut all paths from $\theta_s = \langle s, 0\rangle$ to $\theta_d = \langle d, m\rangle$. I.e., there is no enhanced message chain from θ_s to θ_d in r'. More precisely, given a set B of processes we will define the set T_B to be the set of nodes to which there is *no* $B_{\not\eta}$ path from θ_s. We give an example of such a set in Figure 5. The highlighted nodes are in T_B while the others are not in T_B.[2] As detailed in the complete proof, when constructing r', we make the processes of B fail at times that make these failures unnoticeable by processes appearing in T_B (and hence by the contradiction assumption neither by d at

[2] For the sake of clarity we do not draw all of the nodes and edges of the communication graph. Thus, for example, we do not represent all the nodes along local time lines and the local edges in E_l that connect them.

time m). As a result, process d does not know at (r, m) that an enhanced message chain has reached it. Since by Theorem 7, this is a necessary condition, we conclude that $\neg K_d(v_s = 1)$ at (r, m), as claimed. We can now show:

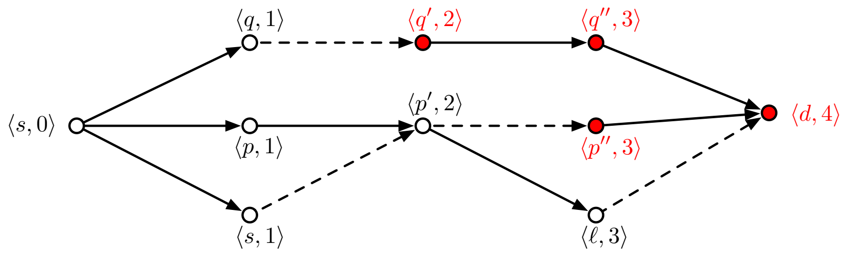


Figure 5 A communication graph and its corresponding set T_B (highlighted) for $B = \{q, p', \ell\}$.

▶ **Theorem 11.** *Let r be a run of a given protocol Q and let $\mathbb{F}^r$ denote the set of faulty processes in r. If $K_d(v_s = 1)$ holds at (r, m), there is an (f/failed)-resilient message block from $\theta_s \triangleq \langle s, 0\rangle$ to $\theta_d \triangleq \langle d, m\rangle$ in $\mathsf{CG}_Q(r)$.*

The complete proof appears in the Appendix.[3] We point out that Theorem 11 is stronger than the Silent Choir Theorem of [7]. Namely, as we claim in Lemma 12 the existence of the depicted resilient message block implies that a silent choir exists. However, the converse is not true. Remember the example of Figure 1. The set of processes $\{s, p\}$ is a silent choir for $f = 1$ but clearly, there is no f-resilient message block – take for instance $B = \{p\}$, there is no $B_{\not{\!\!d}}$ path from $\langle s, 0\rangle$ to $\langle d, 2\rangle$.

As the following lemma establishes, Theorem 11 implies the Silent Choir Theorem:

▶ **Lemma 12.** *If there is an f/failed-resilient message block from θ to θ' in $CG_Q(r)$, then there exists a silent choir from θ to θ' in r.*

We are now ready to characterize the communication structures needed to solve information transfer and coordination in several interesting cases.

4 Application: Information Transfer in Nice Runs

Theorem 11 is at the heart of information transfer in fault-prone synchronous systems. f-resilient message blocks constitute a necessary pattern for information transfer in our model since without them, one cannot know that an enhanced message chain reaches it. Clearly, in a system prone to crash failures, it is natural to require for information to be conveyed in failure-free runs. Focusing on the design of protocols that are optimized for the common case has a long tradition in distributed computing and can be useful in applications including Consensus, Atomic Commitment, and blockchain protocols (see, e.g., [1, 10, 15, 11, 14, 8]).

Clearly, the information that a null message conveys depends crucially on the protocol. More precisely, it depends on the conditions under which i would not send an *actual* message. To account for this, we make the following definition:

[3] Every claim is completely proved either in the main text,in the Appendix or in the full version of the paper [17].

▶ **Definition 13.** *We say that process i* **sends a null message** *over* $\mathsf{ch}_{i,j}$ *at time t* **in case φ in** *a given protocol* $\boldsymbol{Q}$ *if for every run $r \in R_Q$ in which* $\mathsf{ch}_{i,j}$ *is not blocked at (r,t), process i sends a null message over* $\mathsf{ch}_{i,j}$ *at (r,t) iff $(R_Q, r, t) \vDash \varphi$.*

Roughly speaking, a process p "sends a null message in case φ" at time t if whenever it is active at time t and φ does not hold, then p sends an actual message. If φ does hold at time t, then p keeps silent. By definition, a null message is sent in case φ only if φ is true. Therefore, in a reliable system (i.e., when $f = 0$), sending a null message in case φ informs the recipient that φ holds.

We focus on solving the IT problem in *nice runs*, which are formally defined as follows:

▶ **Definition 14.** *Let Q be a protocol. The run of Q in which $v_s = 1$ and no process fails is called* $\boldsymbol{Q}$*'s* **nice run**. *We denote Q's nice run by* $\boldsymbol{\hat{r}(Q)}$ *and the communication graph of $\hat{r}(Q)$ by* $\mathsf{nG}(\boldsymbol{Q})$. *When Q is clear from the context, we simply write* $\boldsymbol{\hat{r}}$.

Every protocol Q has a unique nice run. We show in this section that the conditions of Theorem 11 are also sufficient for information transfer in *nice* runs. Clearly, in a failure-free execution r, it holds that $\mathbb{F}^r = \emptyset$. An f/failed-resilient message block with $\mathbb{F}^r = \emptyset$ is a central structure for information transfer in nice runs and will be used in Section 5. It can be defined in slightly simpler terms as follows:

▶ **Definition 15** (f-resilient message block)**.** *Let $\theta, \theta' \in \mathbb{P} \times \mathbb{N}$ be two nodes. An* f-resilient message block from θ to θ' in $\mathsf{CG}_Q(r)$ *is a set Γ of paths between θ and θ' such that for every set of processes B of size $|B| \leq f$, there is a $B_{\not\gamma}$ path in $\mathsf{CG}_Q(r)$ from θ to θ' in Γ.*

Observe that the presence of an f-resilient message block in the communication graph of a run r suffices to ensure that in any run r' that "looks the same" to d at (r', m) i.e., $r_d(m) = r'_d(m)$, there is an enhanced message chain reaching d.

▶ **Theorem 16** (Nice-run IT)**.** *f-resilient message blocks are necessary and sufficient for solving* IT *in the nice run. Namely,*

- *(Necessity) If $K_d(v_s = 1)$ at $(\hat{r}, m)$ then there is an f-resilient message block from θ_s to θ_d in* nG.
- *(Sufficiency) If a communication graph* CG *contains an f-resilient message block between θ_s and θ_d, then there exists a protocol Q such that $\mathsf{nG}(Q) = \mathsf{CG}$ that solves* IT *between θ_s and θ_d.*

Theorem 16 gives a *tight* characterization of the communication patterns needed to solve IT in nice runs: Every solution must construct an f-resilient message block, and for every f-resilient message block, there exists an IT protocol that uses only the paths in this block in its nice run.

The necessity part of Theorem 16 results from Theorem 11 applied to $\hat{r}$ where the set of faulty processes is $\mathbb{F}^r = \emptyset$. To show sufficiency, we need to describe a protocol Q as claimed. Before sketching the proof, we discuss and define a class of protocols used in the proof.

In protocols solving IT in the nice run, messages only need to convey whether the sender has detected that the run is not nice. To consider this more formally, for a given protocol Q we denote by ψ_{nice} the fact "the current run is $\hat{r}$." Typically, if $f > 0$, it is impossible for a process to know that ψ_{nice} is true: Even if a process has observed no failures, or indeed, even if no failures have occurred by a given time t, there may be run indistinguishable to the process in which one or more processes fail after time t. Nevertheless, a process may readily know that $\neg\psi_{nice}$, if it knows of a failure or detects that $v_s = 0$. Of course, because of the Knowledge property, in the nice run $\hat{r}$ itself, no process will ever know $\neg\psi_{nice}$.

▶ **Definition 17** (Nice-based Message Protocols). *A protocol Q is a Nice-based Message protocol (*NbM *protocol) if (i) All actual messages sent are single bit messages and whenever a process p sends an actual message, it sends a '0' if $K_p\neg\psi_{nice}$ and sends a '1' otherwise (i.e., if $\neg K_p\neg\psi_{nice}$) and (ii) for all processes p, each null message sent by p over any channel is a null message in case $\neg K_p\neg\psi_{nice}$.*

We remark that a process p can efficiently check whether $K_p(\neg\psi_{nice})$ by simply comparing p's local state at (r,t) to its local state at $(\hat{r},t)$. If the two states are identical, then the predicate $K_p(\neg\psi_{nice})$ is false. Otherwise, it is true.

Proof sketch (of sufficiency in Theorem 16). Suppose that CG contains an f-resilient message block between θ_s and θ_d. The desired protocol Q is defined to be a Nice-based Message protocol such that $\mathsf{nG}(Q) = \mathsf{CG}$. I.e., for every edge $e \triangleq (\langle u,t\rangle, \langle v,t+1\rangle)$ in CG: If $e \in E_{\mathsf{n}}$, then u keeps silent if $\neg K(\neg\psi_{nice})$ and should send a '0' to v otherwise. If $e \in E_{\mathsf{a}}$, then u should send '1' to v if $\neg K_u(\neg\psi_{nice})$ and '0' otherwise. We show that the assumptions guarantee that for every run r of Q there is (at least one) path in $\mathsf{nG}(Q)$ from θ_s to θ_d along which no silent process fails in r. We show by induction on time that in every run r in which $v_s = 0$, for each node $\langle p,t\rangle$ along this path, p knows at time (r,t) that the run is not nice. Hence, d also knows at (r,m) that the run is not nice. Since ψ_{nice} holds throughout $\hat{r}$, so does $\neg K_d\neg\psi_{nice}$. It follows that $K_d(v_s = 1)$ at $(\hat{r},m)$, as claimed. ◀

5 Application: Coordination

f-resilient message blocks and Nice-run IT are useful tools for solving more complex problems. We now show how they can be used to characterize solutions to the Ordered Response (O-R) coordination problem. This problem was originally defined in [3], and it requires a sequence of actions to be performed in linear temporal order, in response to a triggering signal from the environment.[4] For simplicity, we identify the signal to be received iff $v_s = 1$. We assume that each process $i_h \in \{i_1, i_2, \ldots, i_k\}$ has a specific action a_h to perform, and that the actions should be performed in order, provided that initially $v_s = 1$.

▶ **Definition 18** (Ordered Response). *We say that a protocol Q is **consistent** with the instance* OR $= \langle v_s = 1, a_1, \ldots, a_k\rangle$ *of the* Ordered Response (O-R) *problem if it guarantees that a_h is performed in a run only if $v_s = 1$ and $a_1, \ldots, a_{h-1}$ are performed. In particular, if both a_{h-1} and a_h are performed at times t_{h-1} and t_h respectively, then $t_{h-1} \le t_h$. Protocol Q **solves** this instance* OR *if, in addition, all of the actions a_h are performed in Q's nice run.*

Let us denote by $\underline{a}_h$ the fact that the action a_h has (already) been performed. Since, by definition of O-R, both $v_s = 1$ and $\underline{a}_{h-1}$ are necessary conditions for performing a_h, the Knowledge of Preconditions principle (Theorem 3) implies that these facts must be known when $\underline{a}_h$ is performed:

▶ **Lemma 19.** *Suppose that Q solves the instance* OR $= \langle v_s = 1, a_1, \ldots, a_k\rangle$ *of* O-R. *For every run r of Q and action a_h performed (at time t_h) in r, we have*
1. $(R_Q, r, t_h) \vDash K_{i_h}(v_s = 1)$, *and*
2. $(R_Q, r, t_h) \vDash K_{i_h}(\underline{a}_{h-1})$ *if $h > 1$.*

[4] In [3] Ordered Response was studied in a reliable setting with no crashes, and upper bounds on message delivery times; a very different set of assumptions than here.

For a protocol Q solving an instance of Ordered Response, every action a_h, is performed at some specific time t_h in the nice run $\hat{r} = \hat{r}(Q)$. For ease of exposition we denote by by $\theta_h \triangleq \langle i_h, t_h \rangle$ the node of $\mathsf{nG}(Q)$ where the action is taken, and by $\theta_h^+ \triangleq \langle i_h, t_h + 1 \rangle$ the node of i_h one time step after the node θ_h.

We can use Lemma 19 to provide necessary conditions on the nice communication graph of protocols that solve Ordered Response. Lemma 19(1) implies that Q must perform Nice-run IT to θ_h, for all actions a_h. Lemma 19(2), in turn, implies that i_h needs to learn that a_{h-1} has been performed in order to perform its action. A straightforward way to do this is by performing Nice-run IT directly between consecutive actions, i.e., by creating an f-resilient block between θ_{h-1} and θ_h. While this is a possible solution, it is *not* the only way that i_h can obtain this knowledge. Process i_h can also learn about the previous action *indirectly*. This requires i_h to know that i_{h-1} couldn't have failed (since otherwise i_h would have an indication that it failed) and that it received the information needed to perform a_{h-1}. The possibility of acting on indirect information is where solving the Ordered Response problem goes beyond Nice-run IT. We can show the following:[5]

▶ **Theorem 20** (O-R Necessity). *Let Q be a protocol solving $\mathsf{OR} = \langle v_s = 1, a_1, \ldots, a_k \rangle$ for some sequence of times $t_1 \leq t_2 \leq \ldots \leq t_k$. Then $\mathsf{nG}(Q)$ must contain the following blocks:*
1. *An f-resilient message block between θ_s and θ_x, for each $x \leq k$; and*
2. *An $(f-1)$-resilient message block between θ_{x-1}^+ and θ_x that does not contain null messages sent by i_{x-1}, for each $1 < x \leq k$.*

Observe that Item 2 of Theorem 20 implies the existence of an $(f-1)$-resilient message block in which i_h does not send null messages. This requirement is weaker than the existence of an f-resilient message block. While the conditions for $\mathsf{O\text{-}R}$ stated in Theorem 20 are necessary, they are not sufficient in general. Indeed, it is unclear what conditions on $\mathsf{nG}(Q)$ might be sufficient to solve $\mathsf{O\text{-}R}$ for general protocols Q. In the Appendix, we present a natural class of protocols for which conditions that are both necessary and sufficient can be stated and proven (see Theorems 31 and 32).

6 Robust Information Transfer

We now turn to consider protocols that convey information in a "robust" way. Namely, they ensure that in every run in which $v_s = 1$ and the source process s does not fail, the destination eventually knows that $v_s = 1$.

▶ **Definition 21** (Robust Information Transfer). *A protocol Q is said to solve* Robust Information Transfer *between processes s and d if, for every run r of Q in which $v_s = 1$ and s does not fail, there is a time m such that $K_d(v_s = 1)$ holds at (r, m).*

Clearly, a protocol that solves the Robust Information Transfer problem also solves, in particular, IT in its nice run. However, Robust Information Transfer is a strictly harder problem and so, as shown in this section, its solutions require more communication than is allowed by f-resilient message blocks.

▶ **Theorem 22** (Robust IT Necessity). *Let Q be a protocol that solves Robust IT between s and d. Then, there exists $m \geq 0$ such that*
- $\mathsf{nG}(Q)$ *contains an actual message sent from s to d no later than at time $m - 1$, or*
- $\mathsf{nG}(Q)$ *contains $f + 1$ paths from $\theta_s = \langle s, 0 \rangle$ to $\theta_d = \langle d, m \rangle$ that are disjoint in message senders (except for s and d) such that in at least one of these paths, s does not send null messages.*

[5] Recall that all proofs for Sections 5 and 6 appear in [17].

Proof sketch. The claim is proved by way of contradiction assuming there exists no m as described in the Theorem. We then consider different cases according to the way the Theorem's assumptions are violated. For each case, we construct a run $r \in R_Q$ in which $v_s = 1$ and s does not fail as well as a corresponding run r' in which there is no enhanced message chain from θ_s to θ_d and that is indistinguishable by d. By Theorem 7, it results that $\neg K_d(v_s = 1)$ at (r, m), completing the proof. ◀

An interesting difference between the necessary conditions for solving the Nice-run IT and the ones for Robust IT is in that for the former, process disjointness is required only for the processes sending null messages, while for the latter it is required for all processes. This resembles the conditions of [5] where in order to solve Byzantine Agreement in the synchronous byzantine model, the communication network must have a connectivity of at least $2f + 1$. Thus, the paths from the source process s must be process disjoint the stronger sense.

We now show that the conditions of Theorem 22 are not only necessary, but also sufficient.

▶ **Theorem 23** (Robust IT Sufficiency). *Let* CG *be a communication graph satisfying the conditions of Theorem 22. Then there is a protocol Q with* $\mathsf{nG}(Q) = \mathsf{CG}$ *that solves the Robust* IT *problem.*

In analogy to how we defined Nice-based Message protocols in Section 4, we define in the Appendix (Definition 29) what we call Robust-based Message protocols – protocols in which whether messages are sent and the contents of the messages sent depend on whether a process knows that $(v_s = 0 \vee s \text{ failed})$. Taken together, Theorems 22 and 23 provide a tight characterization of the communication patterns of protocols solving Robust Information Transfer.

7 Conclusions

Every model of distributed computing provides particular means by which processes can communicate, and these can have a profound impact on the problems that can be solved in the model and on the form that protocols solving them will have. Synchronous systems with global clocks, for example, allow nontrivial use of null messages, which are completely meaningless in the asynchronous model, for example. Since a message not sent can be informative only if there are alternative conditions under which it would be sent, null messages are especially useful as a means of shifting communication costs to optimize for the common case. As illustrated in Figures 3 and 4 an demonstrated in the Atomic Commitment protocols of [7], shifting these costs in a careful way can result in significant savings.

By refining the definitions of null messages, we were able to investigate fundamental aspects of information transfer and coordination in synchronous systems with crash failures. In particular, we obtained characterizations of protocols that solve information transfer and coordination problems in nice, failure-free executions. A central tool in our analysis is the notion of an f-resilient message block, which is significantly more refined than the silent choirs of [7]. Indeed, while constructing silent choirs is a necessary condition on protocols solving information transfer in nice runs, constructing f-resilient blocks is both necessary *and* sufficient. For the Ordered Response coordination problem, where liveness needs to be guaranteed in nice runs, we obtain a condition based on resilient message blocks which, again, is both necessary and sufficient. No similar analysis of Ordered Response has been attempted in the literature.

References

1 Eugene S. Amdur, Samuel M. Weber, and Vassos Hadzilacos. On the message complexity of binary byzantine agreement under crash failures. *Distributed Computing*, 5(4):175–186, 1992.

2 Ido Ben-Zvi and Yoram Moses. On interactive knowledge with bounded communication. *Journal of Applied Non-Classical Logics*, 21(3-4):323–354, 2011. URL: `http://jancl.e-revues.com/article.jsp?articleId=17078`.

3 Ido Ben-Zvi and Yoram Moses. Beyond lamport's happened-before: On time bounds and the ordering of events in distributed systems. *Journal of the ACM (JACM)*, 61(2):1–26, 2014.

4 K. M. Chandy and J. Misra. How processes learn. *Distributed Computing*, 1(1):40–52, 1986.

5 Danny Dolev. The byzantine generals strike again. *Journal of algorithms*, 3(1):14–30, 1982.

6 Ronald Fagin, Joseph Y Halpern, Yoram Moses, and Moshe Y Vardi. *Reasoning About Knowledge*. MIT Press, 1995. `doi:10.7551/mitpress/5803.001.0001`.

7 Guy Goren and Yoram Moses. Silence. *J. ACM*, 67:3:1–3:26, 2020. `doi:10.1145/3377883`.

8 Guy Goren and Yoram Moses. Optimistically tuning synchronous byzantine consensus: another win for null messages. *Distributed Comput.*, 34(5):395–410, 2021. `doi:10.1007/s00446-021-00393-8`.

9 Rachid Guerraoui and Jingjing Wang. How fast can a distributed transaction commit? In Emanuel Sallinger, Jan Van den Bussche, and Floris Geerts, editors, *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 107–122. ACM, 2017. `doi:10.1145/3034786.3034799`.

10 Vassos Hadzilacos and Joseph Y. Halpern. Message-optimal protocols for byzantine agreement. *Mathematical Systems Theory*, 26(1):41–102, 1993.

11 Alex Kogan and Erez Petrank. A methodology for creating fast wait-free data structures. In *ACM SIGPLAN Notices*, volume 47, pages 141–150. ACM, 2012.

12 L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

13 Leslie Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Trans. Program. Lang. Syst.*, 6:254–280, 1984. `doi:10.1145/2993.2994`.

14 Kfir Lev-Ari, Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi. Fairledger: A fair blockchain protocol for financial institutions. In Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller, editors, *23rd International Conference on Principles of Distributed Systems, OPODIS 2019, December 17-19, 2019, Neuchâtel, Switzerland*, volume 153 of *LIPIcs*, pages 4:1–4:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. `doi:10.4230/LIPIcs.OPODIS.2019.4`.

15 Barbara Liskov. Practical uses of synchronized clocks in distributed systems. *Distributed Computing*, 6(4):211–219, 1993.

16 Yoram Moses. Relating knowledge and coordinated action: The knowledge of preconditions principle. In *Proceedings of TARK,*, pages 231–245, 2015. `doi:10.48550/arXiv.1606.07525`.

17 Raïssa Nataf, Guy Goren, and Yoram Moses. Null messages, information and coordination, 2023. `arXiv:2208.10866`.

Appendix

Additional details regarding Failure Patterns defined in Section 2

Our technical analysis is facilitated by defining a strictness ordering among failure patterns, and associating a minimal failure pattern with each run.

Notice that there may be several failure patterns that are compatible with a given run. In particular, blocking a channel along which no message should be sent does not affect processes' local states.

▶ **Definition 24** (Failure patterns comparison). *Let $FP = \{\langle q_i, t_i, Bl(q_i)\rangle\}_{i\le k}$ and $FP' = \{\langle q'_j, t'_j, Bl'(h'_j)\rangle\}_{j\le k'}$ be two failure patterns with $k, k' \le f$, and denote by F and F' the sets of processes that appear in these patterns, respectively. We say that FP' is* harsher *than FP (denoted by $FP' \le FP$) if $F \subseteq F'$ and for every $q_i \in F$:*

- *$t'_i < t_i$, or*
- *$t_i = t'_i$ and $Bl(q_i) \subseteq Bl'(q_i)$*

Note that processes that don't fail in r' might fail in r. We now define what we call a *minimal* failure pattern wrt. a run.

▶ **Definition 25.** *Let r be a run and let FP be a failure pattern compatible with r. We say that FP is minimal wrt. r if for every failure pattern FP' that is compatible with r and such that FP is harsher than FP' (i.e., $FP \le FP'$), it holds that $FP' = FP$.*

Clearly, for a given run r there is only one minimal compatible failure pattern. We denote it by $FP(r)$. In the proofs appearing in this section, we will be interested in comparing communication graphs of two runs of the same protocol. We compare the communication graphs regardless of the edges category. Formally:

▶ **Definition 26** (unlabeled-edge subgraph). *Let $\mathsf{CG} = (\mathbb{V}, E)$ and $\mathsf{CG}' = (\mathbb{V}, E')$ be two communication graphs. We say that CG is an "unlabeled-edge" subgraph of CG', and write $\mathsf{CG} \subseteq_u \mathsf{CG}'$ if for every edge $e \in E$ it is the case that $e \in E'$. (Although e can be an actual message edge in one graph and a null-message edge in the other.) In the rest of paper, we often write "subgraph" to stand in for "unlabeled-edge subgraph".*

We can now show:

▶ **Lemma 27.** *If $FP(r') \le FP(r)$ for two runs of a protocol Q, then $\mathsf{CG}_Q(r') \subseteq_u \mathsf{CG}_Q(r)$.*

Proof. Both graphs have the same set of nodes. We now prove that for each $e \in E_{\mathsf{l}} \cup E_{\mathsf{a}}(r') \cup E_{\mathsf{n}}(r')$, it holds that $e \in E_{\mathsf{l}} \cup E_{\mathsf{a}}(r) \cup E_{\mathsf{n}}(r)$.

- $E_{\mathsf{l}}(r) = E_{\mathsf{l}}(r')$.
- Let $e \triangleq (\langle p, t\rangle, \langle p', t+1\rangle) \in E_{\mathsf{a}}(r')$, i.e., p sends p' an actual message at (r', t). If p sends p' an actual message at (r, t) then $e \in E_{\mathsf{a}}(r)$. Otherwise, since $FP(r') \le FP(r)$, it holds that the channel $\mathsf{ch}_{p,p'}$ is not blocked at (r, t). Hence, by the definitions of communication graphs and of null messages, we have that $e \in E_{\mathsf{n}}(r)$. So $e \in E$.
- Finally, let $e \triangleq (\langle p, t\rangle, \langle p', t+1\rangle) \in E_{\mathsf{n}}(r')$. In particular, the channel $\mathsf{ch}_{p,p'}$ is not blocked at (r', t). Hence by null messages definition and the fact that $FP(r) \le FP(r')$ it holds that $e \in E_{\mathsf{a}}(r) \cup E_{\mathsf{n}}(r)$. ◀

Proof of Theorem 11

In the proof of Theorem 11, we will construct a run r' that is similar to r but in which we make fail additional processes. In order to ensure that these additional failures are not noticed by d (and hence to ensure that d does not distinguish r from r'), we make the processes fail at a specific *critical* time that we define as follows:

▶ **Definition 28** (Critical Time). *Let r be a run of Q, let B be a set of processes and let p be a process. For every pair θ and θ' of nodes of $\mathsf{CG}(Q)$, the* critical time *$t_p = t_p(\theta, \theta')$ wrt. $(\mathsf{CG}_Q(r), B)$ is defined to be the minimal time m_p such that $\mathsf{CG}_Q(r)$ contains a $B_{\not p}$ path from θ to $\langle p, m_p\rangle$ as well as a path from $\langle p, m_p\rangle$ to θ'. If no such time m_p exists, then $t_p = \infty$.*

Informally, the critical time of a process p represents the first time at which p can learn about an event local to s and may be able to inform d about this event. Making relevant processes fail at their critical times ensures that d does not notice these failures and hence that d does not distinguish the constructed run r' from the nice run. We can now prove Theorem 11:

Proof. Assume by way of contradiction that no such block exists in $\mathsf{CG}_Q(r)$. I.e., there exists a set B such that $|B \cup \mathbb{F}^r| \leq f$ and every path from θ_s to θ_d in $\mathsf{CG}_Q(r)$ contains a null message from a process in B. Let B be a minimal set (by set inclusion) with this property. Define the set T_B to be:

$$T_B \triangleq \{\langle p,t\rangle \in \mathbb{V} : \text{There is no } B_{\not\emptyset} \text{ path from } \theta_s \text{ to } \langle p,t\rangle \text{ in } \mathsf{CG}_Q(r)\}$$

Notice that our assumption about θ_d implies that $\theta_d \in T_B$. Moreover, observe that if $\langle i,t\rangle \in T_B$, then $\langle i,t'\rangle \in T_B$ for all earlier times $0 \leq t' < t$.

We show that there exists a run r' of Q such that $r'_d(m) = r_d(m)$ and there is no enhanced message chain from θ_s to θ_d in r'. This will contradict the fact that $K_d(\theta_s \rightsquigarrow \theta_d)$ holds at time m in r. We construct r' as follows: The initial global state is $r'(0) = r(0)$. Each process $b \in B$ crashes in r' at its critical time $t_b \triangleq t_b(\theta_s, \theta_d)$ wrt. $(\mathsf{CG}_Q(r), B)$ without sending any messages from time t_b on. Moreover, every process in $\mathbb{F}^r \backslash B$ crashes in precisely the same manner in r' as it does in r. By definition, the critical time of a process $p \in \mathbb{F}^r$ is necessarily smaller or equal to the actual time at which p fails in r. We hence have that $FP(r') \leq FP(r)$. Clearly, there is no path from θ_s to θ_d in $\mathsf{CG}_Q(r')$. Notice that by minimality of B, each process $p \in B$ has a finite critical time $t_p = t_p(\theta_s, \theta_d)$ wrt. $(\mathsf{CG}_Q(r), B)$. We now prove by induction on t that for all $\langle i,t\rangle \in T_B$, if i has not crashed by time t in r', then $r_i(t) = r'_i(t)$.

Base: $t = 0$. By assumption, $r'(0) = r(0)$. Thus, $r'_i(0) = r_i(0)$ for every process i and in particular for those satisfying $\langle i,0\rangle \in T_B$.

Step. Let $t > 0$ and assume that the claim holds for all nodes $\langle l,t'\rangle$ with $t' < t$. Fix a node $\langle i,t\rangle \in T_B$. Clearly, $\langle i,t-1\rangle \in T_B$, and so by the inductive hypothesis $r_i(t-1) = r'_i(t-1)$. To establish our claim regarding $\langle i,t\rangle$, it suffices to show that i receives exactly the same messages at time t in both runs. Since messages are delivered in one time step in our model, the only messages that i can receive at time t are ones sent at time $t-1$. Hence, we reason by cases, showing that every process $z \neq i$ sends i the same messages at time $t-1$ in both runs.

- Suppose that $\langle z,t-1\rangle \in T_B$.
  - If $z \in \mathbb{F}^r \backslash B$ then it is active at time $t-1$ in r' iff it is active at this time in r. We have by the inductive assumption that it has the same local state in r. Since Q is deterministic, z sends i a message at time $t-1$ in r' iff it does so in r. Moreover, if it sends a message, it sends the same message in both cases.
  - We show that if $z \in B$, then the channel $\mathsf{ch}_{z,i}$ is not blocked at time $t-1$ in r'. Assume by way of contradiction that $\mathsf{ch}_{z,i}$ is blocked at time $t-1$. This means that z has failed in r' by time $t-1$. Since $z \in B$ and z has failed in r' by time $t-1$, we have that $t_z \leq t-1$. By definition of z's critical time t_z, there is a $B_{\not\emptyset}$ path π from θ_s to $\langle z,t_z\rangle$ in $\mathsf{CG}_Q(r)$. There also is a path in $\mathsf{CG}_Q(r)$ from $\langle z,t_z\rangle$ to $\langle z,t-1\rangle$ consisting of locality edges. Together, these two paths form a $B_{\not\emptyset}$ path from θ_s to $\langle z,t-1\rangle$ in $\mathsf{CG}_Q(r)$, contradicting the assumption that $\langle z,t-1\rangle \in T_B$.

  - Assume that $z \in B$ and $\mathsf{ch}_{z,i}$ is not blocked at time $t-1$ in r'. Then, as in the previous case, the inductive assumption and the fact that Q is deterministic imply that exactly the same communication occurs between $\langle z, t-1\rangle$ and $\langle i, t\rangle$ in both runs.
  - Finally, assume that $z \notin B \cup \mathbb{F}^r$. Then z is active at time $t-1$ in both r' and r. We have by the inductive assumption that it has the same local state in r. Since Q is deterministic, z sends i a message at time $t-1$ in r' iff it does so in r. Moreover, if it sends a message, it sends the same message in both cases.
- Now suppose that $\langle z, t-1\rangle \notin T_B$, i.e., there is a $B_{\not\gamma}$ path from θ_s to $\langle z, t-1\rangle$ in $CG_Q(r)$. Since $\langle i, t\rangle \in T_B$ we have that z does not send a message to i at time $t-1$ in r. There is no edge from $\langle z, t-1\rangle$ to $\langle i, t\rangle$ in $\mathsf{CG}_Q(r)$. Since $FP(r') \leq FP(r)$, it holds by Lemma 27 that $\mathsf{CG}_Q(r') \subseteq_u \mathsf{CG}_Q(r)$ and hence, there is no edge from $\langle z, t-1\rangle$ to $\langle i, t\rangle$ in $\mathsf{CG}_Q(r')$ either. Meaning that i does not receive a message from z neither at (r, t) nor at (r', t).

Since $r_i(t-1) = r'_i(t-1)$ process i performs the same actions at time $t-1$ in both runs. Since, in addition, i receives exactly the same messages at time t in r' as it does in $\hat{r}$ as we have shown, it follows that $r_i(t) = r'_i(t)$.

The inductive argument above showed that, for all processes i and all times $t \leq m$, if i is active at time t and $\langle i, t\rangle \in T_B$, then $r_i(t) = r'_i(t)$. Since, $\theta_d \in T_B$ by assumption, it follows that, in particular, $r_d(m) = r'_d(m)$. Since there is no enhanced message chain from θ_s to θ_d in r', we obtain that $\neg K_d(\theta_s \rightsquigarrow \theta_d)$ at time m in r by the Definition 2 of the knowledge operator. This contradicts the assumption that $K_d(\theta_s \rightsquigarrow \theta_d)$ holds at time m in r, completing the proof. ◀

Proof of Lemma 12

Proof. Denote by $\mathbb{F}^r$ the set of faulty processes in r, and assume that the conditions of the Silent Choir Theorem do not hold. I.e., neither a silent choir nor an actual message chain from θ to θ' exist in r. Let $\theta' = \langle q, m\rangle$. Let S be the set of processes such that for each $p \in S$ there is an actual message chain from θ to $\langle p, m-1\rangle$. Since there is no silent choir, the following holds: $|S \cup \mathbb{F}^r| \leq f$.

Let $B \triangleq S \cup \mathbb{F}^r$ and let π be a path from θ to θ' in $CG_Q(r)$. Since there is no actual message chain from θ to θ' in $CG_Q(r)$ we get that there is an edge from a process of B in π that is in E_{n}, i.e., corresponds to a null message sent by a process in B. This holds for every path from θ to θ'. Hence there exists a set of processes B such that $|B \cup \mathbb{F}^r| \leq f$ and such that there is no $B_{\not\gamma}$ path from θ to θ' in $CG_Q(r)$, i.e., the conditions of Theorem 11 do not hold, completing the proof. ◀

Proof of Theorem 16

Proof. The assumptions guarantee that there will always be at least one path from θ_s to θ_d in CG along which no "silent" process fails. Let Q' be an NbM protocol such that $\mathsf{nG}(Q') = \mathsf{CG}$. We show by induction that in all runs in which $v_s = 0$ each process along the path will detect that the run is not nice. In particular, j will be able to distinguish the run from the nice one by time m. It follows that $K_d(v_s = 1)$ holds in $\hat{r}$ at time m.

Let r' be a run in which $v_s = 0$ and denote by B the set of processes that fail in this run. Clearly, $|B| \leq f$. Let π be a $B_{\not\gamma}$ path in $\mathsf{nG}(Q')$, which is guaranteed to exist by the assumption. Let r' be a run in which $v_s = 0$. We now prove by induction on time that for each node $\langle p, t\rangle$ in π it holds that $K_p(\neg\psi_{nice})$ holds at (r', t). .

Base: $t = 0$. In this case, $p = s$. Since v_s appears in s's local state and its value differs to its value in the nice run, $K_s(\neg\psi_{nice})$ holds at time 0.

Step: $t > 0$. We consider the nodes $\langle q, t-1\rangle$ and $\langle p, t\rangle$ in π. By the induction hypothesis $K_q(\neg\psi_{nice})$ holds at $t-1$ in r'. We now reason by cases according to the class of the edge $(\langle q, t-1\rangle, \langle p, t\rangle)$ in $\mathsf{nG}(Q')$.

- Case 1: $(\langle q, t-1\rangle, \langle p, t\rangle) \in E_{\mathsf{l}}$ then $p = q$ and since the fact $\neg\psi_{nice}$ is a stable property we have by the induction hypothesis that $K_p(\neg\psi_{nice})$ holds at (r', t).
- Case 2: $(\langle q, t-1\rangle, \langle p, t\rangle) \in E_{\mathsf{a}}(\hat{r})$:
  - Case 2a: in the run r' process q does not send p a message, then p detects that the run is not $\hat{r}$ (in which, by assumption, it would receive a message from q).
  - Case 2b: q does send a message to p in r' then, by the induction assumption and the fact that q sends 0 if $K_q(\neg\psi_{nice})$ it follows that p receives a different message in r' and in $\hat{r}$, and so $K_p(\neg\psi_{nice})$ holds at time t.
- Case 3: $(\langle q, t-1\rangle, \langle p, t\rangle) \in E_{\mathsf{n}}(\hat{r})$ we have by the choice of π that q does not fail in r' and by the induction assumption $K_q(\neg\psi_{nice})$ holds at time $t-1$. Recall that, by assumption, in Q' process q can send a null message only in case $\neg(K_q\neg\psi_{nice})$. Since, by the inductive assumption on time $t-1$ this is not the case, q must send p a '0'-message. Since such messages are never sent in $\hat{r}$, we again conclude that $K_p(\neg\psi_{nice})$ holds at time t in r', as desired.

We have shown that for all runs r' in which $v_s \neq 1$ it is the case that $r'_d(m) \neq \hat{r}_d(m)$. Consequently, $v_s = 1$ for all runs r such that $r_d(m) = \hat{r}_d(m)$ and so, by Definition 2, we obtain that $K_d(v_s = 1)$ holds at $(\hat{r}, m)$, as claimed. ◀

Definition of Robust-based Message protocols

▶ **Definition 29** (Robust-based Message protocols). *We say that Q is a Robust-based Message (*RbM*) protocol if*

- *All actual messages sent in Q are single-bit messages, and whenever a process p sends an actual message, it sends a '0' if $K_p(v_s \neq 1 \vee s$ is faulty$)$ and sends a '1' otherwise,*
- *for all processes p, each null message sent by p over any channel is a null message in case $\varphi = \neg K_p[(v_s \neq 1) \vee (s$ is faulty$)]$ and*
- *for every run r and process p it holds that if p sends q an actual message at $(\hat{r}, t)$, then if the channel $\mathsf{ch}_{p,q}$ is not blocked at (r, t), process p also sends q an actual message at (r, t).*

Ordered Response

▶ **Definition 30** (Conservative O-R protocols). *Let Q be a deterministic protocol that solves* $\mathsf{OR} = \langle v_s = 1, a_1, \ldots, a_k\rangle$. *We say that Q is conservative for* OR *if for every run r of Q and all $x \leq k$ the following is true: Process i_x performs a_x at t_x only if $\neg K_{i_x}(\neg\psi_{nice})$ holds at (r, t_x).*

In a conservative protocol, if a process i_x knows at $\theta_x = \langle i_x, t_x\rangle$ that a failure has occurred, then it is not allowed to perform its action. Concretely, suppose that process i_x is prevented from action at $\theta_x = \langle i_x, t_x\rangle$ because it observes there that a process b has failed at $\rho_b = \langle b, m_b\rangle$. Since by Theorem 20, only $(f-1)$-resilient message blocks are required between two consecutive processes in the OR instance, the failure of $f-1$ other processes might disconnect θ_x from a node θ_h, for an index $h > x$ in the instance of O-R being solved.

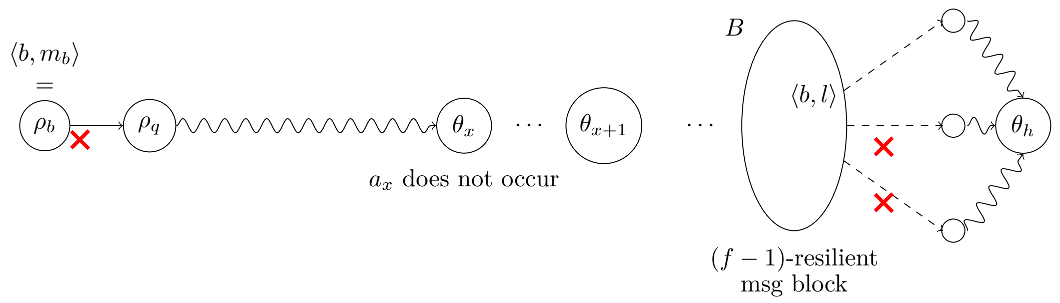


Figure 6 The problematic scenario that Theorem 31 solves. Squiggly arrows represent any kind of message chain or sets of message chains. Red crosses represent process failures. As in previous figures, dashed arrows represent null messages and full arrows represent real messages. If the depicted scenario occurs, then b's failure at ρ_b can cause i_x not to perform a_x. The protocol must therefore provide an $(f-1)$-resilient message block to θ_h from one of θ_x or $\langle b, m_b+1\rangle$.

Moreover, this might also disconnect ρ_b from θ_h. We would then obtain that i_h does not distinguish the current run from the nice run, resulting in a_h being performed. This is clearly a violation of O-R. We illustrate this scenario in Figure 6.

We can show that in order to prevent such a scenario there must be a $B_{\not{v}}$ path from θ_x or from b after its potential failing node ρ_b, to θ_h. This way, if b fails at ρ_b and prevents i_x from acting, then i_h will distinguish the current run from the nice run at θ_h. Acting conservatively, i_h will also refrain from acting, and thus avoid causing a violation of the O-R specification. Formally:[6]

▶ **Theorem 31.** *Let Q be a* conservative *protocol solving* $\mathsf{OR} = \langle v_s = 1, a_1, \ldots, a_k\rangle$.
For all nodes $\rho_b = \langle b, m_b\rangle$, indices $x < h \leq k$ and sets $B \subseteq \mathbb{P}$, if

1. *there is a path π from ρ_b to θ_x in $\mathsf{nG}(Q)$ that starts with an edge $(\rho_b, \rho_q) \in E_{\mathsf{a}}$ and contains no edges corresponding to null message by b, and in addition*
2. *$b \in B$, $|B| \leq f$ and there is no $B_{\not{v}}$ path from θ_x to θ_h in* CG,

then there is a $B_{\not{v}}$ path from $\langle b, m_b+1\rangle$ to θ_h in $\mathsf{nG}(Q)$.

In a precise sense, combining the conditions in this theorem with those of Theorem 20 we obtain a set of conditions that is not only necessary for conservative protocols Q (as already proved), but also sufficient. Indeed, as we now show, there exist protocols solving Ordered Response that satisfy precisely these conditions.

▶ **Theorem 32** (Sufficient conditions for O-R)**.** *The conditions stated in Theorem 20 and Theorem 31 are sufficient for solving an instance* $\mathsf{OR} = \langle v_s = 1, a_1, a_2, \ldots, a_k\rangle$ *of the Ordered Response problem. (For details, see [17].)*

Taken together, Theorems 20, 31, and 32 provide a characterization of the communication patterns that can solve Ordered Response using null messages. This characterization is tight for communication patterns of *conservative* protocols that solve O-R.

[6] The proofs of Theorems 31 and 32 appear in [17].

Gorilla: Safe Permissionless Byzantine Consensus

Youer Pu
Cornell University, Ithaca, NY, USA

Ali Farahbakhsh
Cornell University, Ithaca, NY, USA

Lorenzo Alvisi
Cornell University, Ithaca, NY, USA

Ittay Eyal
Technion, Haifa, Israel

Abstract

Nakamoto's consensus protocol works in a permissionless model and tolerates Byzantine failures, but only offers probabilistic agreement. Recently, the Sandglass protocol has shown such weaker guarantees are not a necessary consequence of a permissionless model; yet, Sandglass only tolerates benign failures, and operates in an unconventional partially synchronous model. We present Gorilla Sandglass, the first Byzantine tolerant consensus protocol to guarantee, in the same synchronous model adopted by Nakamoto, deterministic agreement and termination with probability 1 in a permissionless setting. We prove the correctness of Gorilla by mapping executions that would violate agreement or termination in Gorilla to executions in Sandglass, where we know such violations are impossible. Establishing termination proves particularly interesting, as the mapping requires reasoning about infinite executions and their probabilities.

2012 ACM Subject Classification Computer systems organization → Dependable and fault-tolerant systems and networks

Keywords and phrases Consensus, Permissionless, Blockchains, Byzantine fault tolerance, Deterministic Safety

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.31

Related Version *Full Version*: `https://arxiv.org/abs/2308.04080` [29]

Funding *Youer Pu*: Supported in part by an IC3 grant.
Lorenzo Alvisi: Supported in part by NSF Award 2106954.
Ittay Eyal: Supported in part by the Israel Science Foundation (grant No. 1641/18), an Avalanche Foundation grant, and an IC3 grant.

Acknowledgements We thank Alexandra Silva and Dexter Kozen for useful discussions about proving probabilistic termination.

1 Introduction

Nakamoto's Bitcoin [23] demonstrated that a form of consensus can be reached even if participation is permissionless. Nakamoto achieved this by introducing the cryptographic primitive Proof of Work (PoW) [10, 13] into the common synchronous Byzantine model [9]. With PoW, a process can work for a short while and probabilistically succeed in solving a puzzle. But Bitcoin only achieves a probabilistic notion of consensus: both safety and liveness fail with negligible probability.

Lewis-Pye and Roughgarden showed that deterministic and permissionless consensus cannot be achieved in a synchronous network in the presence of Byzantine failures [18]. Nonetheless, previous work (§2) has achieved deterministic safety and termination with probability 1 under different models. Sandglass [27] assumes a benign model with a non-standard hybrid synchrony model. Momose et al. [22] guarantee termination only if the set of processes stabilizes. Malkhi et al. [20], while leveraging either authenticated channels or digital signatures, propose a solution whose correctness depends on Byzantine nodes comprising fewer than a third of the nodes in the system.

The question is whether it is possible to achieve deterministic safety and termination with probability 1 without limiting either Byzantine behavior or how nodes join and leave, and without relying on authentication.

We answer this question in the affirmative for a synchronous model (§3) with Byzantine failures. We present *Gorilla Sandglass* (or simply *Gorilla*) (§4), a consensus protocol that guarantees deterministic safety and termination with probability 1 in this standard model, which we dub *GM* (for *Gorilla Model*). Gorilla relies on a form of PoW: *Verifiable Delay Functions* (*VDFs*) [6]. We consider an ideal VDF [21] that proves a process waited for a certain amount of time and cannot be amortized. The key difference between a VDF and Nakamoto's PoW is that multiple processes can calculate multiple VDFs concurrently, but cannot, by coordinating, reduce the time to calculate a single VDF. The crux of the protocol is simple. The protocol proceeds in steps. In each step, all (correct) nodes collect VDF solutions from their peers and build new VDFs based on those. Intuitively, correct processes, which are the majority, accrue solutions faster than Byzantine nodes, and progress through the asynchronous rounds of the protocol faster. Eventually, the round inhabited by correct nodes is so far ahead of that occupied by Byzantine nodes that, no longer subject to Byzantine influence, correct nodes can safely decide.

Gorilla Sandglass adopts the general approach of Sandglass [27], in the sense that puzzle results are accrued, with each puzzle built on its predecessors. In Sandglass participants are benign and they send, in each step, a message built on previously received messages. In Gorilla, however, the Byzantine adversary is not limited to acting on step boundaries or communicating at particular times. Surprisingly, Gorilla's correctness can be reduced to the correctness of a variation of Sandglass. We perform this reduction in two steps (§5).

We first show that, for every execution of Gorilla in GM, there is a matching execution where the Byzantine processes adhere to step boundaries, in a model we call *GM+*. In the mapped execution, Byzantine processes only start calculating their VDF at the beginning of a step and only send messages at the end of a step. GM+ is a purely theoretical device, as it allows operations that cannot be implemented by actual cryptographic primitives. In particular, it allows Byzantine processes to start calculating a VDF in a step s building on any VDF computed by other Byzantine nodes that will be completed *by the end* of s, rather than by the start s, as allowed by GM (and actually feasible in reality). Nonetheless, GM+ serves as a crucial stepping stone towards proving Gorilla's correctness.

Next, we show that, given an execution in GM+ that violates correctness, there exists a corresponding execution of Sandglass in a model we call SM+. The SM+ model is similar to that of Sandglass: in both, processes are benign and propagation time is bounded for messages among correct processes and unbounded for messages to and from so-called *defective nodes*. But unlike Sandglass, in SM+ a message from a defective node can reference another message generated by another defective node during the same step (similar to how GM+ allows Byzantine nodes to calculate a VDF that builds on VDFs calculated by other Byzantine nodes in the same step).

Together, this pair of reduction steps establishes that if an execution of Gorilla in GM violates correctness with positive probability, then so does an execution of Sandglass in SM+. To conclude Gorilla's proof of correctness, all that is left to show is that Sandglass retains deterministic safety and termination with probability 1 in the SM+ model: fortunately, the correctness proof of Sandglass [27] works almost without change (§A of [29]) in SM+. Thus, a violation of correctness in Gorilla results in a contradiction, and therefore, Gorilla is correct.

Gorilla demonstrates that it is *possible* to achieve deterministic safety and liveness with probability 1 in a permissionless Byzantine model. Yet, possible does not mean *practical*: Gorilla is not, since, like the Sandglass protocol that inspires it, it requires an exponential number of rounds to terminate. By answering the fundamental question of possibility, Gorilla ups the ante: is there a practical solution to deterministically safe permissionless consensus?

2 Related Work

Lewis-Pye et al. [18] have proven that deterministic consensus is impossible in the permissionless setting. Therefore, for at least one of safety and liveness probabilistic guarantees are inevitable. Gorilla concedes little: it manages to keep safety deterministic, and guarantees liveness with probability 1.

Several protocols [3, 4, 7, 11, 12, 16, 24] have embraced Bitcoin's permissionless participation and probabilistic safety. All rely for correctness on probabilistic mechanisms, which leave open the possibility that Byzantine nodes may overturn safety or liveness guarantees with positive probability. Gorilla avoids this peril by basing correctness on the process of accruing a deterministic number of messages.

Few proposals achieve deterministic safety in a permissionless setting [20, 22, 27]. Momose et al. [22] introduce the concept of *eventually stable participation*, akin to partial synchrony; it requires that, after an unknown global stabilization time, for each T-wide time interval $[t, t+T]$, at least half of the nodes ever awake during the interval are correct and do not leave. Gorilla guarantees progress without assuming stability in participation.

Pu et al. [27] propose Sandglass, which achieves deterministic safety but only in a benign setting. Gorilla extends Sandglass to tolerate Byzantine failures.

Malkhi et al. [20] let nodes join and leave at any time, as in Gorilla. Unlike Gorilla, however, they must rely on authenticated channels to tolerate fluctuations in the number of adversaries. Further, Byzantine nodes must be fewer than a third of active nodes, while in Gorilla they must be fewer than one half.

Several works have modeled permissionless participation [2, 17, 26].

Pass et al. [26] introduce the *sleepy participation model*, in which honest nodes are either awake or asleep. Awake nodes participate in the protocol, while asleep nodes neither participate nor relay messages. Byzantine nodes are always awake, but the scheduler can adaptively turn an honest node Byzantine, as long as Byzantine nodes remain a minority of awake nodes. Gorilla similarly assumes that correct and Byzantine nodes can join and leave at any time, as long as a majority of active nodes are correct. Unlike the sleepy model, however, Gorilla requires no public key infrastructure, and, unlike sleepy consensus, guarantees deterministic safety.

Unlike Gorilla, Lewis-Pye et al. [17] do not offer a consensus protocol, but rather focus on introducing *resource pools*, an abstraction that aims to capture resources used to establish identity in permissionless systems, *e.g.*, computational power through PoW and fiscal power through Proof of Stake (PoS).

Aspnes et al. [2] explore consensus in an asynchronous benign model where an unbounded number of nodes can join and leave, but where at least one node is required to live forever, or until termination. Gorilla instead assumes a synchronous model, tolerates Byzantine failures, and allows any node to join and leave, as long as a majority of active nodes is correct.

Verifiable Delay Functions (VDFs) [6] have been leveraged as a resource against Byzantine adversaries in various works [8, 15, 19, 30], specifically to defend PoS systems from attacks where participants can go back in time and mine blocks. Gorilla leverages VDFs to rate-limit the ability of Byzantine nodes to create valid messages.

3 Model

The system is comprised of an infinite set of nodes $\{p_1, p_2, \dots\}$. Time progresses in discrete ticks $0, 1, 2, 3, \dots$ In each tick, a subset of the nodes is *active*; the rest are *inactive*. The upper bound on active nodes in any tick, necessary to the safety of Nakamoto's permissionless consensus [25], is $\mathcal{N}$, and there is at least one active node in every tick. Starting from tick 0, every K ticks are grouped into a step: each step i consists of ticks $iK, iK+1, \dots, iK+K-1$.

A Verifiable Delay Function (VDF) is a function whose calculation requires completing a given number of sequential steps. Thus, evaluating a VDF requires the evaluator to spend a certain amount of time in the process. Specifically, we require the evaluation of a single VDF to take K ticks. We refer to the intermediate random values that this evaluation produces at the end of each of the K ticks as the *units of the VDF evaluation* (or, more succinctly, the *units of the VDF*). We denote the i-th unit of evaluating the VDF of some input γ by vdf_γ^i; we denote the final result (*i.e.,* vdf_γ^K) by vdf_γ, or, when there is no ambiguity, by *vdf*.

We model the calculation of VDFs with the help of an *oracle* Ω. Nodes use Ω both to iteratively obtain the units of a VDF and to verify whether a given value is the *vdf* of a given input. In particular, Ω provides the following API:

Get(γ, vdf_γ^i): returns $vdf_\gamma^{(i+1)}$. By convention, invoking Get$(\gamma, \perp)$ returns vdf_γ^1. The oracle remembers how it responded to a Get query – so that, even though the units of a VDF are random values, identical queries produce identical responses. Ω accepts at most one call to Get() in any tick from each node.

Verify(vdf, γ): returns True iff $vdf = vdf_\gamma^K$. Ω accepts any number of calls to Verify() in any tick from any node.

If Get$(\gamma, \perp)$ is called at tick t and step s, we say the VDF calculation for γ *starts* at tick t and step s. Similarly, the VDF calculation for γ *finishes* at tick t and step s if Get$(\gamma, vdf_\gamma^{K-1})$ is called at tick t and step s.

In each tick, an active node receives a non-negative number of messages, updates its variables – potentially including calls to the oracle – and then communicates with others using a synchronous broadcast network. The network allows each active node to *broadcast* and *receive* unauthenticated messages. Node p_i invokes $Broadcast_i(m)$ to broadcast a message m, and receives broadcast messages from other nodes (and itself) by invoking $Receive_i$. The network neither generates nor duplicates messages and ensures that if a node receives a message m in tick t, then m is broadcast in tick $(t-1)$. The network propagation time is negligible compared to a tick, *i.e.*, to the time necessary to calculate a unit of a VDF. By executing the command $Receive_i$, a newly joining node p_i receives all messages broadcast by correct nodes prior to its activation. Nodes whose network connections with other nodes are asynchronous can be modeled as Byzantine, as Byzantine nodes can deliberately or unintentionally delay messages sent from or to them. Therefore, Gorilla also tolerates asynchrony, as long as the nodes that communicate asynchronously are a minority.

Correct nodes do not deviate from their specification and constitute a majority of active nodes at each tick. Correct nodes always join at the beginning of a step and leave when a step ends. Hence, a correct node is active from the first to the last tick of a step. The remaining nodes are *Byzantine* and can suffer from arbitrary failures. Byzantine nodes can join and leave at any tick.

All nodes are initialized with a value $v_i \in \{0,1\}$ upon joining the system. An active node p_i decides by calling $Decide_i(v)$ for some value v. A protocol solves the consensus problem if it guarantees the following properties [9]:

▶ **Definition 1** (Agreement). *If a correct node decides a value v, then no correct node decides a value other than v.*

▶ **Definition 2** (Validity). *If all nodes that ever join the system have initial value v and there are no Byzantine nodes, then no correct node decides $v' \neq v$.*

▶ **Definition 3** (Termination). *Every correct node that remains active eventually decides.*

4 Gorilla

Gorilla borrows its general structure from Sandglass (see Algorithm 1) [27]. Executions proceed in asynchronous rounds (even though, unlike Sandglass, Gorilla assumes a standard synchronous model of communication between all nodes). Upon receiving a threshold of valid messages for the current round, nodes progress to the next round; if all the messages received by a correct node propose the same value v for sufficiently many consecutive rounds, the node decides v. The number of active nodes is bounded by $\mathcal{N}$ but otherwise unknown. Within this bound, it can fluctuate arbitrarily, but both safety and liveness depend on the correctness of a majority of nodes.

The key aspects of the protocol can be summarized as follows:

Ticks, steps and VDF Each valid message must contain a *vdf*. A correct node takes a full step, *i.e.*, K consecutive ticks, to individually calculate a *vdf*, and at the end of the step sends a valid message that contains the *vdf*. Byzantine nodes may instead share among themselves the work required to finish the K units of a VDF calculation; even so, it still takes K distinct ticks for Byzantine nodes to compute a *vdf*. Requiring valid messages to carry a *vdf* limits Byzantine nodes to sending messages at the same rate as correct nodes; this ensures that, on average across all steps, the correct majority sends at least one more valid message than the minority of nodes that are Byzantine.

Choosing a threshold A node proceeds to round r if it receives at least $\mathcal{T} = \lceil \frac{\mathcal{N}^2}{2} \rceil$ messages for round $r-1$. Even though setting such a threshold does not prevent Byzantine nodes from advancing from round to round, it nonetheless gives the correct nodes an edge in the pace of such progress, since they constitute a majority.

Exchanging messages In each step of the protocol, a node in any round r – based on the messages it has received so far – searches for the largest round $r_{max} \geq r$ for which it has accrued $\mathcal{T}$ messages. It then broadcasts a message for the next round. The message includes the node's current proposed value v, the *vdf*, and four other attributes discussed below: the message's *coffer*, a nonce, as well as v's *priority* and *unanimity counter*.

Keeping history Nodes can join the system at any time. To help a joining node catch up, every message broadcast by a node p in round r includes a *message coffer* that contains: (i) messages from round $r-1$ received by p to advance to round r; (ii) recursively, messages included in those messages' coffers; and (iii) messages received by p for round r.

Nonce By making it possible to distinguish between messages that are generated from the same coffer, nonces allow correct nodes to broadcast multiple valid messages during a round while, at the same time, preventing Byzantine nodes from reusing the same *vdf* to send multiple valid messages based on a given message coffer.

Priority and unanimity counter If a node p only receives the value v from a majority for a sufficient number of consecutive rounds, it decides v. To guarantee the safety of this decision, p assigns a *priority* to the value v that it proposes. This priority is incremented once v is unanimously proposed for a long stretch of consecutive rounds. To record the length of this stretch, each node computes it upon entering a round r, and includes it as the *unanimity counter* in the messages it sends for round r. If a node collects more than one value in a round r, it chooses the one with the highest priority, and proposes it for round $r+1$. In case of a tie, it uses *vdf* as a source of randomness to choose one of the values randomly. Since *vdf* is a random number calculated based on the message coffer and a nonce (lines 13-15), a Byzantine node is unable to deliberately pick an input to VDF to deterministically get the desired value.

Message internal consistency and validity A message m is *internally consistent* if the attributes carried by m can be generated by following Gorilla correctly based on the message coffer carried in m. We denote the *vdf* in m by vdf_m.
A message m is *valid* (and thus isValid(m) returns true), if (i) vdf_m can be verified by the message coffer and the nonce of m; (ii) m is internally consistent; and (iii) for any message m' in m's coffer, m' is also valid. Otherwise, m is invalid.

In addition to demonstrating variable initialization, Algorithm 1 presents the algorithm each node p_i runs at each step. Each node p_i starts every step by adding all valid messages, in addition to the messages in their coffers, to the set Rec_i (lines 4-6).

Iterating over Rec_i, node p_i computes the largest round r_{max} for which it has received at least $\mathcal{T}$ messages, and updates its current round to $r_{max}+1$ (line 8) if the condition in line 7 holds. Once in a new round, p_i does the following: (i) resets its message coffer M and adds to it the messages it has received from the previous round – alongside the messages in *their* coffers (lines 9-11); (ii) picks a nonce and calculates a *vdf* based on its coffer and the nonce (lines 13-15); (iii) chooses its proposal value (lines 16 -20); it chooses the proposal with the highest priority among the previous round messages in its coffer; in case of a tie, it chooses a random number utilizing the randomness in *vdf*; (iv) determines the priority and the unanimity counter for the messages it will broadcast in the current round (lines 21-25); and finally (v) the node decides v if v's priority is high enough (lines 26-27). If p_i does not enter a new round, it starts to create a message nonetheless: it adds to the message's coffer all messages received for the current round (line 29), and calculates a *vdf* with the new message coffer and a different nonce as the input (lines 30-32), so that the message is unique. Regardless of whether it enters a round or not, p_i ends every step by broadcasting the message it has created (line 33).

Comparing Sandglass and Gorilla

Gorilla retains the structure of Sandglass, adding the requirement that valid messages must include a *vdf* and a nonce. The differences between the protocols are highlighted in orange in Algorithm 1: (i) *vdf* is calculated for each message sent (lines 13-15,30-32), (ii) received messages are checked to see if they are valid (line 5); (iii) *vdf* is used as the source of randomness (line 20) where the protocol requires choosing a value randomly.

These additions are critical to handling Byzantine faults. Both Gorilla and Sandglass rely on correct (respectively, good) nodes sending the majority of unique messages during an execution. In Sandglass, where defective nodes are benign, this property simply follows from requiring correct nodes to be a majority in each step; not so in Gorilla, where faulty nodes can be Byzantine. Requiring valid message in Gorilla to carry a *vdf* preserves correctness by effectively rate-limiting Byzantine nodes' ability to create valid messages.

Algorithm 1 Gorilla: Code for node p_i. The orange text highlights where Gorilla departs from Sandglass.

```
1:  procedure INIT($input_i$)
2:      $v_i \leftarrow input_i$; $priority_i \leftarrow 0$; $uCounter_i \leftarrow 0$; $r_i = 1$; $M_i = \emptyset$; $Rec_i = \emptyset$;
3:  procedure STEP
4:      for all $m = (\cdot,\cdot,\cdot,\cdot,\cdot,M)$ received by $p_i$ do
5:          if isValid(m) then
6:              $Rec_i \leftarrow Rec_i \cup \{m\} \cup M$
7:      if $\max_{|Rec_i(r)| \geq \mathcal{T}}(r) \geq r_i$ then
8:          $r_i = \max_{|Rec_i(r)| \geq \mathcal{T}}(r) + 1$
9:          $M_i = \emptyset$
10:         for all $m = (\cdot, r_i - 1, \cdot,\cdot,\cdot,M) \in Rec_i(r_i - 1)$ do
11:             $M_i \leftarrow M_i \cup \{m\} \cup M$
12:         $M_i \leftarrow M_i \cup Rec_i(r_i)$
13:         $vdf \leftarrow \bot$; $nonce \leftarrow$ a new arbitrary value
14:         for $j : 1..k$ do
15:             $vdf \leftarrow Get((M_i, nonce), vdf)$
16:         Let $C$ be the multi-set of messages in $M_i(r_i - 1)$ with the largest priority.
17:         if all messages in $C$ have the same value $v$ then
18:             $v_i \leftarrow v$
19:         else
20:             $v_i \leftarrow vdf \mod 2$
21:         if all messages in $M_i(r_i - 1)$ have the same value $v_i$ then
22:             $uCounter_i \leftarrow 1 + \min\{uCounter | (\cdot, r_i - 1, v_i, \cdot, uCounter, \cdot) \in M_i(r_i - 1)\}$
23:         else
24:             $uCounter_i \leftarrow 0$
25:         $priority_i \leftarrow \max(0, \lfloor \frac{uCounter_i}{\mathcal{T}} \rfloor - 5)$
26:         if $priority_i \geq 6\mathcal{T} + 4$ then
27:             Decide$_i(v_i)$
28:     else
29:         $M_i \leftarrow M_i \cup Rec_i(r_i)$
30:         $vdf \leftarrow \bot$; $nonce \leftarrow$ a new arbitrary value
31:         for $j : 1..k$ do
32:             $vdf = Get((M_i, nonce), vdf)$
33:     broadcast $(r_i, v_i, priority_i, uCounter_i, M_i, nonce, vdf)$
```

Given their differences in both failure model and timing assumptions, it is perhaps surprising that so little needs to change when moving from Sandglass to Gorilla. After all, Sandglass assumes a model where failures are benign and a hybrid synchronous model of communication [28]; Gorilla instead assumes a Byzantine failure model, and a synchronous network model (§3). Note, however, that although Sandglass assumes benign failures, its hybrid communication model implicitly accounts for Byzantine nodes strategically choosing the timing for receiving and sending messages to correct nodes: Gorilla can then simply inherit from Sandglass the mechanisms for tolerating such behaviors.

5 Correctness

Despite the similarlity between the Gorilla and Sandglass protocols, proving Gorilla's correctness directly is challenging. Unlike Sandglass, Byzantine nodes can act between step boundaries, interleave VDF computations instead of producing one VDF (and hence one message) at the time, etc. To overcome this complexity, our approach is to leverage as much as possible Sandglass's proof of correctness.

Our battle plan was to first map executions of Gorilla to executions of Sandglass. Then we intended to proceed by contradiction: assume that a correctness guarantee is violated in Gorilla, and map this violation to Sandglass; since correctness violations are not possible in Sandglass [27], we could then conclude that neither they can be in Gorilla.

The best laid plans often go awry, and, as we discuss below, ours was no exception – but we were able to nonetheless retain the conceptual simplicity of our initial approach.

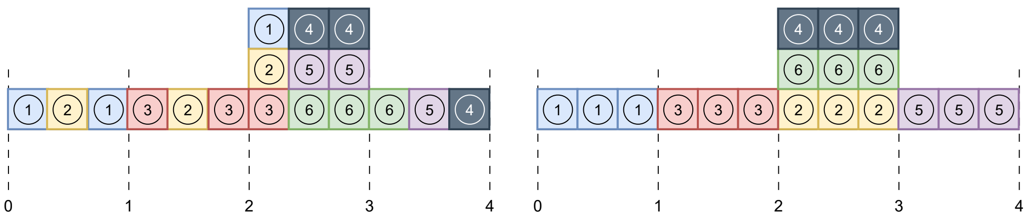
5.1 The Main Story, and How it Fails

The mapping from Gorilla to Sandglass must satisfy certain *well-formedness* and *equivalence* conditions. The former specify how to map a Gorilla execution into one that satisfies the Sandglass model (SM) and follows the Sandglass protocol; the latter allow us to map violations from Gorilla to Sandglass, *i.e.*, they preserve certain properties of the behavior of *correct* nodes in Gorilla and reinterpret them as the behavior of *good* nodes in Sandglass.

Well-formedness requires mapping correct nodes to good nodes, and Byzantine nodes to defective nodes, while respecting model constraints (*e.g.*, at each step defective nodes should be fewer than good nodes). The first half of this mapping is easy: except for calculating a VDF, correct nodes in GM are not doing anything different than good nodes in SM. Thus, mapping a step in GM to a step in SM yields a straightforward connection between correct and good nodes. The second half, however, is trickier. Defective nodes in SM can suffer from benign faults like omission and crashing, but these fall short of fully capturing Byzantine behavior in GM. In particular, Byzantine nodes, even when sending valid messages, can violate the timing constraints that Gorilla places on a node's actions, *e.g.*, by splitting the calculation of a single VDF into multiple steps. Thus, before a Gorilla execution can be mapped to a Sandglass execution, Byzantine nodes' actions must be brought to conform to step boundaries and not spill across steps. After tidying things up this way, it must become possible to map the faulty actions of the Byzantine nodes to a combination of crashes, omissions, and network delays, *i.e.*, to the faults and anomalies that SM allows.

Equivalence in turn requires that, when mapping executions from Gorilla to Sandglass, a correct node and its corresponding good node send and receive in every step messages that allow them to update their proposed value, round number, priority, and unanimity counter in the same way. Since messages play the same role in both protocols, this is sufficient for good nodes in Sandglass to decide identically to the corresponding correct nodes in Gorilla.

Our plan to realize this logical mapping involved splitting it into two concrete, intermediate mappings: a first mapping from an initial Gorilla execution to an intermediate Gorilla execution in which Byzantine actions conform to step boundaries; and a second mapping from that intermediate execution to a Sandglass execution. We require all of our well-formedness and equivalence conditions to hold throughout these mappings: (*i*) model constraints must be always respected, (*ii*) correct nodes in the intermediate execution send and receive the equivalent (indeed, the same!) messages as their counterparts in the initial execution, at the same steps, and (*iii*) good nodes in the final execution send and receive equivalent messages as their correct counterparts in the intermediate execution, at the same steps.



(a) The counterexample. **(b)** The solution enabled by peeking.

Figure 1 An execution that cannot be reorganized in GM (a), and how peeking solves the problem in GM+ (b).

Unfortunately, *well-formedness and equivalence cannot be satisfied by the first mapping*. To see why, consider Figure 1a. Here, each square represents a VDF unit calculated by a Byzantine node for a specific input, denoted by a unique color. Numbered circles represent the corresponding messages, *e.g.*, the VDF units containing ① are associated with message ①. Each VDF calculation takes three ticks, and a step comprises three ticks. The numbered dashed lines indicate the steps, *i.e.*, the three ticks between lines i and $i+1$ belong to step i. Assume that, to maintain a majority of correct nodes in the system, the maximum allowable number of Byzantine nodes in the four steps shown in the figure are, respectively, 1, 1, 3, and 1. Moreover, assume that messages ④, ⑤, and ⑥ all include in their coffers messages ①, ②, and ③. Finally, assume that messages ④, ⑤, and ⑥ are sent to correct nodes at the start of Step 4. Since the actions of Byzantine nodes in Figure 1a do not conform to step boundaries, the first mapping should be able to organize them in a way that ensures that (i) correct nodes receive messages ④, ⑤, and ⑥ at the beginning of Step 4, and (ii) each of these messages in turn includes messages ①, ②, and ③. Thus, the calculation of the VDFs for messages ①, ②, and ③ must be completed before those for ④, ⑤, and ⑥ can start. Now, since steps 0 and 1 include only one Byzantine node, they can only accommodate one VDF, *i.e.*, only one VDF can be calculated in each of steps 0 and 1. Without loss of generality, let those VDFs be ① and ③, respectively. VDF ② must still complete before messages ④, ⑤, and ⑥: thus, it has to be placed in Step 2. Note that, although Step 2 could accommodate two more Byzantine VDFs at Step 2, they cannot be placed there, since the completion of VDF ② must precede the start of the calculation of VDFs ④, ⑤, and ⑥: the earliest step where they can start is Step 3. However, it is impossible to accommodate all three there, since in Step 3 there is a single Byzantine node.

Our first attempt at mapping executions from Gorilla to Sandglass has thus failed. Fortunately, though, it is possible to retain the strategy that underlies it and overcome the above counterexample without weakening our well-formedness and equivalence conditions. Instead, we proceed to weaken the model in which we operate, by giving Byzantine nodes extra power.

5.2 A New Beginning

The first step in our two-step process for mapping a Gorilla execution η_G into a Sanglass execution η_S is to reorganize the actions taken by Byzantine nodes in η_G: we want to map η_G to an execution where Byzantine nodes join the system and receive valid messages at the beginning of a step (by the first tick) and broadcast valid messages and leave the system at the step's end (at its K-th tick). Since, as explained in Section 5.1, satisfying all of these requirements is not possible, we extend GM to a new model.

We need some way to calculate a VDF on an input that includes the final result of VDF calculations that are still in progress. To achieve this, we extend the oracle's API to allow Byzantine nodes to *peek* at those future outcomes. By issuing the oracle a *peek* query, Byzantine nodes active in any step s can learn the result of a VDF computed by Byzantine nodes finishing at step s before its calculation has ended.

We thus introduce GM+, a model that extends GM by having a new oracle, Ω^+, that supports one additional method:

Peek(γ): immediately returns vdf_γ.

In any tick, a Byzantine node in GM+ can call Peek() multiple times, with different inputs. However, Byzantine nodes can only call Peek subject to two conditions:

- A Byzantine node can peek in step s at vdf_γ only if Byzantine nodes commit to finish the VDF calculation for input γ within s; and
- a Byzantine node does not peek at vdf_γ, where $\gamma = (M, nonce)$, if M in turn contains some VDF result v obtained by peeking, and the calculation of v has yet to finish in this tick.

Note that these restrictions only limit the *additional* powers that GM+ grants the adversary: in GM+, Byzantine nodes remain strictly stronger than in GM.

With this new model, taking a detour, we first map an execution of Gorilla in GM to an execution of Gorilla in GM+, in which Byzantine behavior is reorganized with the addition of peeking. Hence follows the first lemma of our scaffolding: the existence of the first mapping.

▶ **Definition 4.** *Consider an execution η_G in GM and an execution η_G^+ in GM+. We say η_G and η_G^+ are* equivalent *iff the following conditions are satisfied:*

Reorg-1 *For every correct node p in η_G, there exists a correct node p^+ in η_G^+, such that p and p^+ (i) join and leave the system at the same ticks in the same steps and (ii) receive and send the same messages at the same ticks in the same steps.*

Reorg–2 *Each Byzantine node in η_G^+ (i) joins at the first tick of a step and leaves after the last tick of that step; (ii) receives messages at the first tick of a step and sends messages at the last tick of that step; and (iii) sends and receives only valid messages.*

Reorg-3 *If in η_G a Byzantine node sends a valid message m at a tick in step s, then in η_G^+ a Byzantine node sends m at a tick in some step $s' \leq s$.*

▶ **Lemma 5.** *There exists a mapping* Reorg *that maps an execution η_G in GM to an execution η_G^+ in GM+, denoted $\eta_G^+ = \textsc{Reorg}(\eta_G)$, such that η_G is equivalent to η_G^+.*

While peeking solves the challenge with reorganizing Byzantine behavior, it complicates our second mapping. The ability to peek granted to Byzantine nodes in GM+ has no equivalent in Sandglass – it simply cannot be reduced to the effects of network delays or to the behavior of defective nodes. Therefore, we weaken SM so that defective nodes can benefit from a capability equivalent to peeking.

We do so by introducing SM+, an extension of SM that is identical to SM, except for the following change: defective nodes at step s can receive any message m sent by a defective node no later than s – as opposed to $(s-1)$ in SM – as long as m does not contain in its coffer a message that is sent at s. Note that allowing defective nodes to receive in a given step a message m sent by defective nodes within that very step maps to allowing Byzantine nodes to peek at a message whose *vdf* will be finished by Byzantine nodes within the same step; and the constraint that m shouldn't contain in its coffer other messages sent in the same step, maps to the constraint that Byzantine nodes cannot peek at messages whose coffer also contains a peek result from the same step.

One might rightfully ask: was not the plan to leverage the correctness of Sandglass in SM? Indeed, but fortunately, *Sandglass still guarantees deterministic agreement and termination with probability 1 under the SM+ model* (§A.3 of [29]). Thus, it is suitable to map a Gorilla execution in GM+ to a Sandglass execution in SM+, and orient our proof by contradiction with respect to the correctness of Sandglass in SM+.

Formally, we specify our second mapping as follows.

▶ **Definition 6.** *Given a message m in the Gorilla protocol, the mapping* MAPM *produces a message in the Sandglass protocol as follows*

1. *Omit the vdf and the nonce from m.*
2. *Let p_i be the node that sends m. Include p_i as a field in m.*
3. *If m is the j-th message sent by p_i, add a field uid $= j$ to m.*
4. *Repeat the steps above for all of the messages in m's coffer.*

Denote the result by $\hat{m} = \text{MAPM}(m)$. We say m and $\hat{m}$ are equivalent*. Furthermore, with a slight abuse of notation, we apply* MAPM *to a set of messages as well,* i.e., *if $\mathcal{M}$ is a set of messages, and we map each message $m \in \mathcal{M}$, we obtain the message set $\text{MAPM}(\mathcal{M})$.*

▶ **Definition 7.** *Consider an execution η_G^+ in GM+ and an execution η_S^+ in SM+. We say η_G^+ and η_S^+ are* equivalent *iff the following conditions are satisfied:*

1. *The nodes in η_G^+ are in a one-to-one correspondence with the nodes in η_S^+. For every node p in η_G^+, we denote the corresponding node in η_S^+ with $\hat{p}$.*
2. *Nodes p and $\hat{p}$ join and leave at the same steps in η_G^+ and η_S^+, respectively. Furthermore, their initial values are the same.*
3. *If p is a Byzantine node, then $\hat{p}$ is defective in SM+; otherwise, $\hat{p}$ is a good node in SM+.*
4. *$\hat{p}$ sends $\hat{m}$ at step s in η_S^+, iff p generates a message m in η_G^+ at step s. Note that in η_G^+, correct nodes send their messages to all as soon as they are generated, while Byzantine nodes may only send their messages to a subset of nodes once their messages are generated.*
5. *$\hat{p}$ receives $\hat{m}$ at step s in η_S^+, iff p receives m at step s in η_G^+.*

▶ **Lemma 8.** *Consider any execution η_G in GM, and an execution η_G^+ in GM+ equivalent to η_G. There exists a mapping* INTERPRET *that maps η_G^+ to an execution η_S^+ in SM+, denoted as $\eta_S^+ = \text{INTERPRET}(\eta_G^+)$, such that η_S^+ is equivalent to η_G^+.*

Finally, for our proof by contradiction to work, we have to show that Sandglass is correct in SM+. The proof is deferred to §A.3 of [29].

▶ **Theorem 9.** *Sandglass satisfies agreement and validity deterministically and termination with probability 1 in SM+.*

5.3 Safety

We prove that Gorilla satisfies Validity and Agreement. The proofs follow the same pattern: assume a violation exists in some execution η_G of Gorilla running in GM; map that execution to $\eta_G^+ = \text{REORG}(\eta_G)$ in GM+; then, map η_G^+ again to $\eta_S^+ = \text{REORG}(\eta_G^+)$ in SM+; and, finally, rely on the fact that these mappings ensure that correct nodes in η_G and good nodes in η_S^+ reach the same decisions in the same steps to drive a contradiction.

This approach is made rigorous in following lemmas, proved in §B.2 of [29].

▶ **Lemma 10.** *Consider an arbitrary Gorilla execution η_G, and $\eta_G^+ = \text{REORG}(\eta_G)$. If a correct node p decides a value v at step s in η_G, then p's corresponding node p^+ decides v at step s in η_G^+.*

▶ **Lemma 11.** *Consider any execution η_G in GM. If an execution $\eta_G^+ = \text{Reorg}(\eta_G)$ in GM+ and an execution η_S^+ in SM+ are equivalent, then the following statements hold:*
1. *If a correct node p decides a value v at step s in η_G^+, then $\hat{p}$ decides v at step s in η_S^+.*
2. *Consider the first message $m = (r, v, priority, uCounter, M, nonce, vdf)$ that p generates for round r. Let the step when m is generated be s. If uCounter is 0, then $\hat{p}$ randomly chooses value v as the proposal value at step s in η_S^+.*

We can now state and prove the safety guarantees.

▶ **Theorem 12.** *Gorilla satisfies agreement in GM.*

Proof. By contradiction, assume that there exists a Gorilla execution η_G in GM that violates agreement. This means that there exist two correct nodes p_1 and p_2, two steps s_1 and s_2, and two values $v_1 \neq v_2$ such that p_1 decides v_1 at s_1 and p_2 decides v_2 at s_2. Consider $\eta_G^+ = \text{Reorg}(\eta_G)$. According to Lemma 10, p_1^+ decides v_1 at s_1 and p_2^+ decides v_2 at s_2, in η_G^+. Now, consider $\eta_S^+ = \text{Interpret}(\eta_G^+)$. According to Lemma 11, $\hat{p}_1^+$ decides v_1 at s_1 and $\hat{p}_2^+$ decides v_2 at s_2, in η_S^+. However, this contradicts the fact Sandglass satisfies agreement in SM+ (Theorem 9). Therefore, Gorilla satisfies agreement in GM. ◀

▶ **Theorem 13.** *Gorilla satisfies validity in GM.*

Proof. By contradiction, assume that there exists a Gorilla execution η_G, such that (i) all nodes that ever join the system have initial value v; (ii) there are no Byzantine nodes; and (iii) a correct node p decides $v' \neq v$.

Since GM+ is an extension of GM, η_G conforms to GM+. According to Definition 4, $\eta_G^+ = \eta_G$ in GM+ is trivially equivalent to η_G. Consider $\eta_S^+ = \text{Interpret}(\eta_G^+)$.

By the construction of the Interpret mapping (in Lemma 8), good nodes in η_S^+ have the same initial values as their corresponding correct nodes in η_G. Furthermore, since there are no Byzantine nodes in η_G^+, there are no defective nodes in η_S^+ by Definition 7. Therefore, by Validity of Sandglass in SM+ (Theorem 9), no good node decides $v' \neq v$. However, by Lemma 10 and Lemma 11, p decides $v' \neq v$, which leads to a contradiction. Therefore, Gorilla satisfies validity in GM. ◀

5.4 Liveness

Similar to the safety proof, the liveness proof proceeds by contradiction: it starts with a liveness violation in Gorilla, and maps it to a liveness violation in Sandglass.

Formalizing the notion of violating termination with probability 1 requires specifying the probability distribution used to characterize the probability of termination. To do so, we first have to fix all sources of non-determinism [1, 5, 14]. For our purposes, non-determinism in GM and GM+ stems from correct nodes, Byzantine nodes and their behavior; in SM+, it stems from good nodes, defective nodes and the scheduler.

For correct, good, and defective nodes, non-determinism arises from the joining/leaving schedule and the initial value of each joining node. For Byzantine nodes in GM and GM+, fixing non-determinism means fixing their action *strategy* according to the current history of an execution. Similarly, fixing the scheduler's non-determinism means specifying the timing of message deliveries and the occurrence of benign failures, based on the current history. We, therefore, define non-determinism formally in terms of an environment and a strategy.

To this end, we introduce the notion of a *message history*, and define what it means for a set of messages exchanged in a given step to be *compatible* with the message history that precedes them.

▶ **Definition 14.** *For any given execution in GM and GM+ (resp., SM+), and any step s, the* message history up to s*, $\mathcal{MH}_s$, is the set of $\langle m, p, s' \rangle$ triples such that p is a correct node (resp., good node) and p receives m at $s' \leq s$.*

▶ **Definition 15.** *We say a set $\mathcal{MP}_{s+1}$ of $\langle m, p, s+1 \rangle$ triples is* compatible *with a message history up to s, $\mathcal{MH}_s$, if there exists an execution such that for any $\langle m, p, s+1 \rangle \in \mathcal{MP}_{s+1}$, the correct node (resp., good node) p receives m at step $(s+1)$.*

▶ **Definition 16.** *An environment $\mathcal{E}$ in GM and GM+ (resp., SM+) is a fixed joining/leaving schedule and fixed initial value schedule for correct nodes (resp., good and defective nodes).*

▶ **Definition 17.** *Given an environment $\mathcal{E}$, a strategy $\Theta_{\mathcal{E}}$ for the Byzantine nodes (resp., scheduler) in GM and GM+ (resp., SM+) is a function that takes the message history $\mathcal{MH}_s$ up to a given step s as the input, and outputs a set $\mathcal{MP}_{s+1}$ that is compatible with $\mathcal{MH}_s$.*

Before proceeding, there is one additional point to address. The most general way of eliminating non-determinism is to introduce randomness through a fixed probability distribution over the available options. However, the following lemma, proved in §B.3 of [29], establishes that Byzantine nodes do not benefit from employing such a randomized strategy.

▶ **Lemma 18.** *For any environment $\mathcal{E}$, if there exists a* randomized *Byzantine strategy for Gorilla that achieves a positive non-termination probability, then there exists a* deterministic *Byzantine strategy for Gorilla that achieves a positive non-termination probability.*

Since the output *vdf* of a call to the VDF oracle is a random number, the $(vdf \mod 2)$ operation in line 20 of Gorilla is equivalent to tossing an unbiased coin. Given a strategy $\Theta_{\mathcal{E}}$,[1] the nodes might observe different coin tosses as the execution proceeds; thus, the strategy specifies the action of the Byzantine nodes for all possible coin toss outcomes. The scheduler's strategy in SM+ is similarly specified for all coin toss outcomes. Therefore, once a strategy is determined, it admits a *set* of different executions based on the coin toss outcomes; we denote it by H_Θ. Specifically, a strategy determines an action for each outcome of any coin toss.

Given a strategy Θ, we can define a probability distribution P_{H_Θ} over H_Θ. For each execution $\eta \in H_\Theta$, there exists a unique string of zeros and ones, representing the coin tosses observed during η. Denote this bijective correspondence by $\textsc{Coins} : H_\Theta \to \{0,1\}^* \cup \{0,1\}^\infty$, and the probability distribution on the coin toss strings in $\textsc{Coins}(H_\Theta)$ by $\tilde{P}_{H_\Theta}$. For every event $E \subset H_\Theta$, if $\textsc{Coins}(E)$ is measurable in $\textsc{Coins}(H_\Theta)$, then $\tilde{P}_{H_\Theta}(\textsc{Coins}(E))$ is well-defined; thus, $P_{H_\Theta}(E)$ is also well-defined and $P_{H_\Theta}(E) = \tilde{P}_{H_\Theta}(\textsc{Coins}(E))$. We denote P_{H_Θ} as the probability distribution induced over H_Θ by its coin tosses.

Equipped with these definitions, we can formally define termination with probability 1.

▶ **Definition 19.** *The Gorilla protocol terminates with probability 1 iff for every environment $\mathcal{E}$ and every Byzantine strategy Θ based on $\mathcal{E}$, the probability of the termination event T in H_Θ,* i.e., *$P_{H_\Theta}(T)$, is equal to 1.*

This definition gives us the recipe for proving by contradiction that Gorilla terminates with probability 1. We first assume there exists a Byzantine strategy Θ that achieves a non-zero non-termination probability, and map this strategy through the Reorg and Interpret mappings to a scheduler strategy Λ that achieves a non-zero non-termination probability in SM+. However, Λ cannot exist, as the Sandglass protocol terminates with probability 1 in SM+ (Theorem 9).

[1] When it is clear from the context, we will omit the environment from the subscript of the strategy.

▶ **Lemma 20.** *If there exists an environment $\mathcal{E}$ and a Byzantine strategy $\Theta_{\mathcal{E}}$ in GM that achieves a positive non-termination probability, then there exists an environment $\mathcal{E}'$ and a Byzantine strategy $\Psi_{\mathcal{E}'}$ in GM+ that also achieves a positive non-termination probability.*

Proof. Assume there exist an environment $\mathcal{E}$ and a Byzantine strategy $\Theta_{\mathcal{E}}$ in GM that achieves a positive non-termination probability. Consider the Reorg mapping. Since, according to Lemma 5, the joining/leaving and initial value schedules for correct nodes remain untouched by the Reorg mapping, we just set $\mathcal{E}' = \mathcal{E}$. In the rest of the proof, we omit the environments for brevity.

We now show that the strategy Ψ exists, and is in fact the same as Θ. For brevity, let R_Θ denote Reorg(H_Θ), and consider any execution η in H_Θ. By Lemma 5, correct nodes in η receive the same messages, at the same steps, as the correct nodes in Reorg(η) and, moreover, the coin results in η are exactly the same as the ones in Reorg(η). Thus, the message history of correct nodes up to any step s in η is the same as the message history of correct nodes up to the same step in Reorg(η). In addition, because Reorg(η) is a GM+ execution, compatibility is trivially satisfied. Thus, we conclude that Byzantine nodes in R_Θ follow the same strategy as in Θ, conforming to the same coin toss process. Let us denote this strategy with Ψ.

Note that according to Lemma 10, whenever a correct node decides at some step s in η, its corresponding correct node in Reorg(η) decides the same value at the same step. Therefore, the set of non-terminating executions in H_Θ are mapped to the set of non-terminating executions in R_Θ in a bijective manner. Let us denote these sets as NT_H and NT_R, respectively. Since the same coin toss process induces probability distributions P_{H_Θ} and P_{R_Θ} on H_Θ and R_Θ, respectively, we conclude that $P_{H_\Theta}(NT_H) = P_{R_\Theta}(NT_R)$. Therefore, since $P_{H_\Theta}(NT_H) > 0$ by assumption, this concludes our proof, as we have shown the existence of a strategy Ψ in GM+ that achieves a positive non-termination probability. ◀

A similar lemma applies to the second mapping. We prove it in §B.3 of [29].

▶ **Lemma 21.** *If there exists an environment $\mathcal{E}$ and a strategy Ψ for Byzantine nodes in GM+ that achieves a positive non-termination probability, then there exists an environment $\mathcal{E}'$ and a scheduler strategey $\Lambda_{\mathcal{E}'}$ in SM+ that also achieves a positive non-termination probability.*

Based on these lemmas, we are finally ready to prove Gorilla's liveness guarantee.

▶ **Theorem 22.** *The Gorilla protocol terminates with probability 1.*

Proof. By contradiction, assume that there exist a GM environment and a Byzantine strategy Θ in Gorilla that achieve a positive non-termination probability. By Lemma 20, there exist a GM+ environment and a strategy Ψ for the Byzantine nodes in GM+ that achieve a positive non-termination probability. Similarly, by Lemma 21, there exists an SM+ environment and a scheduler strategy Λ in SM+ that achieve a positive non-termination probability. But this is a contradiction, since Sandglass terminates with probability 1 in SM+ (Theorem 9). Thus, Byzantine strategy Θ cannot force a positive non-termination probability; Gorilla terminates with probability 1. ◀

6 Conclusion

Gorilla Sandglass is the first Byzantine-tolerant consensus protocol to guarantee, in the same synchronous model adopted by Nakamoto, deterministic agreement and termination with probability 1 in a permissionless setting. To this end, Gorilla leverages VDFs to extend the approach of Sandglass, the first protocol to provide similar safety guarantees in the presence of benign failures. Neither Gorilla nor Sandglass are practical protocols, however: they exchange a very large number of messages and the number of rounds they require to decide is large even under favorable circumstances, and can, in general, be exponential. Is there a *practical* permissionless protocol that can achieve deterministic safety and tolerate fewer than a half Byzantine nodes?

References

1 James Aspnes. Randomized protocols for asynchronous consensus. *Distributed Computing*, 16(2-3):165–175, 2003.

2 James Aspnes, Gauri Shah, and Jatin Shah. Wait-free consensus with infinite arrivals. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 524–533, 2002.

3 Christian Badertscher, Peter Gaži, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 913–930, 2018.

4 Vivek Bagaria, Sreeram Kannan, David Tse, Giulia Fanti, and Pramod Viswanath. Prism: Deconstructing the blockchain to approach physical limits. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 585–602, 2019.

5 Gilles Barthe, Joost-Pieter Katoen, and Alexandra Silva. *Foundations of Probabilistic Programming*. Cambridge University Press, 2020.

6 Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In *Annual international cryptology conference*, pages 757–788. Springer, 2018.

7 Phil Daian, Rafael Pass, and Elaine Shi. Snow white: Robustly reconfigurable consensus and applications to provably secure proof of stake. In *Financial Cryptography and Data Security: 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18–22, 2019, Revised Selected Papers 23*, pages 23–41. Springer, 2019.

8 Soubhik Deb, Sreeram Kannan, and David Tse. Posat: proof-of-work availability and unpredictability, without the work. In *Financial Cryptography and Data Security: 25th International Conference, FC 2021, Virtual Event, March 1–5, 2021, Revised Selected Papers, Part II 25*, pages 104–128. Springer, 2021.

9 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

10 Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *Proceedings CRYPTO '92: 12th International Cryptology Conference*, pages 139–147. Springer, 1992.

11 Matthias Fitzi, Peter Ga, Aggelos Kiayias, and Alexander Russell. Parallel chains: Improving throughput and latency of blockchain protocols via parallel composition. *Cryptology ePrint Archive*, 2018.

12 Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th symposium on operating systems principles*, pages 51–68, 2017.

13 Markus Jakobsson and Ari Juels. Proofs of work and bread pudding protocols. In *Secure Information Networks*, pages 258–272. Springer, 1999.

14 Benjamin Lucien Kaminski. *Advanced weakest precondition calculi for probabilistic programs*. PhD thesis, RWTH Aachen University, 2019.

15 Rami Khalil and Naranker Dulay. Short paper: Posh proof of staked hardware consensus. *Cryptology ePrint Archive*, 2020.

16 Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Advances in Cryptology–CRYPTO 2017: 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20–24, 2017, Proceedings, Part I*, pages 357–388. Springer, 2017.

17 Andrew Lewis-Pye and Tim Roughgarden. Resource pools and the cap theorem. *arXiv preprint arXiv:2006.10698*, 2020.

18 Andrew Lewis-Pye and Tim Roughgarden. Byzantine generals in the permissionless setting, 2021. `doi:10.48550/ARXIV.2101.07095`.

19 Jieyi Long. Nakamoto consensus with verifiable delay puzzle. *arXiv preprint arXiv:1908.06394*, 2019.

20 Dahlia Malkhi, Atsuki Momose, and Ling Ren. Byzantine consensus under fully fluctuating participation. *Cryptology ePrint Archive*, 2022.

21 Michael Mirkin, Lulu Zhou, Ittay Eyal, and Fan Zhang. Sprints: Intermittent blockchain pow mining. *Cryptology ePrint Archive*, 2023.

22 Atsuki Momose and Ling Ren. Constant latency in sleepy consensus. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2295–2308, 2022.

23 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, page 21260, 2008.

24 Joachim Neu, Ertem Nusret Tas, and David Tse. Ebb-and-flow protocols: A resolution of the availability-finality dilemma. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 446–465. IEEE, 2021.

25 Rafael Pass, Lior Seeman, and Abhi Shelat. Analysis of the blockchain protocol in asynchronous networks. In *Advances in Cryptology – EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Proceedings*, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), pages 643–673. Springer Verlag, 2017.

26 Rafael Pass and Elaine Shi. The sleepy model of consensus. In *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part II 23*, pages 380–409. Springer, 2017.

27 Youer Pu, Lorenzo Alvisi, and Ittay Eyal. Safe Permissionless Consensus. In Christian Scheideler, editor, *36th International Symposium on Distributed Computing (DISC 2022)*, volume 246 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 33:1–33:15, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.DISC.2022.33`.

28 Youer Pu, Lorenzo Alvisi, and Ittay Eyal. Safe permissionless consensus. Cryptology ePrint Archive, Paper 2022/796, 2022. URL: `https://eprint.iacr.org/2022/796`.

29 Youer Pu, Ali Farahbakhsh, Lorenzo Alvisi, and Ittay Eyal. Gorilla: Safe permissionless byzantine consensus, 2023. `arXiv:2308.04080`.

30 Ronghua Xu and Yu Chen. Fairledger: a fair proof-of-sequential-work based lightweight distributed ledger for iot networks. In *2022 IEEE International Conference on Blockchain (Blockchain)*, pages 348–355. IEEE, 2022.

Distributed Sketching Lower Bounds for k-Edge Connected Spanning Subgraphs, BFS Trees, and LCL Problems

Peter Robinson
School of Computer & Cyber Sciences, Augusta University, GA, USA

Abstract

We investigate graph problems in the distributed sketching model, where each node sends a single message to a referee who computes the output. We define a class of graphs and give a framework for proving lower bounds for certain embeddable problems, which leads to several new results: First, we present a tight lower bound of $\Omega(n)$ bits for the message size of computing a breadth-first search (BFS) tree. For locally-checkable labeling (LCL) problems, we show that verifying whether a given vertex labeling forms a weak 2-coloring requires messages of $\Omega(n^{1/3}\log^{2/3} n)$ bits, and the same lower bound holds for verifying whether a subset of nodes forms a maximal independent set. We also prove that computing a k-edge connected spanning subgraph (k-ECSS) requires messages of size at least $\Omega\big(k\log^2(n/k)\big)$, which is tight up to a logarithmic factor. To show these results, we define a simultaneous multiparty (SMP) model of communication complexity, where the players obtain certain subgraphs as their input, and develop a generic embedding argument that allows us to prove lower bounds for the graph sketching model by using reductions from the SMP model. We point out that these results also extend to single-round algorithms in the broadcast congested clique.

We also (nearly) settle the space complexity of the k-ECSS problem in the streaming model by extending the work of Kapralov, Nelson, Pachoki, Wang, and Woodruff (FOCS 2017): We prove a communication complexity lower bound for a direct sum variant of the $\mathsf{UR}_k^{\subset}$ problem and show that this implies $\Omega(k\,n\log^2(n/k))$ bits of memory for computing a k-ECSS. This is known to be optimal up to a logarithmic factor.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Distributed graph algorithm, graph sketching, streaming

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.32

1 Introduction

Understanding the amount of communication that is required for solving fundamental graph problems has been at the forefront of research in distributed computing. In this work, we consider the *distributed graph sketching model* (SKETCH), introduced in [5]. In SKETCH there are n nodes and each node starts out knowing its neighborhood of the input graph. After observing its initial state and the shared randomness, each node sends a single message to the referee, who does not get any input and is responsible for computing the output by inspecting the received messages. As elaborated in [16, 3, 28], the distributed sketching model is equivalent to the *single-round broadcast congested clique* (BCC_1), where each node sends a single message of β bits, where β denotes the *link bandwidth*, and these messages are received by all nodes simultaneously at the end of the round. Consequently, the results of our work apply to both models.

We assume that the nodes are assigned unique IDs from the set $[n]$. In addition, we equip the nodes with some amount of initial knowledge of the input graph, namely, each node knows not only its own ID but also the IDs of all of its neighbors. This is known as the KT_1 *assumption*, which has turned out to be a key ingredient for achieving communication-efficiency in distributed algorithms (see [19, 14, 13, 4]). We point out that KT_1 knowledge presents a significant obstacle when proving lower bounds, due to fact that each edge is part of the input of *both* of its endpoints. Consequently, we cannot independently modify the input of a player (i.e. node) without affecting other nodes. This is a crucial difference between the models assumed in our work and, for instance, the edge-partition models, where each player obtains a subset of edges as their input [25, 27].

At a first glance, it may seem that the non-interactive computation of one-round algorithms presents a severe handicap to solving *any* interesting problem with a distributed algorithm in this setting, despite the initial KT_1 knowledge. However, the breakthrough results of Ahn, and Guha, and McGregor [2] (see also the work of Kapron, King, and Mountjoy [18]) introduced a linear sketching technique that opened up the possibility of communication-efficient solutions in SKETCH and BCC_1 for several fundamental graph problems, including computing spanning trees and deciding graph connectivity, while requiring messages (also called "sketches") of only polylogarithmic length.

1.1 Our Contributions and Related Work

A Lower Bound Technique for the Distributed Graph Sketching Model. We present an embedding approach for proving lower bounds in the distributed sketching model (SKETCH) and, equivalently, in the single-round broadcast congested clique (BCC_1). This technique generalizes an approach that was pioneered by Nelson and Yu [22], who proved an $\Omega(\log^3 n)$ lower bound in this setting for computing a spanning forest. In a subsequent breakthrough, Yu [28] extended this work by showing that this is a tight lower bound even for the easier problem of graph connectivity.

Our approach differs from previous works by defining a *simultaneous multiparty (*SMP*) model* as an intermediate step, where some of the players may get an entire subgraph as their input rather than just the neighborhood of a single node. A technical challenge is that the inputs of different players overlap with each other, which rules out using simple product distributions for the lower bound. We specify a class of fairly generic lower bound graphs and introduce the notion of *embeddable problem*, which captures a broad range of intuitive properties, making it applicable to seemingly unrelated problems such as computing a k-edge connected spanning subgraph and verifying a weak 2-coloring. For embeddable problems that have unique outputs for a given input (e.g., decision problems), we obtain a reconstruction procedure that succeeds with sufficiently high probability in recovering the output, while omitting the transcript of some players. For general embeddable problems, which may not have uniquely determined outputs, we use Pinsker's inequality to argue that omitting the transcript of some players does not significantly skew the probability distribution of certain important cut sets. We point out that the reconstruction mechanism for unique output problems has a significantly improved error probability compared to using Pinsker's inequality as in [22, 28], which may be useful for other applications.

In more detail, we choose the class of lower bound graphs such that there is a large set of nodes V with the property that all nodes in V have neighborhoods that are "similar", i.e., are identically distributed. We show that, for solving an embeddable problem, the referee needs to obtain a sufficient amount of information about the neighborhood of one specific important node $v_\sigma \in V$. However, since the index σ is not given to the algorithm, the neighbors of the nodes in V do not know which one of their own neighbors is v_σ and consequently end up sending messages of large size to ensure a small probability of error. For instance, when computing a BFS tree, the node v_σ is chosen to be the only node in V for which all of its incident edges are part of any BFS tree. Due to the lack of knowledge of σ thus effectively requires the referee to learn about the neighborhoods of all nodes in V.[1]

[1] The author would like to thank the anonymous DISC 2023 reviewer for suggesting this intuition.

Computing a BFS Tree. Similarly to computing a spanning tree, computing a BFS tree has a small output size of $\Theta(n \log n)$ bits, and hence one might expect that the sketching technique of [2], which allows recovering an incident edge for each vertex, would lead to a solution using only sketches of length $O(\text{poly} \log n)$. We show that this intuition is misleading by presenting a tight bound of $\Omega(n)$ on the message size for computing a BFS tree in BCC_1 and SKETCH. This reveals a near-linear gap to the problem of computing a spanning tree, which requires only messages of $O\big(\log^3 n\big)$ bits. For the proof of this result, we only need to use the generic lower bound graph construction and do not require the full machinery of the embedding argument. With the right lower bound construction in place, the result readily follows from a reduction to the index problem in two-party communication.

▶ **Theorem 1.** *Any public coin constant-error randomized algorithm that computes a BFS tree rooted at a designated node of an n-node graph, requires a worst case message length of $\Omega(n)$ bits in the distributed sketching model* (SKETCH) *and the one-round broadcast congested clique* (BCC_1)*.*

Verifying Symmetry Breaking Problems. We apply the embedding technique to locally-checkable labeling (LCL) problems [21], which have been studied extensively in the distributed computing literature and, roughly speaking, are graph problems that can be verified locally in the sense that each node only needs to check the consistency of the assigned labels in its $O(1)$-neighborhood. Here, we focus on verifying a weak 2-coloring, which is a vertex coloring of the graph with two colors, with the only restriction being that each non-isolated vertex has at least one neighbor with a different color. Since a weak 2-coloring can be computed from the output of other symmetry breaking problems, it comes as no surprise that more difficult LCL problems such as verifying a maximal independent set adhere to the same lower bound as weak 2-coloring. While the work of Assadi, Kol, and Oshman [3] shows a lower bound of $\Omega\big(n^{1/2-\epsilon}\big)$ bits on the message size for *computing* an MIS in the distributed sketching model, it is unclear whether their result has any implications for the verification problem, due to the fundamentally different nature of computation and verification of symmetry breaking problems. We instantiate the embedding technique to prove the following result:

▶ **Theorem 2.** *Any $\frac{1}{25}$-error randomized algorithm that verifies if a labeling of a subset of vertices forms a weak 2-coloring of an n-node input graph, requires a worst case message length of $\Omega\Big(n^{1/3}\log^{2/3} n\Big)$ bits in* SKETCH *and* BCC_1*. The same bound holds for deciding whether a subset of nodes forms a maximal independent set.*

Computing a k-Edge Connected Spanning Subgraph. By applying the embedding technique, we obtain the first lower bounds for computing a k-edge connected spanning subgraph. Prior to our work, the only known lower bound for this problem was the one for spanning tree construction (i.e., $\Omega(\log^3 n)$ bits, see [22]), which does not scale with k. In particular, for $k = O(\log n)$, the lower bound of [22] for computing a spanning forest immediately implies an $\Omega(\log^3 n)$ lower bound, since the referee can recover a spanning tree from a k-ECSS.

▶ **Theorem 3.** *Any public coin randomized algorithm that computes a k-edge connected spanning subgraph of an n-node graph in* SKETCH *or* BCC_1 *with probability at least $1 - o(1)$, has a worst case message length of $\Omega\big(k \log^2 \frac{n}{k}\big)$ bits, for any $k = o\Big(\frac{n^{1/4}}{\log^{1/2} n}\Big)$.*

We point out that Theorem 3 is tight up to a logarithmic factor, since the algorithm for deciding k-edge connectivity of Ahn, Guha, and McGregor [2] also computes a "witness", i.e., a k-edge connected subgraph. It is straightforward to implement their technique in SKETCH using messages of $O\big(k \log^3 n\big)$ bits.

In Section 6, we consider the k-ECSS problem in the dynamic data streaming setting where the graph is represented as a stream of edge arrivals and departures. To prove a lower bound, we introduce a new communication complexity problem called ℓ-fold $\mathsf{UR}_k^{\subset}$, which essentially consists of ℓ instances of the $\mathsf{UR}_k^{\subset}$ problem, defined by Kapralov, Nelson, Pachoki, Wang, and Woodruff [17]. In the $\mathsf{UR}_k^{\subset}$ problem, there are two players, Alice and Bob. Alice gets a set S, whereas Bob gets a proper subset $T \subset S$. After Alice sends a single message to Bob, he must output k elements in $S \setminus T$. It was shown in [17] that the $\mathsf{UR}_k^{\subset}$ problem requires $\Omega\big(k \log^2 \frac{n}{k}\big)$ bits in the one-way two-party model. The ℓ-fold $\mathsf{UR}_k^{\subset}$ problem is a direct-sum variant of the $\mathsf{UR}_k^{\subset}$ problem and, by a simple extension of the lower bound technique of [17], we prove that ℓ-fold $\mathsf{UR}_k^{\subset}$ requires $\Omega(k\,\ell \log^2 \frac{n}{k})$ bits. This in turn gives rise to a lower bound on the required memory:

▶ **Theorem 4.** *Any Monte Carlo data structure for computing a k-edge connected spanning subgraph of an n-node graph requires $\Omega\big(k\,n \log^2 \frac{n}{k}\big)$ space in the one-pass fully dynamic turnstile model.*

1.2 Additional Related Work

Closely related to k-ECSS is the problem of computing a spanning forest of the input graph in the distributed sketching model. As mentioned above, Nelson and Yu [22] prove a lower bound of $\Omega(\log^3 n)$ bits and this is known to be optimal due to the graph sketching approach of [2], which relies on access to shared randomness. Holm, King, Thorup, Zamir, and Zwick [15] show that a spanning tree can be computed with a message length of $O(\sqrt{n}\log n)$ bits, *without* access to shared randomness. Currently, there are no lower bounds known for the distributed sketching model if nodes only have access to private random bits.

While our results only apply to single-round algorithms in the BCC_1 model, several other works have studied multi-round lower bounds in this setting: Drucker, Kuhn, and Oshman [10] show round lower bounds for subgraph detection problems, whereas Chen and Grossman [7] prove a lower bound for the directed planted clique problem. Pai and Pemmaraju [23] give round complexity lower bounds depending on the per-round bandwidth for graph connectivity and finding connected components in BCC_1. The work of [12] considers so called hybrid models resulting from combining the broadcast congested clique with other distributed computing models.

Several other works show lower bounds for one-round algorithms in the related $\mathsf{CONGEST}$ model [24], which differs from the congested clique by assuming that the input graph corresponds to the actual communication network. Fischer, Gonen, Kuhn, and Oshman [11] show that one-round randomized algorithms for triangle detection require nodes to send messages of at least $\Omega(\Delta)$ bits, where Δ is the maximum degree of the graph. Previously, Abboud, Censor-Hillel, Khoury, and Lenzen [1] showed that a slightly stronger bound of $\Omega(\Delta \log n)$ bits for deterministic algorithms based on their novel fooling views framework. We point out that the proof of [1] assumes that all three nodes must detect that they are part of a triangle (if one exists), rather than just at least node as in [11]. A related question is the minimum link bandwidth necessary for obtaining a solution in a certain number of rounds, which is also called bandwidth complexity in [6].

2 A Lower Bound Technique for Embeddable Problems

In this section, we present a generic technique for showing lower bounds for problems that satisfy certain "embeddability" properties. We first define a general class of graphs that we will use for all our lower bounds in Sections 3, 4 and 5, albeit with somewhat different parameters. On these graphs, we define the simultaneous multiparty (SMP) model, and show that embeddable problems have specific properties that enable us to compute a solution while omitting the messages of some player.

2.1 The Lower Bound Graph $\mathcal{G}_\ell$

For a positive integer parameter ℓ, we define a class of graphs $\mathcal{G}_\ell$ that contains all graphs G defined as follows. The vertices of G consist of sets U, V, and W, whereby $|V| = \ell$, and we further partition U into vertex sets $U_1, \dots, U_\ell$. Each v_i is connected to a subset of the vertices in U_i and W. We use E_i to denote the edges in the cut $E(v_i, W)$. Figure 1 on page 19 shows the general structure of the graphs in $\mathcal{G}_\ell$. We will fix the precise cardinalities of U and W as well as the edges $E(U, V)$ and $E(V, W)$ when we introduce the specific input distributions in the subsequent sections. In the problems that we consider, the output will depend on the neighborhood of a particular vertex v_σ, where $\sigma \in [\ell]$ is called the *embedding index.*

We give each vertex a unique integer as its *ID.* In addition to an ID, we also assume that each vertex in W has a *label.* For instance, in the context of verifying a weak 2-coloring, a label of a vertex indicates its color. For k-edge connected spanning subgraphs, on the other hand, we simply omit the labels. The crucial difference between IDs and vertex labels will become apparent when considering the SKETCH model: Every node knows only its own label, but knows the IDs of all nodes in its neighborhood.

2.2 The Simultaneous Multiparty (SMP) Model

In our lower bound constructions, we use the following simultaneous multiparty model as an intermediate step: There are $\ell + 2$ players Alice$_1, \dots,$ Alice$_\ell$, Bob, and Charlie. When revealing the *neighborhood of a vertex u* to a specific player, the player learns the ID and the label of u, as well as the IDs of all of u's neighbors in G. The inputs of the players are defined as follows; see Figure 2 on page 19: For each $v_i \in V$, player Alice$_i$ knows the neighborhood of vertex v_i, whereas Bob knows the neighborhoods of all vertices in W. In other words, Bob knows the entire cut $E(V, W)$, including the labels of W. Charlie gets as input the neighborhoods of all nodes in U, the index σ, and the IDs and labels of the nodes in W.

Alice$_1, \dots,$ Alice$_\ell$ and Bob each send a single message to Charlie who must output the solution. Apart from these messages there is no other communication between the players. However, we assume that they have access to an infinite string R of random bits when considering randomized algorithms.

Random Variables and Notation

Let Π_i denote the message sent by Alice$_i$ and let Π_B denote Bob's message. We use random variable C to denote Charlie's output. By a slight abuse of notation, we assume that U and W also denote the IDs of the vertex sets U and W, respectively. Furthermore, we use $\mathcal{L}_W$ to denote the labels of the nodes in W. We define the abbreviation $\Pi_{(\leqslant j)} := (\Pi_1, \dots, \Pi_j)$ and define $\Pi_{\geqslant j}$ analogously. Observe that Charlie computes C based on his initial knowledge, the received messages Π_B, $\Pi_{(\leqslant \ell)}$, and the shared randomness R, i.e., $C := C(R, U, W, E(U, V), \mathcal{L}_W, \Pi_{(\leqslant \ell)}, \Pi_B, \sigma)$. To shorten the notation, we define

$$Z := (U, W, E(U, V), \mathcal{L}_W),$$

and point out that Charlie's input is Z and σ. We use the indicator random variable $\mathbf{1}_{\mathsf{Succ}}$ for the event that the protocol succeeds.

Throughout this section, we make use of basic notions from information theory. We refer the reader to Appendix A for the formal definition of these quantities and pointers to further references. For random variables X, Y, and Z, we use $\mathbf{H}[X \mid Y]$ to denote the *conditional entropy* of X conditioned on Y, which, intuitively speaking, captures the expected remaining uncertainty of X's value after revealing Y. We use $\mathbf{I}[X : Y \mid Z]$ for the *conditional mutual information* between X and Y conditioned on Z, which is the expected amount of information X reveals about Y (and vice versa) after revealing Z.

2.3 Embeddable Problems

We say that a problem P is *embeddable* if there is an input distribution $\mathcal{D}$ on graphs in $\mathcal{G}_\ell$ that satisfies the following two properties:

(P1) **Independence of the embedding index $\boldsymbol{\sigma}$**: Random variable σ is sampled uniformly from $[\ell]$, and is independent of the edges, labels, and vertex IDs.

(P2) **Independence of cut sets under conditioning**: Random variables $E_1, \dots, E_\ell$ are mutually independent conditioned on Z.

Intuitively speaking, Property (P1) guarantees that the specific value of the index σ does not bias the distribution of the transcripts of the players Alice$_1, \dots,$ Alice$_\ell$, and Bob. Property (P2) ensures that knowing some of the cut sets does not leak information about the remaining cut sets, in particular E_σ. As we will see in Lemma 6 below, Properties (P1) and (P2) are sufficient for obtaining a probability distribution on E_σ that is close to the one that Charlies has access to when computing his output, even though it does not take into account Bob's transcript. For problems that also satisfy the following property (P3), which avoids dependencies between Charlie's output and parts of the graph that are unrelated to E_σ, we give a bound on the probability of directly reconstructing Charlie's output in Lemma 5 below. Note that (P3) is a natural property of decision problems, where E_σ and the labels of its neighbors fully determines the output of the algorithm.

(P3) **Unique Output**: Conditioned on Charlie's input Z, σ, the cut set E_σ, the shared randomness R, and the event that Charlie's output C correctly solves problem P, it holds that C is a deterministic function of E_σ, i.e., $\mathbf{H}[C \mid E_\sigma, R, Z, \sigma, \mathbf{1}_{\mathsf{Succ}}\!=\!1] = 0$.

In the next lemma, we formalize a crucial property of embeddable problems: We can compute a solution with sufficiently large probability just by inspecting Charlie's input and the transcripts of Alice$_1, \dots,$ Alice$_\ell$.

▶ **Lemma 5** (Existence of Reconstruction Protocol). *Consider an embeddable problem* P *with input distribution $\mathcal{D}$ on $\mathcal{G}_\ell$ that satisfies (P1), (P2), and (P3). Suppose that there is a public coin randomized* SMP *protocol that solves* P *with error $\delta \leqslant \min\left\{\frac{1}{2}, \frac{1}{|C|^2}\right\}$. Then there exists a reconstruction protocol $\mathcal{R}(R, Z, \sigma, \Pi_{(\leqslant \ell)})$ that returns Charlie's output C with probability at least $2^{-\left(\frac{|\Pi_B|}{\ell} + 3\sqrt{\delta}\right)}$.*

To gain some intuition for applying Lemma 5, suppose that Charlie just outputs a single bit, i.e., $|C| \leqslant 1$ and that $\delta \leqslant \frac{1}{25}$, which means that $\sqrt{\delta} \leqslant \frac{1}{5}$. Now assume that Bob sends a message of at most $\ell/5 \leqslant \sqrt{\delta}\ell$ bits, which means that, on average, his message can reveal only a $(\frac{1}{5})$-fraction of a bit of information for each of the ℓ cut sets E_i between V and

W. Then, Lemma 5 tells us that we can recover Charlie's output with probability at least $\frac{1}{2^{1/5+3/5}} \approx 0.57$ without Bob's message Π_B. Moreover, if we consider protocols that succeed with high probability, i.e., $\delta \leqslant \frac{1}{\ell}$, and restrict the length of Bob's message to at most $\sqrt{\ell}$ bits, we get a recovery protocol that succeeds with probability at least $\frac{1}{2^{(4/\sqrt{\ell})}} = 1 - o(1)$.

For random variables X and Y, consider the probability distributions $\mu(X)$ and $\mu(X \mid Y{=}y)$. We define $|\mu(X) - \mu(X \mid Y{=}y)|_{TV}$ to be the *total variation distance*, which is the maximum difference in the probability of any event $\mathcal{E}$ on X for these two distributions. We use parts of the techniques developed in the proof of Lemma 5 to show the following result:[2]

▶ **Lemma 6.** *Consider an algorithm for an embeddable problem that satisfies (P1) and (P2). Then, it holds that*

$$\mathbf{E}\left[\left|\mu(E_\sigma \mid Z,\sigma,\Pi_{\leqslant \ell}) - \mu(E_\sigma \mid Z,\sigma,\Pi_{\leqslant \ell},\Pi_B)\right|_{TV}\right] \leqslant 2\sqrt{|\Pi_B|/\ell},$$

where the expectation is taken over Z, σ, $\Pi_{\leqslant \ell}$, and Π_B.

Note that, strictly speaking, Lemma 6 does not give a concrete reconstruction protocol, but instead only an upper bound on the statistical distance between the distribution of E_σ, conditioned on Charlie's input and $\Pi_{\leqslant \ell}$, and the distribution of E_σ where we also condition on Π_B. However, this turns out to be sufficient for obtaining a concrete reconstruction protocol, as we demonstrate in Section 5.

2.4 Proof of Lemma 5

High-Level Overview. Recalling that our goal is to obtain a protocol that recovers the output C without seeing Bob's message Π_B, we start by deriving an upper bound on how much information his message may contain about C. We show that this is roughly equivalent to the amount of information that Π_B conveys about the cut set E_σ (see Lemma 7). In particular, since Bob does not know σ, the amount of information that Π_B contains about E_σ is only a $\frac{|\Pi_B|}{\ell}$-fraction on average (see Lemma 8). In other words, if Bob's message is short compared to the number of cut sets ℓ, then it cannot convey a significant amount of information about E_σ. In Lemma 9, we combine these observations to show that we can guess Charlie's output with a probability of at least $2^{-\frac{|\Pi|}{\ell}}$, where we have omitted some error terms that depend on the success probability of the original protocol.

We now give the detailed argument. Observe that $\Pi_{(\leqslant \ell)}$, Π_B, R, Z, and σ fully determine C, and thus

$$\mathbf{I}\left[C : \Pi_{(\leqslant \ell)},\Pi_B \;\middle|\; R,Z,\sigma\right] = \mathbf{H}[C \mid R,Z,\sigma]. \tag{1}$$

Therefore, by the chain rule, we have that

$$\begin{aligned}\mathbf{I}\left[C : \Pi_{(\leqslant \ell)} \;\middle|\; R,Z,\sigma\right] &= \mathbf{I}\left[C : \Pi_{(\leqslant \ell)},\Pi_B \;\middle|\; R,Z,\sigma\right] - \mathbf{I}\left[C : \Pi_B \;\middle|\; R,Z,\Pi_{(\leqslant \ell)},\sigma\right]\\ \text{(by (1))}\quad &= \mathbf{H}[C \mid R,Z,\sigma] - \mathbf{I}\left[C : \Pi_B \;\middle|\; R,Z,\Pi_{(\leqslant \ell)},\sigma\right]. \end{aligned} \tag{2}$$

The next lemma shows that we can upper-bound the amount of information that Bob's transcript reveals about Charlie's output in terms of the information that the transcript reveals about the cut set E_σ, assuming that the protocol succeeds.

▶ **Lemma 7.** $\mathbf{I}\left[C : \Pi_B \;\middle|\; R,Z,\Pi_{(\leqslant \ell)},\sigma\right] \leqslant \mathbf{I}\left[E_\sigma : \Pi_B \;\middle|\; R,Z,\Pi_{(\leqslant \ell)},\sigma\right] + 3\sqrt{\delta}$.

[2] Omitted proofs are presented in the full version of the paper.

Next, we show that Bob's message reveals the same amount of information about any cut set (on average), which holds for E_σ in particular.

▶ **Lemma 8.** $\mathbf{I}\big[E_\sigma : \Pi_B \;\big|\; R, Z, \Pi_{(\leqslant \ell)}, \sigma\big] \leqslant \dfrac{|\Pi_B|}{\ell}$ *and also* $\mathbf{I}\big[E_\sigma : \Pi_B \;\big|\; Z, \Pi_{(\leqslant \ell)}, \sigma\big] \leqslant \dfrac{|\Pi_B|}{\ell}$.

Plugging the bound of Lemma 8 into Lemma 7, we obtain

$$\mathbf{I}\big[C : \Pi_B \;\big|\; R, Z, \Pi_{(\leqslant \ell)}, \sigma\big] \leqslant \frac{|\Pi_B|}{\ell} + 3\sqrt{\delta}.$$

Returning to (2), we get

$$\mathbf{I}\big[C : \Pi_{(\leqslant \ell)} \;\big|\; R, Z, \sigma\big] \geqslant \mathbf{H}[C \mid R, Z, \sigma] - \frac{|\Pi_B|}{\ell} - 3\sqrt{\delta}. \tag{3}$$

Intuitively speaking, (3) says that the transcript of Alice$_1, \ldots,$ Alice$_\ell$ reveal all except a fraction of a bit of the information contained in Charlie's output C, in expectation. However, we cannot directly use the assumed SMP protocol $\mathcal{P}$ for reconstructing Charlie's output, because $\mathcal{P}$ is only guaranteed to work given the transcripts of *all* players. Nevertheless, the next lemma shows that there exists a simple reconstruction protocol, which completes the proof of Lemma 5.

▶ **Lemma 9.** *There exists a protocol $\mathcal{R}$ that takes R, Z, $\Pi_{(\leqslant \ell)}$, and σ as input, and correctly computes Charlie's output C with probability at least $2^{-\left(\frac{|\Pi_B|}{\ell} + 3\sqrt{\delta}\right)}$.*

Proof. Protocol $\mathcal{R}$ works as follows: Given the input $\{\Pi_1 = \pi_1, \ldots, \Pi_\ell = \pi_\ell, R = r, Z = z, \sigma = i\}$, it returns the output of Charlie that maximizes the probability, which is

$$\arg\max_c \mathbf{Pr}[C = c \mid \pi_1, \ldots, \pi_\ell, r, z, i].$$

Let $Y = (\Pi_{(\leqslant \ell)}, R, Z, \sigma)$, and let $p_{c|y} = \mathbf{Pr}[C = c \mid Y = y]$. We observe that

$$\begin{aligned}
\mathbf{Pr}[\mathcal{R} \text{ succeeds}] &= \sum_y \mathbf{Pr}[Y = y] \max_c p_{c|y} \\
&= \mathop{\mathbf{E}}_y\big[2^{\log \max_c p_{c|y}}\big] \\
\text{(by Jensen's inequality)} \quad &\geqslant 2^{\mathbf{E}[\log \max_c p_{c|y}]} \\
&= 2^{\mathbf{E}\left[\log\left(\max_c (p_{c|y}) \cdot \sum_c p_{c|y}\right)\right]} \\
&\geqslant 2^{\mathbf{E}\left[\log\left(\sum_c p_{c|y}^2\right)\right]}.
\end{aligned} \tag{4}$$

For a fixed y, we define the random variable $P_y(c) := p_{c|y}$, which is a function of c. In this notation, the exponent on the right-hand side becomes

$$\begin{aligned}
\mathop{\mathbf{E}}_y\left[\log\left(\sum_c p_{c|y} P_y(c)\right)\right] &= \mathop{\mathbf{E}}_y\Big[\log\Big(\mathop{\mathbf{E}}_c[P_y(c)]\Big)\Big] \\
\text{(by Jensen's inequality)} \quad &\geqslant \mathop{\mathbf{E}}_y\Big[\mathop{\mathbf{E}}_c[\log P_y(c)]\Big] \\
&= -\mathop{\mathbf{E}}_y[\mathbf{H}[C \mid Y = y]] \\
&= -\mathbf{H}[C \mid Y]
\end{aligned}$$

Returning to (4), we get

$$\mathbf{Pr}[\mathcal{R} \text{ succeeds}] \geqslant 2^{-\mathbf{H}[C \mid Y]} = 2^{-\mathbf{H}[C \mid \Pi_{(\leqslant \ell)}, R, Z, \sigma]}.$$

Since $\mathbf{H}[C \mid \Pi_{(\leqslant \ell)}, R, Z, \sigma] = \mathbf{H}[C \mid R, Z, \sigma] - \mathbf{I}\big[C : \Pi_{(\leqslant \ell)} \bigm| R, Z, \sigma\big]$, it follows that

$$\begin{aligned}\mathbf{Pr}[\mathcal{R} \text{ succeeds}] &\geqslant 2^{-\left(\mathbf{H}[C \mid R,Z,\sigma] - \mathbf{I}\left[C : \Pi_{(\leqslant \ell)} \mid R,Z,\sigma\right]\right)} \\ \text{(by (3))} \quad &\geqslant 2^{-\left(\frac{|\Pi_B|}{\ell} + 3\sqrt{\delta}\right)}.\end{aligned}$$

◀

3 A Lower Bound for Computing a BFS Tree

As a warm-up, we instantiate the generic class of lower bound graphs defined in Section 2.1 to show a tight bound on the message length for computing a breadth-first search (BFS) tree, where a fixed node s starts out knowing that it is designated as the source and the goal for the referee is to output a BFS tree rooted at s.

▶ **Theorem 1** (restated). *Any public coin constant-error randomized algorithm that computes a BFS tree rooted at a designated node of an n-node graph, requires a worst case message length of $\Omega(n)$ bits in the distributed sketching model* (SKETCH) *and the one-round broadcast congested clique* (BCC_1)*.*

Proof. We are able to obtain this lower bound via a direct reduction from the Index_N problem in the two-party one-way setting, where there are two players, Diane and Edward. Diane starts with a binary vector $\mathbf{x}$ of length N and Edward gets an index $i \in [N]$. Diane can send a single message to Edward who must output the i-th bit of $\mathbf{x}$.

As discussed in Section 1, the models SKETCH and BCC_1 are equivalent and we will focus on the former out of convenience. Suppose that there is a SKETCH algorithm $\mathcal{A}$ that computes a BFS tree rooted at any given source node. We describe how Diane and Edward can simulate $\mathcal{A}$ to solve the Index_{ℓ^2} problem. Based on the lower bound graph class $\mathcal{G}_\ell$ that we described in Section 2.1, they sample a graph as follows: All the IDs of the nodes are fixed and the cardinalities of the vertex sets are defined as $|U| = |V| = |W| = \ell$. Moreover, there is a fixed perfect matching between U and V known to both players, i.e., we have edges $(u_1, v_1), \dots, (u_\ell, v_\ell)$. Assume that Diane gets input $\mathbf{x}$, which is a binary vector of length ℓ^2. Diane interprets her input $\mathbf{x}$ as the characteristic vector of the ℓ^2 possible edges between the sets V and W, for the fixed ordering ρ of $V \times W$ where $\rho = ((v_1, w_1), \dots, (v_1, w_\ell), (v_2, w_1), \dots, (v_2, w_\ell), \dots, (v_\ell, w_\ell))$. That is, Diane adds the i-th edge of ρ to the graph if and only if $\mathbf{x}_i = 1$. As a result, Diane knows the neighborhoods of all nodes in $V \cup W$. She simulates $\mathcal{A}$ on each one of them and sends the resulting messages to Edward.

Edward gets as input some index $i \in [\ell^2]$. Since he knows the ordering ρ, he computes the index $\sigma \in [\ell]$ such that $v_\sigma \in V$ is incident to the i-th (potential) edge in ρ, and adds the edge (s, u_σ). Figure 3 in the attached full paper shows the resulting graph. Then, he simulates $\mathcal{A}$ on s and each vertex in U, whereby s is designated as the source node of the tree. Upon receiving Diane's message, he simulates the referee and obtains the BFS tree assuming that $\mathcal{A}$ succeeded. If the i-th edge is included in the BFS-edges leading from v_σ to W, he outputs 1, otherwise he answers 0. Correctness follows since the BFS tree rooted at s must contain all the edges in the cut (v_σ, W).

It was shown in [20] that the Index_{ℓ^2} problem requires $\Omega(\ell^2)$ bits in the one-way two-party model, for achieving constant probability of success. Therefore, Diane's simulation produces a message of length $\Omega(\ell^2)$ bits, and thus one of the 2ℓ vertices simulated by her must have sent a message of size $\Omega(\ell)$ bits. The result follows since the lower bound graph has $n = 3\ell + 1$ vertices in total. ◀

4 A Lower Bound for Verifying Symmetry Breaking Problems

We now turn our attention to the problem of verifying whether a given labeling of the vertices is a weak 2-coloring of the input graph, which means that each non-isolated vertex has at least one differently-colored neighbor.[3]

High-level Overview. As we plan to employ Lemma 5, we start by defining the EI_m problem in the SMP model and show that it is an embeddable problem satisfying Properties (P1), (P2), and (P3) with a suitable input distribution on the graphs in $\mathcal{G}_\ell$. Next, we show how to simulate a given EI_m algorithm in the one-way two-party communication complexity model for solving set disjointness. From this, we derive a lower bound on the length of Alice$_\sigma$'s message. We obtain the sought lower bound by showing that a protocol for 2-weak coloring can be used to solve the EI_m problem in the SMP model.

4.1 The Edge Intersection Problem EI_m

Here, in addition to vertex IDs, we assume that each vertex in W is labeled with a bit indicating its color, and we define $W_b \subseteq W$ to be the b-labeled vertices, for $b \in \{0, 1\}$. As defined in Section 2, random variable $\mathcal{L}_W$ represents the label assignment for nodes in W. We consider the simultaneous multiparty (SMP) model with the input assignments as described in Section 2.2, i.e., Alice$_i$ knows the neighborhood of v_i, Bob knows all nodes (and labels) in W as well as their neighbors, and Charlie knows the IDs of U, W, the labels $\mathcal{L}_W$, and the embedding index σ. Charlie receives a message from Alice$_1, \ldots,$ Alice$_\ell$ and Bob, and then computes his answer. The goal is to determine whether an edge in E_σ "intersects" with (i.e., has an endpoint in) the 1-labeled nodes in W. Thus, to correctly solve the EI_m problem, it must hold for Charlie's output that

$$C = \begin{cases} \text{"yes"} & \text{if } E_\sigma \cap W_1 \neq \emptyset; \\ \text{"no"} & \text{otherwise.} \end{cases} \tag{5}$$

4.2 The Hard Input Distribution $\mathcal{D}_{\mathsf{EI}_m}$

Let $\ell = \lceil m^3 \log m \rceil$. We define the following distribution $\mathcal{D}_{\mathsf{EI}_m}$ on the class $\mathcal{G}_\ell$. We fix the IDs of all vertices in advance, i.e., they are the same for all graphs sampled from $\mathcal{D}_{\mathsf{EI}_m}$. In particular, we specify that $|W| = m^2$ and we assign the set $[m^2]$ as the IDs of the vertices in W. The sets U_i are singletons, i.e., $U_i = \{u_i\}$, and there is a perfect matching $\{u_1, v_1\}, \ldots, \{u_\ell, v_\ell\}$ between U and V.

We will choose the edges in the cut sets $E_1, \ldots, E_\sigma$ and the labels of W by sampling the input from the product distribution on certain set families for which set disjointness is known to be hard:

▶ **Lemma 10** (follows from Lemmas 1 and 2 in [9]). *There exist set families $\mathcal{X}, \mathcal{Y} \subseteq \binom{[m^2]}{m}$*[4] *such that (a) $|\mathcal{X}| \geqslant 2^{(m \log m)/4}$, (b) $|\mathcal{Y}| \leqslant \frac{1}{4} m \log m$. Moreover, for all distinct $X, X' \in \mathcal{X}$, it holds that (c) $|X \cap X'| \leqslant \frac{m}{4}$, and (d) there exists $Y \in \mathcal{Y}$ such that Y has a nonempty intersection with either X or X'.*

[3] The weak 2-coloring problem was introduced in the seminal work of [21].
[4] $\binom{[N]}{m}$ denotes the family of all m-element subsets of $[N]$.

We make use of the set families guaranteed by Lemma 10 to sample a graph G from $\mathcal{D}_{\mathsf{EI}_m}$ as follows:

1. Sample σ uniformly from $[\ell]$.
2. For each v_i, sample a random set $X \in \mathcal{X}$ and connect v_i to each $w \in W$ that has an ID in X.
3. Randomly pick a set $Y \in \mathcal{Y}$ and label the nodes in W according to the output of the resulting indicator function on W: That is, for $j \in [m^2]$, the label of w_j is 1 if $j \in Y$ and 0 otherwise.

Figure 4a shows a graph sampled from $\mathcal{D}_{\mathsf{EI}_m}$.

▶ **Lemma 11.** *Problem EI_m is embeddable with input distribution $\mathcal{D}_{\mathsf{EI}_m}$, and satisfies Properties (P1), (P2), and (P3) (as defined in Section 2).*

4.3 A Lower Bound for the EI_m Problem

To prove that the EI_m problem requires a large transcript length, we use a reduction from set disjointness.

▶ **Lemma 12** (implicit in Theorem 4 in [9])**.** *Solving set disjointness in the one-way two-party model with a public coin randomized protocol that succeeds with probability $\frac{1}{2}+\epsilon$, for some constant $\epsilon > 0$, has a communication complexity of $\Omega(m \log m)$ bits, when Diane's input is sampled uniformly from $\mathcal{X}$ and Edward's input is sampled uniformly from $\mathcal{Y}$.*

▶ **Lemma 13.** *Consider a public coin randomized protocol $\mathcal{P}$ that solves the EI_m problem with error $\delta \leqslant \frac{1}{25}$. If $|\Pi_B| \leqslant \frac{1}{16}m^3 \log m$, then $|\Pi_\sigma| = \Omega(m \log m)$.*

Proof. We show a reduction from the set disjointness problem [26] in the one-way two-party model, where there are two players, Diane and Edward that are given subsets X and Y respectively. Diane sends a single message to Edward who must decide whether $X \cap Y = \emptyset$.

Given an instance of set disjointness, Diane and Edward will simulate the assumed EI_m protocol $\mathcal{A}$ on a graph sampled from $\mathcal{D}_{\mathsf{EI}_m}$ by embedding the set disjointness instance into the neighborhood of node v_σ. Suppose that Diane has input $X \in \mathcal{X}$ and Edward has input $Y \in \mathcal{Y}$, both of which were sampled uniformly. As required, they choose the cardinalities $|U| = \ell$ and $|W| = m^2$, and make each U_i a singleton set. Moreover, they fix the IDs of all nodes in advance such that the set $[m^2]$ defines the IDs of the nodes in W. Note that this also determines the perfect matching between U and V. In the simulation, Diane will simulate only Alice$_\sigma$, whereas Edward simulates Alice$_i$ $(i \neq \sigma)$ and Charlie. Note that, in addition to the edges, the players also need to assign binary vertex labels to the vertices in W:

1. Using public randomness, they uniformly sample an index σ from $[\ell]$.
2. Diane uses her input X to define the IDs of the neighbors of v_σ in W.
3. Similarly, for each $v_i \in V$ $(i \neq \sigma)$, Edward uniformly samples a random $X_i \in \mathcal{X}$ and connects v_i to W by connecting v_i to each $w \in W$ with an ID in X_i.
4. Edward uses his input Y to assign the labels of the nodes in W. That is, for each index $j \in Y$, the label of w_j is 1, and he labels all w_k $(k \notin Y)$ with 0.

It is straightforward to verify that the sampling procedure executed by Diane and Edward results in an input assignment to players Alice$_1, \dots,$ Alice$_\ell$, and Charlie that is the same as in distribution $\mathcal{D}_{\mathsf{EI}_m}$. Therefore, Diane can simulate the EI_m protocol for Alice$_\sigma$ and send the resulting message to Edward who, in turn, is able to simulate Alice$_i$ $(i \neq \sigma)$. Once Edward receives Diane's message, he knows Z, $\Pi_{(\leqslant \ell)}$, and σ, and hence he also knows Charlie's input. Since $|\Pi_B| \leqslant \frac{1}{16}m^3 \log m$ and $\ell = \lceil m^3 \log m \rceil$ in $\mathcal{D}_{\mathsf{EI}_m}$, it follows that $\frac{|\Pi_B|}{\ell} \leqslant \frac{1}{16}$. Thus, Edward invokes the reconstruction protocol $\mathcal{R}$ guaranteed by Lemma 5 to recover Charlie's output. He answers that X and Y are disjoint if and only if Charlie's output is "no". Since $\mathcal{R}$ succeeds with probability at least $2^{-\left(\frac{|\Pi_B|}{\ell}+3\sqrt{\delta}\right)} \geqslant 2^{-(1/16+3/5)} > 0.63$, applying Lemma 12 completes the proof of Lemma 13. ◀

4.4 Proof of Theorem 2

▶ **Theorem 2** (restated). *Any $\frac{1}{25}$-error randomized algorithm that verifies if a labeling of a subset of vertices forms a weak 2-coloring of an n-node input graph, requires a worst case message length of $\Omega\left(n^{1/3}\log^{2/3} n\right)$ bits in* SKETCH *and* BCC$_1$. *The same bound holds for deciding whether a subset of nodes forms a maximal independent set.*

We prove the theorem via a reduction from the EI_m problem in the SMP model. Let G be an input graph sampled from $\mathcal{D}_{\mathsf{EI}_m}$ and let $\mathcal{Q}$ be a protocol that satisfies the premise of the theorem. The players will simulate $\mathcal{Q}$ on a graph H, which extends G with some edges and adds a vertex coloring, as we describe in more detail below. Each player Alice$_i$ simulates $\mathcal{Q}$ for node v_i, while assigning color 0 to v_i. Bob simulates all nodes in W and adds an edge between some arbitrary node in $w \in W_1$ and every node in W_0. For the simulation, the nodes in W_b ($b \in \{0,1\}$) are colored with color b. See Figure 4b for an example of the resulting graph. Charlie, on the other hand, simulates the referee and all nodes in U, where he colors u_σ with 0 and the nodes in $U \setminus \{u_\sigma\}$ with 1. Moreover, he adds an edge between u_σ and some arbitrary u_j ($j \neq \sigma$). Note that Charlie knows which node is u_σ since the index σ is part of his input. The edges added by Bob and Charlie ensure that every node (with the possible exception of v_σ) has a differently-colored neighbor by construction. It follows that the output of the EI_m protocol verifies whether the given coloring of G is valid:

▶ **Observation 14.** *The coloring is a weak 2-coloring if and only if $E_\sigma \cap W_1$ is nonempty. An analogous property holds for the question whether the vertices with color 1 form a maximal independent set or a minimal dominating set.*

Now, assume towards a contradiction that the worst case sketch length produced by protocol $\mathcal{Q}$ is at most $\frac{1}{16}m \log m$. This ensures that Bob sends a message of at most $\frac{1}{16}|W|m \log m = \frac{1}{16}m^3 \log m$ bits in the simulation. Since this satisfies the premise of Lemma 13, it follows that the node v_σ simulated by Alice$_\sigma$ sends a sketch of length $\Omega(m \log m)$ in the worst case. The total number of nodes in H is $n = |U| + |V| + |W| = 2\lceil m^3 \log m \rceil + m^2 = \Theta\left(m^3 \log m\right)$. Hence it follows that $\log m = \Omega(\log n)$ and thus $m = \Omega\left((n/\log n)^{1/3}\right)$. We conclude that Alice$_\sigma$'s sketch must have a length of $\Omega(n^{1/3}\log^{2/3} n)$ bits. By Observation 14, the same result holds for verifying a maximal independent set or a minimal dominating set.

5 A Lower Bound for k-ECSS in Sketching Model

In this section, we will apply Lemma 6 for showing a lower bound of $\Omega\left(k \log^2 \frac{n}{k}\right)$ on the message size for computing a k-edge connected spanning subgraph (k-ECSS).

High-level Overview. We first define an embeddable problem, the $\mathsf{ER}_{k,m}$ problem, and a suitable input distribution on the lower bound graphs $\mathcal{G}_\ell$ (see Section 2.1). We consider the $\mathsf{ER}_{k,m}$ problem in the SMP model, where each vertex in V has m neighbors and the goal is to find a subset of k edges in the cut E_σ of the input graph. Subsequently, we show that it is in fact an embeddable problem (see Sec. 2) that satisfies Properties (P1) and (P2), and thus Lemma 6 applies. We then simulate the assumed $\mathsf{ER}_{k,m}$ protocol in the one-way two-party communication model and use it to solve the $\mathsf{UR}_k^{\subset}$ problem defined in [17], which implies a lower bound on the length of Alice$_\sigma$'s message. The final step is to simulate a given k-ECCS protocol $\mathcal{P}$ designed for the SKETCH model to solve the $\mathsf{ER}_{k,m}$ problem in the SMP model, and this will yield the sought lower bound on the worst case sketch size of $\mathcal{P}$.

5.1 The Edge Recovery Problem $\mathsf{ER}_{k,m}$

We consider the simultaneous multiparty model and the graph class $\mathcal{G}_\ell$, where each vertex in V has exactly m neighbors. A protocol solves the $\mathsf{ER}_{k,m}$ problem if, after receiving the messages from Alice$_1, \dots,$ Alice$_\ell$, and Bob, player Charlie outputs a subset of k edges in the cut E_σ.

5.2 The Hard Input Distribution $\mathcal{D}_{\mathsf{ER}_{k,m}}$

Our distribution is similar to the one used in [22], albeit with some crucial differences. Let $\ell := \lceil m^2 \log^2 \frac{m}{k} \rceil$ and let $\Gamma = \lceil m^2/k \rceil$. To sample a graph $G \in \mathcal{G}_\ell$ from $\mathcal{D}_{\mathsf{ER}_{k,m}}$, we fix the IDs of nodes in V to be the set $[\ell]$ and perform the following steps:

1. Uniformly sample σ from $[\ell]$.
2. Fix the size of the sets $U_1, \dots, U_\ell$, and W to be Γ. Sample $(\ell+1)$ disjoint random subsets $A_1, \dots, A_{\ell+1}$, each of size Γ from $F_0 := [\ell^2] \setminus [\ell]$, and use A_i to assign the IDs to the nodes in U_i, whereas, for the nodes in W, we use $A_{\ell+1}$.
3. For each $v_i \in V$, we choose a random m-element set $S \subseteq [\Gamma]$ and a uniformly random $T \subset S$ such that $|S \setminus T| \geqslant k$ and $|T| \geqslant k$. We connect v_i to $|S \setminus T|$ random vertices from W and $|T|$ random vertices in U_i.

Figure 5a shows an instance of a graph sampled from $\mathcal{D}_{\mathsf{ER}_{k,m}}$.

▶ **Lemma 15.** *The total number of nodes in graph G is $n = O\big(m^4 \log^2(m/k)\big)$.*

▶ **Lemma 16.** *The $\mathsf{ER}_{k,m}$ problem is embeddable with distribution $\mathcal{D}_{\mathsf{ER}_{k,m}}$, i.e., satisfies properties (P1) and (P2).*

5.3 A Lower Bound for the $\mathsf{ER}_{k,m}$ Problem

We use Lemma 6 to show the following:

▶ **Lemma 17.** *Consider a deterministic protocol $\mathcal{P}$ that solves the $\mathsf{ER}_{k,m}$ problem with probability at least $1 - o(1)$ on inputs sampled from $\mathcal{D}_{\mathsf{ER}_{k,m}}$, and suppose that $|\Pi_B| = o(\ell)$. Then, there exists a deterministic protocol $\mathcal{R}$ that succeeds with probability at least $1 - o(1)$ on inputs from $\mathcal{D}_{\mathsf{ER}_{k,m}}$, just by inspecting Charlie's input and the transcripts of Alice$_1, \dots,$ Alice$_\ell$, i.e., while omitting Bob's transcript Π_B.*

The $\mathsf{UR}_k^{\subset}$ Problem. There are two players, Diane and Edward. For some integer $N > 0$, Diane is given a set $S \subseteq [N]$ and Edward starts with a subset $T \subset S$. To solve the $\mathsf{UR}_k^{\subset}$ problem, Diane sends a single message to Edward, who in turn must output k elements in $S \setminus T$. Theorem 3 in [17] shows a worst case communication complexity lower bound of $\Omega(k \log^2(N/k))$ bits for algorithms that succeed with constant probability. For the purpose of our reduction, we need a slightly more specific result:

In the full version of the paper, we show how to adapt Theorem 3 in [17] to obtain the following result:

▶ **Lemma 18.** *Consider a universe of size N. Let $m = \lfloor\sqrt{Nk}\rfloor$ and suppose that $k \leqslant N/2^{10}$. Suppose that Diane's set S is chosen from $\binom{[N]}{m}$ and Edward's input set is any proper subset $T \subset S$ under the restriction that $|S \setminus T| \geqslant k$ and $|T| \geqslant k$. Then, any $\mathsf{UR}_k^{\subset}$ protocol $\mathcal{P}$ that errs with small constant probability in the one-way 2-party model with public coins, has a worst case transcript length of $\Omega(k \log^2(\frac{N}{k}))$ bits.*

In the proof of the next lemma, we use the hardness of $\mathsf{UR}_k^{\subset}$ to show a lower bound for the $\mathsf{ER}_{k,m}$ problem by simulating the SMP model in the 2-party setting. We point out that the approach is similar to Section 3 of [22] and postpone the full proof to the full version.

▶ **Lemma 19.** *Consider a protocol $\mathcal{P}$ that solves the $\mathsf{ER}_{k,m}$ problem with error at most $o(1)$ on inputs sampled from $\mathcal{D}_{\mathsf{ER}_{k,m}}$. If $|\Pi_B| = o(\ell)$, where $\ell = \lceil m^2 \log^2 \frac{m}{k}\rceil$, then $|\Pi_\sigma| = \Omega\big(k \log^2 \frac{m}{k}\big)$.*

5.4 Proof of Theorem 3

▶ **Theorem 3** (restated). *Any public coin randomized algorithm that computes a k-edge connected spanning subgraph of an n-node graph in SKETCH or BCC_1 with probability at least $1 - o(1)$, has a worst case message length of $\Omega\big(k \log^2 \frac{n}{k}\big)$ bits, for any $k = o\Big(\frac{n^{1/4}}{\log^{1/2} n}\Big)$.*

We first describe the hard input distribution $\mathcal{D}_{k\text{-ECSS}}$ for computing a k-connected spanning subgraph in the SKETCH model, which is a simple extension of $\mathcal{D}_{\mathsf{ER}_{k,m}}$:

1. Sample a graph G from $\mathcal{D}_{\mathsf{ER}_{k,m}}$.
2. Graph H contains all edges of G; in addition, we make the subgraph induced by each U_i ($i \in [\ell]$) a clique, and we also add a clique on W.

Let $\mathcal{A}$ be a deterministic algorithm that computes a k-connected spanning subgraph in the SKETCH model on inputs sampled from $\mathcal{D}_{k\text{-ECSS}}$ with probability at least $1 - o(1)$. Note that the result immediately extends to randomized algorithms by a simple application of Yao's lemma. Given a graph G sampled from the hard input distribution $\mathcal{D}_{\mathsf{ER}_{k,m}}$, we add the necessary edges to G according to $\mathcal{D}_{k\text{-ECSS}}$ and then simulate $\mathcal{A}$ on the resulting graph H to solve the $\mathsf{ER}_{k,m}$ problem in the SMP model. Figure 5b shows an example of this graph.

Observe that every U_i and W consist of $\lceil m^2/k\rceil > k$ vertices and hence there are k edge-disjoint paths between any two vertices that lie within such a set. Moreover, we sample the neighborhoods of v_i such that $|E(v_i, W)| \geqslant k$ and $|E(v_i, U_i)| \geqslant k$ (see Step 3 of $\mathcal{D}_{\mathsf{ER}_{k,m}}$). This ensures that there are at least k edge-disjoint paths between all pairs of vertices of H:

▶ **Observation 20.** *Graph H is k-edge connected.*

For each $i \in [\ell]$, player Alice$_i$ simulates $\mathcal{A}$ for node v_i and sends the corresponding sketch to Charlie. Bob, on the other hand, sends Charlie the concatenated sketches of the nodes in W, which he computes by simulating $\mathcal{A}$ given their respective neighborhood in H as an input. Finally, Charlie, simulates $\mathcal{A}$ for all nodes in U, and he also simulates the referee. It follows immediately from the input assignment of the $\mathsf{ER}_{k,m}$ problem that the players have the necessary information to perform the simulation. Observe that every k-edge connected subgraph of H must include k edges in the cut $E(v_\sigma, W)$, and hence the simulation solves the $\mathsf{ER}_{k,m}$ problem with the same probability of success.

Let L be the worst case sketch length of protocol $\mathcal{A}$. In our simulation, Bob's message is of length $|\Pi_B| \leqslant L\,|W| \leqslant O\Big(\frac{L\,m^2}{k^2}\Big)$. Assume that $L = o\big(k \log^2 \frac{m}{k}\big)$, as otherwise we are done. By Lemma 15, we know that $\log \frac{m}{k} = \Omega\big(\log \frac{n}{k}\big)$, which implies that $|\Pi_B| = o\big(m^2 \log^2 \frac{n}{k}\big) = o(\ell)$. By applying Lemma 19, we conclude that Alice$_\sigma$ must send a message of $\Omega\big(k \log^2 \frac{m}{k}\big) = \Omega\big(k \log^2 \frac{n}{k}\big)$ bits in the worst case.

6 A Streaming Lower Bound for k-ECSS

In this section, we consider the data streaming setting, where the algorithm learns about the input graph as a stream of edges. That is, in the *fully dynamic turnstile model*, the algorithm observes the stream entries sequentially. Each entry of the stream refers to two vertices u and v and indicates whether the edge $\{u, v\}$ is added or removed from the current graph, and the algorithm needs to react to this update. The main objective is to minimize the amount of memory used by the algorithm while taking (preferably) only a single pass over the data stream.

We show a memory lower bound for computing a k-ECSS in the fully dynamic turnstile model by extending the work of [17].

▶ **Theorem 4** (restated). *Any Monte Carlo data structure for computing a k-edge connected spanning subgraph of an n-node graph requires $\Omega\big(k\,n \log^2 \frac{n}{k}\big)$ space in the one-pass fully dynamic turnstile model.*

In our proof of Theorem 4, we use a reduction from the 2-party communication complexity, where we need to solve multiple instances of $\mathsf{UR}_k^{\subset}$ in parallel. We first recall the definition of the $\mathsf{UR}_k^{\subset}$ problem from Section 5: We are given the universe $[N]$ and there are two players called Alice and Bob. Alice obtains a set $S \subseteq [N]$ and Bob has a subset $T \subset S$. Alice sends a message to Bob who must then output k elements in $S \setminus T$.[5] We define the *ℓ-fold* $\mathsf{UR}_k^{\subset}$ *problem*, where Alice and Bob obtain ℓ independently sampled instances of $\mathsf{UR}_k^{\subset}$ (on the same universe) and they need to solve all of them, again, assuming that Alice can send only a single message to Bob.

▶ **Lemma 21.** *Consider any $k = \Omega(\log N)$ and a universe of size $N > k$. Any one-way communication protocol that solves the ℓ-fold $\mathsf{UR}_k^{\subset}$ problem with error at most δ requires $\Omega\big((1-\delta)k\,\ell\,\log^2\big(\frac{N}{k}\big)\big)$ bits.*

We now show how the lemma implies Theorem 4: Suppose that there exists an algorithm $\mathcal{A}$ that maintains a k-edge connected spanning subgraph in the turnstile model. We simulate $\mathcal{A}$ in the 2-party model. Our simulation is similar to the one used for showing a lower bound on the memory needed for maintaining a spanning forest in Lemma 1 of [22]. Consider a graph G with vertex sets X and Y, each of size ℓ, for some $\ell > k$. The IDs of the vertices in X are given by $[\ell]$, whereas the neighborhood of the i-th vertex in Y will correspond to the i-th instance of $\mathsf{UR}_k^{\subset}$. Recall that Alice starts with the input $S_1, \dots, S_\ell$ and Bob has input $T_1, \dots, T_\ell$, where (S_i, T_i) is the i-th instance of $\mathsf{UR}_k^{\subset}$. Alice and Bob will perform edge insertions/removals and execute the streaming algorithm $\mathcal{A}$ accordingly. Alice first inserts edges such that X forms a clique. Then, for each set S_i and each $x \in S_i$, Alice adds an edge $\{x, y_i\}$ to G. Subsequently, Alice sends the memory state of $\mathcal{A}$ to Bob who, in turn, for each T_i and each $x' \in T_i$, continues to simulate $\mathcal{A}$ by removing $\{x', y_i\}$ from G. Recall that $|S_i \setminus T_i| \geqslant k$, which guarantees that the degree of each node in Y is at least k after the last update. Moreover, the nodes in X form a clique of size greater than k, and hence it follows that G is k-edge connected. Finally, Bob returns the output of $\mathcal{A}$ which must include k edges incident to each $y_i \in Y$ with probability at least $1-\delta$ to ensure k-connectivity. Each one of these edges corresponds to an element in $S_i \setminus T_i$, and hence the simulation solves ℓ-fold $\mathsf{UR}_k^{\subset}$ with precisely the same probability. Since Lemma 21 tells us that ℓ-fold $\mathsf{UR}_k^{\subset}$ has a communication complexity of $\Omega(k\,n \log^2(n/k))$ for $\ell = n$, it follows that $\mathcal{A}$ must use at least $\Omega(k\,n \log^2(n/k))$ bits of memory. This completes the proof of Theorem 4.

[5] Here, we restrict ourselves to the case where the inputs satisfy that $|S \setminus T| \geqslant k$.

7 Future Work and Open Problems

Our results reveal insights into the communication complexity of distributed graph verification problems. However, we are not aware of any communication-efficient 1-round verification algorithm for these type of symmetry breaking problems.

▶ **Open Problem 1.** Is there an algorithm that verifies an LCL problem in just a single round of the broadcast congested clique while sending $o(m)$ bits on graphs with m edges?

While we showed a lower bound on the memory for maintaining a k-edge connected spanning subgraph in the turnstile model, the more fundamental question regarding the space required to solve graph connectivity (i.e., $k = 1$) has yet to be answered, as pointed out in [28]:

▶ **Open Problem 2.** Is there a lower bound of $\Omega(n \log^3 n)$ memory for solving graph connectivity in the turnstile model?

References

1 Amir Abboud, Keren Censor-Hillel, Seri Khoury, and Christoph Lenzen. Fooling views: a new lower bound technique for distributed computations under congestion. *Distributed Comput.*, 33(6):545–559, 2020. doi:10.1007/s00446-020-00373-4.

2 Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Analyzing graph structure via linear measurements. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 459–467. SIAM, 2012.

3 Sepehr Assadi, Gillat Kol, and Rotem Oshman. Lower bounds for distributed sketching of maximal matchings and maximal independent sets. In *PODC '20: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, August 3-7, 2020*, pages 79–88, 2020. doi:10.1145/3382734.3405732.

4 Baruch Awerbuch, Oded Goldreich, David Peleg, and Ronen Vainish. A trade-off between information and communication in broadcast protocols. *J. ACM*, 37(2):238–256, 1990. doi:10.1145/77600.77618.

5 Florent Becker, Martin Matamala, Nicolas Nisse, Ivan Rapaport, Karol Suchan, and Ioan Todinca. Adding a referee to an interconnection network: What can (not) be computed in one round. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 508–514. IEEE, 2011.

6 Matthias Bonne and Keren Censor-Hillel. Distributed detection of cliques in dynamic networks. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*, volume 132 of *LIPIcs*, pages 132:1–132:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ICALP.2019.132.

7 Lijie Chen and Ofer Grossman. Broadcast congested clique: Planted cliques and pseudorandom generators. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 248–255, 2019.

8 T. Cover and J.A. Thomas. *Elements of Information Theory, second edition.* Wiley, 2006.

9 Anirban Dasgupta, Ravi Kumar, and D Sivakumar. Sparse and lopsided set disjointness via information theory. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 517–528. Springer, 2012.

10 Andrew Drucker, Fabian Kuhn, and Rotem Oshman. On the power of the congested clique model. In *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*, pages 367–376, 2014. doi:10.1145/2611462.2611493.

11 Orr Fischer, Tzlil Gonen, Fabian Kuhn, and Rotem Oshman. Possibilities and impossibilities for distributed subgraph detection. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures, SPAA 2018, Vienna, Austria, July 16-18, 2018*, pages 153–162, 2018. doi:10.1145/3210377.3210401.

12 Pierre Fraigniaud, Pedro Montealegre, Pablo Paredes, Ivan Rapaport, Martín Ríos-Wilson, and Ioan Todinca. Computing power of hybrid models in synchronous networks. *arXiv preprint arXiv:2208.02640*, 2022.

13 Mohsen Ghaffari and Fabian Kuhn. Distributed MST and broadcast with fewer messages, and faster gossiping. In Ulrich Schmid and Josef Widder, editors, *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018*, volume 121 of *LIPIcs*, pages 30:1–30:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.DISC.2018.30.

14 Robert Gmyr and Gopal Pandurangan. Time-message trade-offs in distributed algorithms. In Ulrich Schmid and Josef Widder, editors, *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018*, volume 121 of *LIPIcs*, pages 32:1–32:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.DISC.2018.32.

15 Jacob Holm, Valerie King, Mikkel Thorup, Or Zamir, and Uri Zwick. Random k-out subgraph leaves only o(n/k) inter-component edges. In *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 896–909, 2019. doi:10.1109/FOCS.2019.00058.

16 Tomasz Jurdzinski, Krzysztof Lorys, and Krzysztof Nowicki. Communication complexity in vertex partition whiteboard model. In *International Colloquium on Structural Information and Communication Complexity*, pages 264–279. Springer, 2018.

17 Michael Kapralov, Jelani Nelson, Jakub Pachocki, Zhengyu Wang, David P Woodruff, and Mobin Yahyazadeh. Optimal lower bounds for universal relation, and for samplers and finding duplicates in streams. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 475–486. Ieee, 2017.

18 Bruce M. Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 1131–1142, 2013. doi:10.1137/1.9781611973105.81.

19 Valerie King, Shay Kutten, and Mikkel Thorup. Construction and impromptu repair of an MST in a distributed network with o(m) communication. In Chryssis Georgiou and Paul G. Spirakis, editors, *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*, pages 71–80. ACM, 2015. doi:10.1145/2767386.2767405.

20 Ilan Kremer, Noam Nisan, and Dana Ron. On randomized one-round communication complexity. *Computational Complexity*, 8(1):21–49, 1999.

21 Moni Naor and Larry J. Stockmeyer. What can be computed locally? *SIAM J. Comput.*, 24(6):1259–1277, 1995. doi:10.1137/S0097539793254571.

22 Jelani Nelson and Huacheng Yu. Optimal lower bounds for distributed and streaming spanning forest computation. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 1844–1860, 2019. doi:10.1137/1.9781611975482.111.

23 Shreyas Pai and Sriram V Pemmaraju. Connectivity lower bounds in broadcast congested clique. In *40th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

24 David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. Society for Industrial and Applied Mathematics, 2000. doi:10.1137/1.9780898719772.

25 Jeff M. Phillips, Elad Verbin, and Qin Zhang. Lower bounds for number-in-hand multiparty communication complexity, made easy. *SIAM J. Comput.*, 45(1):174–196, 2016. doi:10.1137/15M1007525.

26 Anup Rao and Amir Yehudayoff. *Communication Complexity: and Applications*. Cambridge University Press, 2020.

27 David P. Woodruff and Qin Zhang. When distributed computation is communication expensive. *Distributed Computing*, 30(5):309–323, 2017. doi:10.1007/s00446-014-0218-3.

28 Huacheng Yu. Tight distributed sketching lower bound for connectivity. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 1856–1873, 2021. doi:10.1137/1.9781611976465.111.

A Tools from Information Theory

We give the definitions of some basic notions from information theory and restate some facts (without proofs) that we use throughout the paper. We refer the reader to [8] for additional details. Throughout this section, we assume that X, Y, Z, etc. are discrete random variables. We use capitals to denote random variables and corresponding lowercase characters for values, unless stated otherwise. When computing expected values, we sometimes use the subscript notation $\mathbf{E}_x$ to make it explicit that the expectation is taken over the distribution of a specific random variable X.

▶ **Definition 22.** *The* entropy of X *is defined as*

$$\mathbf{H}[X] = \sum_x \mathbf{Pr}[X=x] \log_2(1/\mathbf{Pr}[X=x]). \tag{6}$$

The conditional entropy of X conditioned on Y *is given by*

$$\begin{aligned}\mathbf{H}[X \mid Y] &= \mathop{\mathbf{E}}_y[\mathbf{H}[X \mid Y=y]] \\ &= \sum_y \mathbf{Pr}[Y=y]\,\mathbf{H}[X \mid Y=y].\end{aligned} \tag{7}$$

▶ **Definition 23.** *Let X and Y be discrete random variables. The* mutual information between X and Y *is defined as*

$$\mathbf{I}[X:Y] = \sum_{x,y} \mathbf{Pr}[x,y] \cdot \log\left(\frac{\mathbf{Pr}[x,y]}{\mathbf{Pr}[x]\,\mathbf{Pr}[y]}\right) \tag{8}$$

▶ **Definition 24.** *Let X, Y, and Z be discrete random variables. The* conditional mutual information of X and Y *is defined as*

$$\mathbf{I}[X:Y \mid Z] = \mathbf{H}[X \mid Z] - \mathbf{H}[X \mid Y, Z]. \tag{9}$$

▶ **Lemma 25.** $\mathbf{I}[X:Y \mid Z] \leqslant \mathbf{H}[X \mid Z] \leqslant \mathbf{H}[X]$.

▶ **Lemma 26** (Theorem 6.1 in [26]). *Consider any random variable X. Every encoding of X has expected length at least $\mathbf{H}[X]$.*

▶ **Lemma 27** (Theorem 6.12 in [26]). *Let $X_1, \ldots, X_k$ be independent random variables, and let B be jointly distributed. Then,*

$$\sum_{i=1}^{k} \mathbf{I}[X_1 : B] \leqslant \mathbf{I}[X_1, \ldots, X_k : B].$$

▶ **Lemma 28** (Data Processing Inequality, see Theorem 2.8.1 in [8]). *If random variables X, Y, and Z form the Markov chain $X \to Y \to Z$, i.e., the conditional distribution of Z depends only on Y and is conditionally independent of X, then*

$$\mathbf{I}[X : Y] \geqslant \mathbf{I}[X : Z].$$

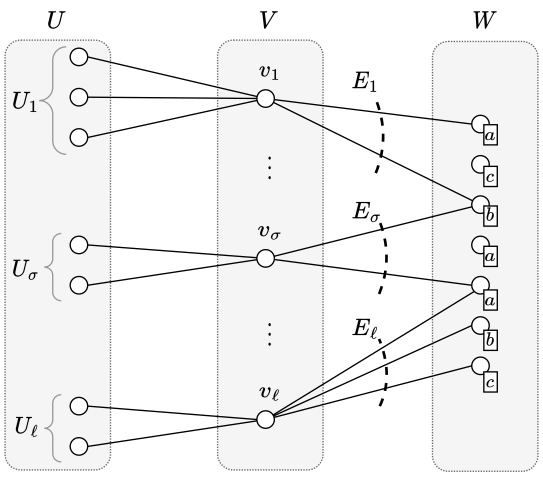
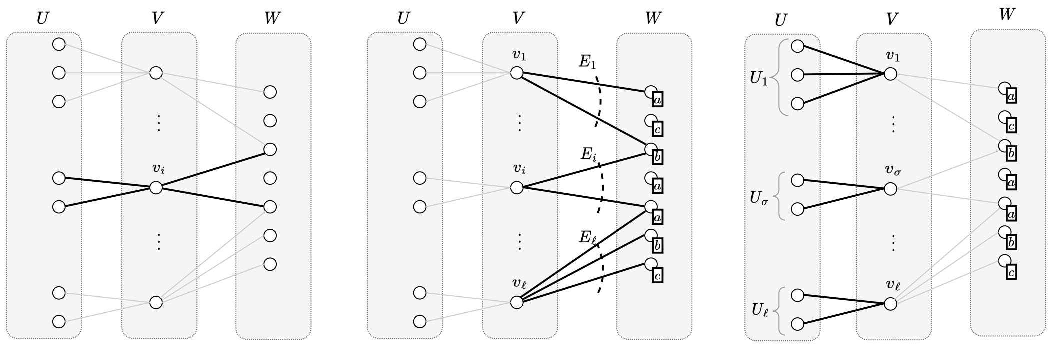


Figure 1 The general structure of the lower bound graphs in $\mathcal{G}_\ell$. Each v_i is connected to a subset of the vertices in U_i and to a subset of the vertices in W. Note that the cardinalities of the sets $U_1, \ldots, U_\ell$, and W, as well as the edges $E(U, V)$ and $E(V, W)$ depend on the hard input distribution, which is problem-specific. In this example, the labels of the nodes in W are chosen from $\{a, b, c\}$.



(a) Alice$_i$'s input: the entire neighborhood of v_i.

(b) Bob's input: $E(V, W)$ and $\mathcal{L}_W$.

(c) Charlie's input: $E(U, V)$, σ, and $\mathcal{L}_W$.

Figure 2 The input assignment in the SMP model.

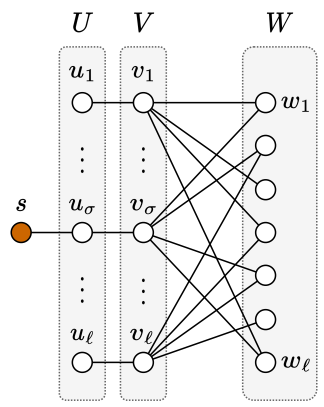
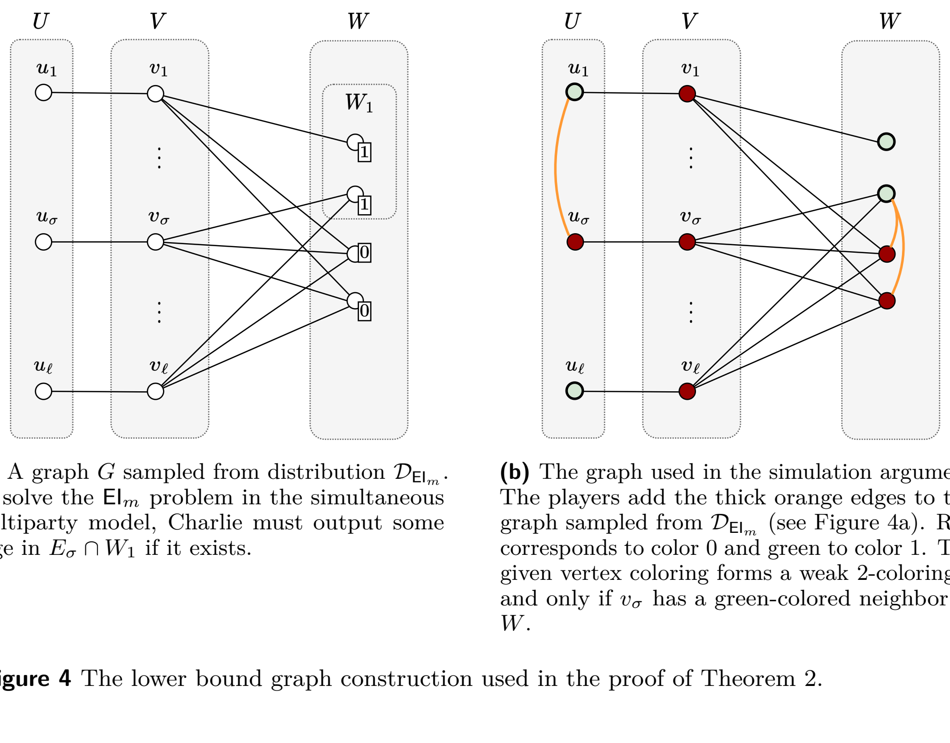


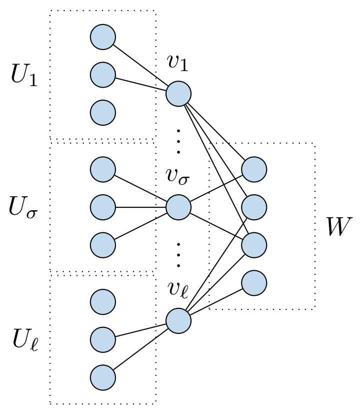
Figure 3 The lower bound graph G for proving the hardness of computing a BFS tree in the distributed sketching model and the one-round broadcast congested clique. The BFS tree rooted at s must include all edges in the cut $E(v_\sigma, W)$. Note that we sample the edges in the cut $E(V, W)$ according to the hard input distribution of the Index_{ℓ^2} problem.



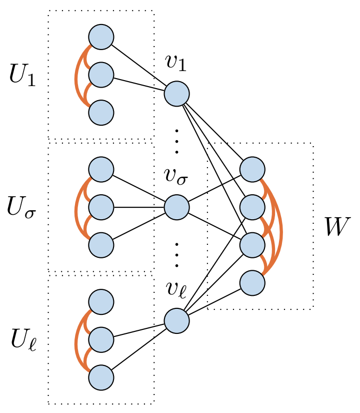
(a) A graph G sampled from distribution $\mathcal{D}_{\mathsf{EI}_m}$. To solve the EI_m problem in the simultaneous multiparty model, Charlie must output some edge in $E_\sigma \cap W_1$ if it exists.

(b) The graph used in the simulation argument. The players add the thick orange edges to the graph sampled from $\mathcal{D}_{\mathsf{EI}_m}$ (see Figure 4a). Red corresponds to color 0 and green to color 1. The given vertex coloring forms a weak 2-coloring if and only if v_σ has a green-colored neighbor in W.

Figure 4 The lower bound graph construction used in the proof of Theorem 2.



(a) A graph sampled from the lower bound distribution $\mathcal{D}_{\mathsf{ER}_{k,m}}$. The distribution ensures that, for all $i \in [\ell]$, the cuts $E(U_i, v_i)$ and $E(v_i, W)$ have at least $k = 2$ edges.



(b) A k-edge connected graph G that we use to prove a lower bound for the k-ECSS problem, for $k = 2$. To simulate a SKETCH algorithm, the players sample G from the lower bound distribution $\mathcal{D}_{\mathsf{ER}_{k,m}}$ and then add the thick orange edges to form cliques of size greater than k.

Figure 5 The lower bound graph construction of Theorem 3.

Memory-Anonymous Starvation-Free Mutual Exclusion: Possibility and Impossibility Results

Gadi Taubenfeld
Reichman University, Herzliya, Israel

Abstract

In an anonymous shared memory system, all inter-process communications are via shared objects; however, unlike in standard systems, there is no a priori agreement between processes on the names of shared objects [14, 15]. Furthermore, the algorithms are required to be symmetric; that is, the processes should execute precisely the same code, and the only way to distinguish processes is by comparing identifiers for equality. For such a system, read/write registers are called anonymous registers. It is known that symmetric deadlock-free mutual exclusion is solvable for any finite number of processes using anonymous registers [1]. The main question left open in [14, 15] is the existence of starvation-free mutual exclusion algorithms for two or more processes. We resolve this open question for memoryless algorithms, in which a process that tries to enter its critical section does not use any information about its previous attempts. Almost all known mutual exclusion algorithms are memoryless. We show that,

1. There is a symmetric memoryless starvation-free mutual exclusion algorithm for two processes using $m \geq 7$ anonymous registers if and only if m is odd.
2. There is no symmetric memoryless starvation-free mutual exclusion algorithm for $n \geq 3$ processes using (any number of) anonymous registers.

Our impossibility result is the only example of a system with fault-free processes, where global progress (i.e., deadlock-freedom) can be ensured, while individual progress to each process (i.e., starvation-freedom) cannot. It complements a known result for systems with failure-prone processes, that there are objects with lock-free implementations but without wait-free implementations [2, 5].

2012 ACM Subject Classification Theory of computation → Distributed computing models; Theory of computation → Shared memory algorithms; Theory of computation → Distributed algorithms

Keywords and phrases anonymous shared memory, memory-anonymous algorithms, anonymous registers, starvation-free mutual exclusion

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.33

1 Introduction

1.1 Anonymous shared memory

A central issue in distributed systems is coordinating the actions of asynchronous processes. In the context where processes communicate via reading and writing from shared memory, in almost all published concurrent algorithms, it is assumed that the shared memory locations have global names, which are a priori known to all the participating processes. The intriguing question of what and how coordination can be achieved without relying on such lower-level agreement about the names of the memory locations was introduced and studied in [14, 15].

We assume that all inter-process communications are via shared read/write registers which are initially in a known state. However, unlike in the standard model, from the point of view of the processes, the registers do not have global names. Such registers are called *anonymous* registers. Algorithms correct for a model where the registers are anonymous are called *memory-anonymous* algorithms.

There are fundamental differences between the standard shared memory model and the strictly weaker anonymous shared memory model [15]. Besides enabling us to understand better the intrinsic limits for coordinating the actions of asynchronous processes, the anonymous shared memory model with symmetric processes has been shown to be useful in modeling biologically inspired distributed computing methods, especially those based on ideas from molecular biology [9]. The main question left open in [14, 15] is regarding the existence of symmetric starvation-free mutual exclusion algorithms for two or more processes when using anonymous registers. In this article, we resolve this open question for memoryless algorithms.

1.2 Mutual exclusion, symmetric algorithms, memoryless algorithms

The mutual exclusion problem. The mutual exclusion problem is to design an algorithm that guarantees mutually exclusive access to a critical section among several competing processes [4]. It is assumed that each process executes a sequence of instructions in an infinite loop. The instructions are divided into four continuous sections: the remainder, entry, critical, and exit. The exit section is required to be wait-free – its execution must always terminate. It is assumed that processes do not fail and that a process always leaves its critical section. The *mutual exclusion problem* is writing the code for the entry and exit sections to satisfy the following two basic requirements.

- *Deadlock-freedom:* If a process tries to enter its critical section, then some process, not necessarily the same one, eventually enters its critical section.
- *Mutual exclusion:* No two processes are in their critical sections simultaneously.

The satisfaction of the above two properties is the minimum required for a mutual exclusion algorithm. For an algorithm to be fair, the satisfaction of the following stronger progress condition is required.

- *Starvation-freedom:* If a process is trying to enter its critical section, then this process eventually enters its critical section.

Symmetric algorithms. A symmetric algorithm is an algorithm in which the processes execute exactly the same code, and the only way to distinguish processes is by comparing identifiers for equality. A process can determine if two identifiers are the same, but nothing else can be determined when they are different. Identifiers can be written, read, and compared, but there is no way of looking inside any identifier. Thus, for example, knowing whether an identifier is odd or even is impossible.

Furthermore, (1) a process can only compare its identifier with another and cannot compare it with a constant value, and (2) the local variables of the different processes have the same names, and (local variables with the same names) are initialized to the same values. Otherwise, it would be possible to distinguish different processes. In symmetric algorithms, as defined above, we say that the processes are symmetric.

As symmetric algorithms do not depend on an order relation between process identities, they require fewer assumptions and are consequently more general than non-symmetric algorithms. The symmetry constraint on process identities can be seen as the "last step" before process anonymity [10, 11]. Symmetric algorithms with non-anonymous memory have been investigated for years [12]. Following [14], we consider a model in which the memory is anonymous, and the processes are symmetric.

Memoryless algorithms. A memoryless mutual exclusion algorithm is an algorithm in which when all the processes are in their remainder section, the values of all the registers (local and shared) are the same as their initial values. This means that a process that tries to enter its critical section does not use any information about its previous attempts (like the fact that it has entered its critical section five times so far). Put another way, in a memoryless algorithm, processes have only a single remainder state and hence cannot retain any memory of prior executions of the algorithm.

All known mutual exclusion algorithms which use anonymous registers and almost all known mutual exclusion algorithms that use non-anonymous registers are memoryless. For example, Lamport's Bakery algorithm is memoryless [8]. Memoryless mutual exclusion algorithms are usually simpler to understand than those that are not memoryless. Furthermore, memoryless algorithms can better handle system-wide failures (i.e., all processes crash simultaneously), as upon recovery, the system can be initialized to the (single) initial state, not affecting processes that had not participated in the algorithm during the failure. When using non-anonymous registers, $2n-1$ registers are necessary and sufficient for designing a symmetric memoryless starvation-free mutual exclusion algorithm for n processes [12].

1.3 Contributions: individual progress vs. global progress

It is known that symmetric memoryless deadlock-free mutual exclusion is solvable for any finite number of processes using anonymous registers [1]. The main question left open in [14, 15] is the existence of symmetric starvation-free mutual exclusion algorithms for two or more processes. We resolve this open question for memoryless algorithms by proving the following possibility and impossibility results,

1. There is a symmetric memoryless starvation-free mutual exclusion algorithm for two processes using $m \geq 7$ anonymous registers if and only if m is odd.
2. There is no symmetric memoryless starvation-free mutual exclusion algorithm for $n \geq 3$ processes using (any number of) anonymous registers.

Our possibility result shows that (1) there is no separation between deadlock-freedom and starvation-freedom for two processes, and (2) there is a separation between deadlock-freedom and starvation-freedom for three or more processes. These results enable us to understand better the intrinsic limits for achieving fairness between asynchronous processes.

Interestingly, as a byproduct of the proof of our impossibility result, we get a general time complexity lower bound for every symmetric *deadlock-free* mutual exclusion algorithm for $n \geq 2$ processes using m anonymous registers. Namely, a process must incur $\lceil m/2 \rceil$ remote memory references (RMR) to enter and exit its critical section once. The RMR complexity of our starvation-free algorithm is $O(m)$.

Two main progress conditions have been studied for asynchronous shared-memory systems with failure-prone processes. The first, wait-freedom, ensures individual progress to each process, i.e., its operations complete as long as it takes an infinite number of steps [6]. The second, lock-freedom, requires only global progress; namely, if a process takes an infinite number of steps, then some (possibly other) processes complete their operations [7]. Wait-freedom corresponds to starvation-freedom in fault-free systems, while lock-freedom corresponds to deadlock-freedom.

It was shown in [5] that there is an object for which there is a lock-free implementation for two processes using only non-anonymous atomic registers, and there is no wait-free algorithm in the same setting. A more general result was presented in [2], showing that such a separation exists (1) also for more than two processes and (2) when primitives stronger than atomic registers are used. Our result, which shows a separation between deadlock-freedom and starvation-freedom for three or more processes, complements these results. The results together show that in shared memory systems where either failures are possible or process symmetry and memory anonymity are assumed, it is not always possible to ensure individual progress in situations where global progress is possible. Thus, achieving various levels of fairness depends on the underlying system assumptions.

2 Preliminaries

Processes. Our model of computation consists of a fully asynchronous collection of n deterministic processes that communicate via m anonymous registers. Asynchrony means that there is no assumption on the relative speeds of the processes. Each process has a unique identifier, which is a positive integer. Since we want to make as few assumptions as possible, it is *not* assumed that the identifiers of the n processes are taken from the set $\{1, ..., n\}$. Thus, a process does not a priori know the identifiers of the other processes. The processes do know the values of n and m. We assume that processes do not fail. As always assumed when solving the mutual exclusion problem, participation is not required – a process may stay in its remainder section and never move to its entry section.

Memory. The shared memory consists of m *anonymous* shared registers. For m anonymous registers, $r_1, ..., r_m$, the adversary can fix, for each process p, a permutation $\pi_p : \{r_1, ..., r_m\} \to \{r_1, ..., r_m\}$ of the registers such that, for process p, the j'th anonymous register is $\pi_p(r_j)$. In particular, when process p accesses its j'th anonymous register, it accesses $\pi_p(r_j)$. Algorithms designed for such a system must be correct regardless of the permutations chosen by the adversary. The permutation fixed for process p is called the naming assignment of p.

All the anonymous registers are assumed to be initialized to the same value. Otherwise, thanks to their different initial values, it would be possible to distinguish different registers, and consequently, the registers would no longer be fully anonymous.

With an *atomic* register, it is assumed that operations on the register occur in some definite order. That is, each operation is an indivisible action. In the sequel, by *registers*, we mean anonymous atomic read/write registers. A read/write register is a shared register that supports (atomic) read and write operations. The fact that anonymous registers do not have global names implies that only multi-writer multi-reader anonymous registers are possible. Such registers can both be written and read by all the processes.

Known results. In [14], it has been proven that a necessary and sufficient condition for the design of a symmetric deadlock-free mutual exclusion algorithm for two processes using anonymous registers is that the number of registers is odd.

▶ **Theorem 1** ([14]). *There is a symmetric deadlock-free mutual exclusion algorithm for two processes using $m \geq 2$ anonymous registers if and only if m is odd.*

We will use this result for two processes later in the paper. The *only if* part of the above theorem is a special case of the following more general result from [14] (Theorem 3.4): There is a symmetric deadlock-free mutual exclusion algorithm for $n \geq 2$ processes using $m \geq 2$ anonymous registers only if for every positive integer $1 < \ell \leq n$, m and ℓ are relatively prime. An optimal symmetric deadlock-free mutual exclusion algorithm using anonymous registers that matches the above general space bound for $n \geq 2$ processes was presented in [1].

3 A starvation-free mutual exclusion algorithm for two processes

We show that, for two processes, it is possible to design a starvation-free mutual exclusion algorithm. In the next section, we prove this is impossible for three or more processes.

▶ **Theorem 2.** *There is a symmetric memoryless starvation-free mutual exclusion algorithm for two processes using $m \geq 7$ anonymous registers if and only if m is odd.*

The *only if* direction follows from Theorem 1 (proven in [14]), where it has been proven that (when using anonymous registers) any symmetric deadlock-free mutual exclusion algorithm for two processes must use an odd number of anonymous registers. To prove the *if* direction, we present in Algorithm 1 a symmetric memoryless starvation-free mutual exclusion algorithm for two processes using m anonymous registers, where m is an odd number greater than or equal to 7. The question of whether a symmetric memoryless starvation-free mutual exclusion algorithm exists for two processes using 3 or 5 anonymous registers is open.

As the m registers do not have global names, each process independently numbers them. We use the notation $p.i[j]$ to denote the j^{th} register according to process i's numbering, for $1 \leq j \leq m$. Recall that a process's identifier is a positive integer.

3.1 An informal description of the algorithm

A shared register is *free* when its value is 0. Initially, all the registers are free. A register is owned by process i, when its value is i. There are two ways for a process to get permission to enter its CS (i.e., Critical Section). The first way is when a process owns $m-2$ registers. Initially, a process tries to own $m-2$ registers by writing its identifier into free registers. If the process succeeds, it may enter its CS, and when done, it releases (i.e., sets to 0) all the registers it owns. By design, a process will never own more than $m-2$ registers.

When there is contention, each process first tries to own as many registers as possible, but no more than $m-2$ registers. Thus, each process will always succeed in owning at least one register (and not at least two registers, as explained in the sequel). After it attempts to own $m-2$ registers, if a process notices that it owns less than $\lceil m/2 \rceil$ registers, it becomes a loser. A loser acts as follows: if it owns more than two registers, it releases all its owned registers except two. Otherwise, when a loser owns one or two registers, it releases no registers. Then, the loser writes "waiting" into the (one or two) registers it owns and waits. Since the waiting process owns at most two registers, the other process, the winner, keeps on trying to own more registers until it eventually succeeds in owning $m-2$ registers and gets permission to enter its CS.

When a winning process exits its CS, the winner releases all the $m-2$ registers it owns, which, as explained below, will prevent it from entering its CS again before a waiting process gets a chance to enter its CS. This guarantees that starvation-freedom is satisfied. To guarantee that one process will not enter its CS twice while the other process is waiting, when a process starts its entry code, it repeatedly scans the registers until none of them has the value "waiting," and only then may it proceed.

The second way a process can enter its CS is by waiting first. A waiting process owns one or two registers with the value "waiting." The waiting process waits until all the registers it does not own are released (i.e., have the value 0). Once this happens, the waiting process may immediately enter its CS. That is, it need not own additional registers. Upon exiting its CS, the (previously waiting) process releases the (one or two) registers it owns.

There is one very delicate possible race condition that should be avoided. Assume there is contention; process i writes its identifier into $m-2$ registers, while process j writes its identifier into two registers. At that point, just before process j writes "waiting" into the two registers it owns, process i enters and exits its CS, releases the $m-2$ registers it owned and then attempts to enter its CS again. Process i reads all the registers, finds out that no process is waiting, and is ready to try to own $m-2$ free registers. Now, process j continues to write "waiting" into the two registers it owns, finds out that all the other $m-2$ registers are free, and enters its CS. Process i is now scheduled, owns the $m-2$ free registers and enters its CS, violating the mutual exclusion requirement.

It is easy to resolve this race condition while satisfying only deadlock-freedom. Resolving it while still satisfying starvation-freedom is more challenging. Our solution is as follows: the first thing that process i is doing (upon entering its entry code) is to own one register and only then check whether j is waiting. This will guarantee that either,

1. process i notices that j is waiting, in which case i releases its owned register and waits until no register has the value "waiting," letting the waiting process enter its CS first; or
2. process i notices that no process is waiting, in which case, after j writes "waiting" into its owned registers; it will find out that not all the other registers are free, and will wait for process i to enter its CS first.

This solution resolves the race condition.

There is one additional thing that needs to be explained. Why do we reserve *two* registers for the waiting process, not just one? The answer is that by reserving two registers, when "waiting" is written into two registers by some process, at *least one* of the two will not be overwritten by the other process. Consider the following scenario. Let r_1, r_2 and r_3 be free registers. Process j writes into r_1,r_2 and is ready to write into r_3. Process i writes into all the $m - 3$ other registers and is ready to write into r_3. Process j writes into r_3, finds out it is a loser (because $m \geq 7$), releases r_1, and writes "waiting" into r_2 and r_3. Now, process i is activated and writes into r_3, leaving only one register with the value "waiting."

Finally, we explain why the algorithm does not work when $m = 5$ (or when $m = 3$). Assume $m = 5$. Let r_1, r_2, r_3, r_4 and r_5 be the five free registers. Consider the following scenario. Process i writes into r_1 and r_2 and is ready to write into r_3. Process j writes into r_5 and r_4 and is ready to write into r_3. Process i writes into r_3, finds out that it owns $m - 2$ registers, and enters its CS. Then, Process j writes into r_3, finds out that it owns $m - 2$ registers, and enters its CS, violating the mutual exclusion requirement. We point out that, in contrast, there is a symmetric memoryless deadlock-free mutual exclusion algorithm for two processes using $m \geq 3$ anonymous registers when m is odd.

3.2 Correctness Proof

▶ **Lemma 3.** *The algorithm satisfies mutual exclusion.*

Proof. We assume to the contrary that both processes enter their CS simultaneously and show that this leads to a contradiction. There are two ways for a process to enter its CS: (1) by observing that it owns $m - 2$ registers, and (2) by writing "waiting" into the registers it owns and then waiting until all the other registers have the value 0. So, four possible combinations exist for having two processes in their CS simultaneously. We show that none of them may happen. Let's call the two processes p and q.

1. *Both processes observe that they own $m - 2$ registers and enter their CS.* Once p observes that it owns $m - 2$ registers and enters its CS, at most one of these $m - 2$ registers may later be overwritten by q. Thus, while p is in its CS, q may own at most 3 registers. Since $m \geq 7$, q cannot observe that it owns $m - 2$ registers and hence will not enter its CS, while p is in its CS – a contradiction.
2. *Both processes write "waiting" into the registers they own and later enter their CS.* This may happen only if p and q observe that each one of them owns less than half of the registers. However, if p observes that it owns less than half of the registers, it must be the case that q owns more than half of the registers and will not write "waiting" into the registers it owns – a contradiction.

Algorithm 1 A symmetric memoryless starvation-free mutual exclusion algorithm for two processes.

```
CODE OF PROCESS $i$                                                    // $i \neq 0$

Constant:
    $m$: an odd integer $\geq 7$                   // $m \geq 7$ is the # of shared registers
Shared variables:
    $p.i[1..m]$: array of $m$ anonymous registers, of type integer + the symbol "waiting," init. all 0
                          // $p.i[j]$ is the $j^{th}$ register according to process $i$'s numbering
Local variables:
    $myview[1..m]$: array of $m$ variables, initially all 0
    $mycounter, j, k$: integer, initially 0
    $mygo$: boolean, initially $false$

                                                     //give priority to a waiting process
repeat $mycounter \leftarrow mycounter + 1$ until $p.i[mycounter] = 0$   //looking for a zero entry
$p.i[mycounter] \leftarrow i$                                                 //own one register
for $j = 1$ to $m$ do $myview[j] \leftarrow p.i[j]$ end for                       //read the shared array
if $\exists j \in \{1, ..., m\} : myview[j] = waiting$ then                   //is other process waiting?
    if $p.i[mycounter] = i$ then $p.i[mycounter] \leftarrow 0$ end if            //release owned register
    repeat                                                     //the other process is waiting
        for $j = 1$ to $m$ do $myview[j] \leftarrow p.i[j]$ end for                 //read the shared array
    until $\forall j \in \{1, ..., m\} : myview[j] \neq waiting$                //wait for CS to be released
end if

repeat                                                           //try to own $m - 2$ registers
    for $k = 1$ to $m$ do                                               //access the $m$ registers
        if $p.i[j] = 0$ then                                              //try to own one more
            for $j = 1$ to $m$ do $myview[j] \leftarrow p.i[j]$ end for             //read the shared array
            if $i$ appears in less than $m - 2$ of the entries of $myview[1..m]$ then
            $p.i[j] \leftarrow i$ end if end if                                           //own one more
    end for
                                                                               //lose or win?
    for $j = 1$ to $m$ do $myview[j] \leftarrow p.i[j]$ end for                     //read the shared array
    if $i$ appears in less than $\lceil m/2 \rceil$ of the entries of $myview[1..m]$ then          //lose
        $mycounter \leftarrow 0$
        for $j = 1$ to $m$ do if $p.i[j] = i$ then                  //release all owned registers
            if $mycounter = 2$ then $p.i[j] \leftarrow 0$                              //except two of them
                else $p.i[j] \leftarrow waiting$; $mycounter \leftarrow mycounter + 1$ end if     //signal waiting
        end for
        repeat                                                      //wait for CS to be released
            for $j = 1$ to $m$ do $myview[j] \leftarrow p.i[j]$ od                  //read the shared array
        until $\forall j \in \{1, ..., m\} : myview[j] \in \{0, waiting\}$        //no sign from other process
        $mygo \leftarrow true$                                                       //may enter CS
    end if
until $i$ appears in $m - 2$ of the entries of $myview[1..m]$ or $mygo = true$
critical section
                                                     //release all owned shared registers
if $mygo = true$ then for $j = 1$ to $m$ do if $p.i[j] = waiting$ then $p.i[j] \leftarrow 0$ end if end for
    else for $j = 1$ to $m$ do if $p.i[j] = i$ then $p.i[j] \leftarrow 0$ end if end for
end if
set all local variables to their initial values
```

3. *p observes that it owns $m-2$ registers and enters its CS, while q writes "waiting" into the registers it owns and* later *enters its CS while p is still in its CS.* Once p enters its CS, all the registers it owns are not free. So, q will not be able to proceed since not all the registers it does not own are free as required – a contradiction.
4. *p writes "waiting" into the registers it owns, waits until all the other registers have the value 0 and enters its CS. q observe that it owns $m-2$ registers and enters its CS, while p is still in its CS.* Since, after writing "waiting," p must observe that all the registers it does not own have the value 0 before it may enter its CS; it must be the case that q has written into one of the registers (line 2), after p has written "waiting." Thus, q will notice (line 4) that p is waiting and will not proceed to its CS – a contradiction. ◀

▶ **Lemma 4.** *The algorithm satisfies starvation-freedom.*

Proof. There are three loops where a process can get stuck.

- **Loop 1:** The repeat loop at lines 6-8, where a process waits until none of the registers is set to waiting.
- **Loop 2:** The inner repeat loop at lines 24-26, where a (waiting) process waits until all the registers are set to 0 or waiting.
- **Loop 3:** The outer repeat loop at lines 10-29 where a process waits until it either owns $m-2$ registers or its local register *mygo* is set to *true*.

We show that a process cannot get stuck (i.e., loop forever) in any of these loops, which implies starvation-freedom. Let's call the processes p and q.

Assume p is waiting in loop 1, and cannot proceed. This means that (1) p is not owning any of the registers, and (2) that q has set at least one of its owned registers to "waiting" (line 22). By the time q reaches loop 3 (line 24), all its owned registers are set "waiting," and all the other registers are free. Thus, q will exit the loop, set *mygo* to *true* and enter its CS. Later, in its exit code q will release its owned registers and return to its remainder region. From that point on (even if q will try to enter its CS again), as long as p is waiting in loop 1, no register will be set to waiting. This means that the condition in line 8 is evaluated to *true* and thus p can proceed beyond loop 1.

Assume p is waiting in loop 2, and cannot proceed. This means that (1) p has set the (one or two) registers it owns to "waiting" and is not trying to own more registers, and (2) q owns at least one register. There are two cases to consider:

- q notices (in line 4) that p is waiting, in which case q releases its owned register and waits until no register has the value "waiting," letting the waiting process p proceed beyond loop 2, setting *mygo* to *true* and proceed beyond loop 3.
- q notices that no process is waiting, in which case, after p writes "waiting" into its owned registers and waits in loop 2, there is nothing that prevents q from owning $m-2$ registers and entering its CS. Later, in its exit code q will release its owned registers and return to its remainder region. If q does not try to enter its CS again, then p can proceed beyond loop 2 and loop 3. If q does try to enter its CS again, then q might acquire (line 3) one register before p notices that the register is free; however, q will later notice (in line 4) that p is waiting, in which case q will release its owned register and waits until no register has the value "waiting," letting the p proceed beyond loop 2 and loop 3.

Assume p is looping in loop 3, never waits in loop 2, and cannot proceed. Since m is odd, this means that p owns more than half of the registers, while q holds less than half of the registers. Thus, q will eventually release the registers it owns, except at most two of them. This will enable p to acquire $m-2$ registers and proceed beyond loop 3. ◀

RMR complexity. An operation that a process performs on a memory location is considered a remote memory reference (RMR) if the process cannot perform the operation locally on its cache and must transact over the multiprocessor's interconnection network to complete the operation. RMRs are undesirable because they take long to execute and increase the interconnection traffic. Our algorithm achieves the ideal RMR complexity of $O(m)$ for cache coherent machines. (Distributed Shared Memory machines are irrelevant for anonymous shared memory systems.) This means that a process incurs $O(m)$ number of RMRs to satisfy a request (i.e, to enter and exit the critical section once). It follows from Observation 12 (Section 4) that this bound is tight.

4 An impossibility result

In the previous section, we have shown that, for two processes, it is possible to design a symmetric memoryless starvation-free mutual exclusion algorithm. Next, we show this is impossible for three or more processes.

▶ **Theorem 5.** *There is no symmetric memoryless starvation-free mutual exclusion algorithm for $n \geq 3$ processes using (any number of) anonymous registers.*

The main argument is that, under the appropriate assignment of names to registers, there is a way to run two processes when they are in their remainder section, requiring them to write to all registers before one of them can enter its critical section. This is essentially accomplished by renaming unwritten registers on the fly so that if the two processes are about to write to the same unwritten register for the first time, then they end up writing to two distinct, unwritten ones. Thus, it is possible to hide all the write operations of a third process, which will prevent it from ever entering its critical section.

4.1 Basic definitions and observations

We first prove some basic (but general) observations regarding the mutual exclusion problem. All the lemmas and definitions in Subsection 4.1 refer to one arbitrary deadlock-free mutual exclusion algorithm for n processes using read/write registers. Here, we do not need to assume that the algorithm is starvation-free, symmetric, or memoryless, nor do we need to assume that the registers are anonymous.

We will use the following notions and notations. An *event* corresponds to an atomic step performed by a process. An algorithm's (global) state is entirely described by the values of the (local and shared) registers and the values of the program counters of all the processes. A *run* is a sequence of alternating states and events (also called steps). For the purpose of the impossibility proof, it is more convenient to define a run as a sequence of events omitting all the states except the initial state. Since the events and the initial state uniquely determine the states in a run, no information is lost by omitting the states.

Each event in a run is associated with a specific process that is *involved* in the event. We will use x, y, and z to denote runs. When x is a prefix of y, we denote by $(y - x)$ the suffix of y obtained by removing x from y. Also, we denote by $x; seq$ the sequence obtained by extending x with the sequence of events *seq*. Processes are *deterministic*; that is, for every two runs $x; e$ and $x; e'$ if e and e' are events by the same process, then $e = e'$.

We will often use statements like "in run x process p is in its remainder", and implicitly assume that there is a function that for any run and process, lets us know whether a process is in its remainder, entry code, critical section, or exit code. Also, saying that an extension y of x involves only process p, means that all events in $(y - x)$ involve only process p. Finally, by a run we always mean a finite run, by a register we mean a shared register, and by the value of register r in run x, we always mean, the value of r at the end of x. Our first definition captures when two runs are indistinguishable to a given process.

▶ **Definition.** *Run x **looks like** run y to process p, if the subsequence of all events by p in x is the same as in y, and the values of all the registers in x are the same as in y.*

The looks like relation is an equivalence relation.[1] The next step by a given process always depends on the process's previous steps and the registers' current values. The previous steps uniquely determine whether the next step is a read or a write. The current values of the registers determine what value will be read in case of a read step. If two runs look alike to process p, then the next step by p in both runs is the same.

▶ **Lemma 6.** *Let x be a run which looks like run y to every process in a set P. If z is an extension of x which involves only processes in P, then $y; (z - x)$ is a run.*

Proof. By a simple induction on k – the number of events in $(z - x)$. The basis when $k = 0$ holds trivially. We assume the lemma holds for $k \geq 0$ and prove for $k + 1$. Assume that the number of events in $(z - x)$ is $k + 1$. For some event e, it is the case that $z = z'; e$. Since the number of events in $(z' - x)$ is k, by the induction hypothesis $y' = y; (z' - x)$ is a run. Let $p \in P$ be the process which is involved in e. Then, from the construction, the runs z' and y' look alike to p, which implies that the next step by p in both runs is the same. Thus, since $z = z'; e$ is a run, also $y'; e = y; (z - x)$ is a run. ◀

We next define the notion of a hidden process.[2] Intuitively, a process is hidden in a given run, if all the steps it has taken since the last time it has been in its remainder, communicate no information to the other processes. We say that a write event e_1 is *overwritten* by event e_2 in a given run r if e_2 is a write event that happens after e_1 in r, and both e_1 and e_2 are writing events to the same register.

▶ **Definition.** *For process p and run z, let z' be the longest prefix of z such that p is in its remainder in z'. Process p is **hidden** in run z if each event which p is involved in $(z - z')$ is either: a read event, or a write event that is overwritten (in z) before any other process has read the value written.*

We notice that a process is not hidden if it is involved in a write event that is not later overwritten, even if the write does not change the current value of a register. Also, if a process is in its remainder in z then it is hidden in z, and thus initially, all the processes are hidden. A hidden process looks just like a process halted in its remainder, and hence no process can wait until a hidden process takes a step.

▶ **Lemma 7.** *If a process p is in its critical section in run z then p is not hidden in z.*

Proof. Assume to the contrary that process p is hidden and is in its critical section in run z. Let z' be the longest prefix of z such that p is in its remainder in z'. Since p is *hidden* in run z, it is possible to remove from z all the events in which p is involved in $(z - z')$ and get a new run y. The run y looks like z to all processes other than p, and p is in its remainder in y. By the deadlock-freedom property, there is an extension of y that does not involve p in which some process $q \neq p$ enters its critical section. Since y looks like z to all processes

[1] The term *looks like*, adopted from [13], is also called *indistinguishable* in the literature.
[2] The notion of a hidden process was first defined in [3].

other than p, by Lemma 6, a similar extension exists starting from z. That is, q can enter its critical section in an extension of z, while p is still in its critical section. However, this violates the mutual exclusion property. ◀

It follows from Lemma 7 that a process must write before it enters its critical section.

4.2 Anonymity

The lemma in Subsection 4.2 refers to one arbitrary deadlock-free mutual exclusion algorithm for n processes using *anonymous* registers. Here, we do not need to assume that the algorithm is starvation-free, symmetric, or memoryless. We denote by e_p a (read or write) event which involves process p. When x is a run and π_p is a naming assignment of p, we denote by

- $x[p, r_i \leftrightarrow r_j]$ the sequence obtained by replacing every read event of register r_i by process p in x with a read event of register r_j by p which returns the same value (as the event of reading r_i), and vice versa.
- $\pi_p[r_i \leftrightarrow r_j]$ is the naming assignment where, (1) $\pi_p[r_i \leftrightarrow r_j](r_i) = \pi_p(r_j)$; (2) $\pi_p[r_i \leftrightarrow r_j](r_j) = \pi_p(r_i)$; and (3) for every $k \notin \{i, j\}$, $\pi_p[r_i \leftrightarrow r_j](r_k) = \pi_p(r_k)$.

Recall that all the anonymous registers are initialized to the same value.

▶ **Lemma 8.** *Let x be a run, p a process, π_p^x the naming assignment of p (used in the run x), and r_i and r_j two registers that were never written (by any process) in x. Then, $y = x[p, r_i \leftrightarrow r_j]$ is a run, where the naming assignment for p (used in y) is $\pi_p^x[r_i \leftrightarrow r_j]$, and for the other processes the naming assignments are the same as in x. Furthermore, if $x; e_p$ is a run where e_p is an event of writing the value v into r_i then $y; e'_p$ where e'_p is an event of writing the value v into r_j is also a run.*

Proof. Since r_i and r_j were never written in x, a read event by p or any other process of each of those registers in x would return the initial value. So, swapping the read events of r_i and r_j by process p, may only affect process p.

As for p, before its first event in x, the adversary has fixed, for process p, a naming assignment $\pi_p^x : \{r_1, ..., r_m\} \to \{r_1, ..., r_m\}$ of the registers such that, for process p, the j'th anonymous register is $\pi_p^x(r_j)$. In particular, when process p accesses its j'th anonymous register, it accesses $\pi_p^x(r_j)$. Let us define the permutation $\pi_p^y : \{r_1, ..., r_m\} \to \{r_1, ..., r_m\}$ as follows, (1) $\pi_p^y(r_i) = \pi_p^x(r_j)$; (2) $\pi_p^y(r_j) = \pi_p^x(r_i)$; and (3) for every k where $k \notin \{i, j\}$, $\pi_p^y(r_k) = \pi_p^x(r_k)$. That is, whenever p accessed r_i before it will now access r_j and vice versa. Now, consider a run in which the processes are scheduled exactly in the order as they are scheduled in x, the naming assignment fixed (by the adversary) for p is π_p^y, and the naming assignments (in y) of the other processes are as in x. By construction, the resulting run is run y. Furthermore, if the next event by p in x is of writing the value v into r_i then, since $\pi_p^y(r_i) = \pi_p^x(r_j)$, the next event by p in y is of writing the value v into r_j. ◀

4.3 The notions of a symmetric run and a symmetric state

A (global) state is entirely described by the values of the local and shared registers and the values of the program counters of all the processes.

Intuitively, a state σ is symmetric w.r.t. two processes if the "subjective views" of the processes at σ are the same. In the standard non-anonymous model, once two (symmetric) processes write their ids (one after the other) into the same register, say r_1, their views after these two writes are completed are no longer the same, if they inspect the current state, one process will see that the value of r_1 equals its id, while the other will see that the value is different from its id. In the anonymous model, it is possible for each process to see that the value of the first register according to its naming assignment equals its id.

Below we define this notion more formally. Let m be the number of registers; let $val_\sigma(r)$ be the value of register r in state σ, and assume that the names of the local variables of the processes are the same. To distinguish the local variables of the different (symmetric) processes, we will add the process id as a subscript to the variable names.

▶ **Definition.** *Let σ be a state and let π_p and π_q be the naming assignments of p and q, respectively. State σ is* ***symmetric*** *w.r.t. p and q and their naming assignments π_p and π_q, if for every $1 \le k \le m$ either,*

- $val_\sigma(\pi_p(r_k)) = val_\sigma(\pi_q(r_k))$ *and* $val_\sigma(\pi_p(r_k)) \notin \{p, q\}$, *or*
- $val_\sigma(\pi_p(r_k)) = p$ *and* $val_\sigma(\pi_q(r_k)) = q$, *or*
- $val_\sigma(\pi_p(r_k)) = q$ *and* $val_\sigma(\pi_q(r_k)) = p$.

Furthermore, in σ, for every local variable, say local, either (1) $local_p = local_q$ and the value of $local_p$ is not in $\{p, q\}$, or (2) $local_p = p$ and $local_q = q$, or (3) $local_p = q$ and $local_q = p$.

▶ **Definition.** *Run x is* ***symmetric*** *w.r.t. p and q and their naming assignments π_p and π_q, if the state at the end of x is symmetric w.r.t. p and q and their naming assignments π_p and π_q.*

The following lemma follows immediately from the definitions of a symmetric run and a symmetric state.

▶ **Lemma 9.** *When processes p and q are symmetric (and deterministic), if run x is symmetric w.r.t. p and q and their naming assignments π_p and π_q, then*

1. *either the next step of both processes is a read or the next step of both is a write;*
2. *if p accesses $\pi_p(r_k)$ in its next step, then q accesses $\pi_q(r_k)$ in its next steps;*
3. *if x is extended by a read event of p followed by a read event of q, then the resulting run is symmetric w.r.t. p and q and their naming assignments;*
4. *if x is extended by a write event of p followed by a write event of q then the resulting run is symmetric w.r.t. p and q and their naming assignments, provided that p and q do not write into the same physical location; and*
5. *p and q cannot be in their critical sections at (the end of) x.*

▶ **Lemma 10.** *In a symmetric deadlock-free mutual exclusion algorithm for $n \ge 2$ processes using anonymous registers, the initial state where all the processes are in their remainder sections is symmetric w.r.t. every two processes and their naming assignments.*

Proof. The proof follows immediately from the fact that in anonymous shared memory, all the anonymous registers are initialized to the same value; and that when the processes are symmetric, the local variables of the different processes have the same names and (local variables with the same names) are initialized to the same values. ◀

To simplify the presentation, for the rest of the section, by a *symmetric run* (resp. *symmetric state*), we always mean a symmetric run (resp. symmetric state) w.r.t. to every two processes and their naming assignments.

4.4 Symmetry and anonymity

The following lemma in Subsection 4.4 refers to one arbitrary symmetric deadlock-free mutual exclusion algorithm for $n \ge 2$ processes using m *anonymous* registers. Here, there is no need to assume that the algorithm is starvation-free or memoryless.

A *quiescent* state is one in which all the processes are in their remainder sections. A memoryless algorithm is an algorithm that has exactly one quiescent state (which is the initial state). When the possible number of quiescent states of a symmetric starvation-free mutual exclusion algorithm is more than one, by Lemma 10, the initial (quiescent) state is symmetric w.r.t. every two processes and their naming assignments. However, for a non-memoryless algorithm, except for the initial state, the other quiescent states are not necessarily symmetric.

▶ **Lemma 11** (main technical lemma). *For every two processes p and q, and every symmetric quiescent state σ, there exist naming assignments π_p and π_q for p and q, and a run ρ with the following properties,*

1. *ρ starts from the state σ and ends in some quiescent state,*
2. *during ρ, only p and q take steps, and they enter and exit their critical sections once,*
3. *during ρ, each one of the two processes writes into $\lceil m/2 \rceil$ different registers, and*
4. *during ρ, each one of the m anonymous registers is written at least once.*

Proof. It follows from Theorem 1 that (when using anonymous registers) any symmetric deadlock-free mutual exclusion algorithm for two processes must use an odd number of anonymous registers. So, for the rest of this proof, we assume that m is odd.

We prove the lemma by running the two processes p and q, starting from the (symmetric) state σ, keeping the run symmetric (i.e., without breaking symmetry). As long as symmetry is not broken, by Lemma 9, none of the processes can enter its CS, because if it does, then the other process may enter its CS as well, violating the mutual exclusion requirement. Each time two more registers are written until only one is left. At that point, the two processes try to write this last register. Only when this last register is written is symmetry broken, all the m anonymous registers are written, and we are done.

Initially, the adversary fixes for process p, the identity permutation $\pi_p(r_k) = r_k$ (where $1 \leq k \leq m$), and fix for q its reverse permutation $\pi_q(r_k) = r_{m-k+1}$. For the sake of explanation, we arrange the registers in pairs, where the k^{th} pair is $(\pi_p(r_k), \pi_q(r_k))$, where $1 \leq k \leq m$. In this pairing, each pair includes two different registers except the $\lceil m/2 \rceil$ pair in which the register $r_{\lceil m/2 \rceil}$ appears twice. Each other register appears in two pairs, and the other register is the same in those two pairs. For example, when $m = 5$, the pairs are (r_1, r_5), (r_2, r_4), (r_3, r_3),(r_4, r_2), and(r_5, r_1).

Next, we run the processes in *lock-steps*. Each lock-step includes a step by p followed by a step of q. We observe that, as long as the constructed run is symmetric, in one lock-step, if p accesses r_i and q accesses r_j, then (1) the pair (r_i, r_j) appears in the pairing described above, and (2) by Lemma 9, both processes either read r_i and r_j or write into them. Because of the initial symmetry, none of the processes may enter its CS without writing first. (this also follows from Lemma 7).

The run ρ is constructed by iteratively executing the following procedure until all the m registers are written at least once, as required. Let π_p and π_q denote the current naming assignments of p and q. Let us denote by x_{i-1}, the run constructed so far, before the beginning of the i^{th} iteration. As we will see, by construction, x_i will be a symmetric run for all $i \geq 0$. By assumption, the quiescent state σ is symmetric, and thus, the run x_0, where all the processes are still in their remainder sections, is symmetric.[3]

Iteration $i \geq 1$ begins here. As explained, x_{i-1} is the symmetric run constructed so far. First, p and q run (in lock-steps) until they are ready to write into r_i and r_j, respectively. Recall that by Lemma 9, either both steps are read or both are write, in each lock-step.

[3] We notice that in the special case where the algorithm is memoryless, there is no need to assume that σ is symmetric, as by Lemma 10, the single initial (quiescent) state σ is symmetric.

Thus, to break symmetry they must eventually try to write. Let's denote by y_{i-1} the run just before the two writes. Clearly, if x_{i-1} is symmetric, then so is y_{i-1}. There are four possible cases,

1. r_i and r_j are different registers. In this simple case, we let both processes complete their write operations. The run x_i is the run just after these two writes. If x_{i-1} is symmetric, then so is x_i. The i^{th} iteration completes here, and we are ready to start the next $i+1$ iteration.
2. r_i and r_j refer to the same register, which has been written before. The construction ensures this situation never happens (see Case 4 below).
3. r_i and r_j refer to the same register which has not been written before, and all other $m-1$ registers have already been written. In this case, we let both processes complete their write operations. At this point, all the m registers have been written as requested. So we let the processes continue running until, as guaranteed by the deadlock-freedom property, each one of them will eventually enter its CS (but not simultaneously), return to its remainder section, and we are done. The constructed run is ρ, and **the construction of the run ρ terminates here.**
4. r_i and r_j refer to the same register which has not been written before, and *not* all other $m-1$ registers have already been written. This is the more challenging case. For some k, let $\pi_p(r_k)$ and $\pi_q(r_k)$ be two registers (different from r_i) that have not been written so far (we do not care whether $\pi_p(r_k)$ and $\pi_q(r_k)$ have or have not been read so far). By Lemma 8, $z^1_{i-1} = y_{i-1}[p, r_i \leftrightarrow \pi_p(r_k)]$ is a legal run, assuming π_p is replaced with $\pi_p[r_i \leftrightarrow \pi_p(r_k)]$. Furthermore, by applying lemma 8 again, $z^2_{i-1} = z^1_{i-1}[q, r_j \leftrightarrow \pi_q(r_k)]$ is a legal run, assuming π_q is replaced with $\pi_q[r_i \leftrightarrow \pi_q(r_k)]$. Since r_i, $\pi_p(r_k)$ and $\pi_q(r_k)$ have never been written yet, if x_{i-1} is symmetric, so is z^2_{i-1}.
   Furthermore, at (the end of) z_2, p and q are ready to write into the two registers $\pi_p(r_k)$ and $\pi_q(r_k)$, respectively. We now continue with run z^2_{i-1}, and let both processes complete their write operations into $\pi_p(r_k)$ and $\pi_q(r_k)$. Again symmetry is preserved. The resulting run, after the writes is x_i with which we continue to the next iteration. It is important to notice that in this case (unlike in the other two cases) x_i is *not* an extension of x_{i-1}, it is a completely new run. In addition, in the next round, which starts with x_i, we use the (updated) naming assignments $\pi_p[r_i \leftrightarrow \pi_p(r_k)]$ and $\pi_q[r_j \leftrightarrow \pi_q(r_k)]$ for processes p and q. As explained x_i is symmetric w.r.t. p and q and their updated naming assignments.

In case 4, we switch to a new run and new naming assignments. We emphasize that, during a specific run, we do not change the naming assignments associated with that run. We change them only when we switch to a completely new run. The run ρ that we end up with at the end of the construction uses fixed naming assignments for p and q from the beginning to the end. This whole construction can be viewed as searching for a run ρ and fixed naming assignments π_p and π_q for which *Case 4* will never happen, and this is exactly what we end up with. That is, once run ρ is constructed with its associated naming assignments for p and q, if we consider execution of run ρ starting from σ with the associated naming assignments, then, by construction, the only time p and q will try to write into the same register, would be *after* all other $m-1$ registers have already been written. ◀

▶ **Observation 12.** *The RMR complexity of every symmetric deadlock-free mutual exclusion algorithm for $n \geq 2$ processes using m anonymous registers is at least $\lceil m/2 \rceil$.*

Proof. Let σ be the initial state. By Lemma 10, the initial state σ is symmetric. By Lemma 11, there exists a run which starts from σ, in which exactly two processes participate during which they enter and exit their critical sections once, and each one of the two processes writes into $\lceil m/2 \rceil$ different registers. The result follows. ◀

4.5 Proof of Theorem 5

Proof. Assume to the contrary that there is a symmetric memoryless *starvation-free* mutual exclusion algorithm for $n \geq 3$ processes using anonymous registers. Let's call this algorithm A. Let us denote by σ the single possible initial state. By Lemma 10, σ is symmetric. Let p_1, p_2 and q be three processes. Using Lemma 11, we will reach a contradiction by hiding all the write operations of q, which, by Lemma 7, will prevent q from ever entering its critical section.

This is done as follows. Assume all the processes are in their remainder sections; thus, the current state is σ. Now, process q tries to enter its critical section. Before doing so, q should execute its entry section which must, by Lemma 7, involve at least one write operation. So, we run q alone until it is about to execute its first write operation. Let's call the first register that q is about to write register r_1. Since q has not modified any register yet, all the processes except q cannot distinguish the current state from the state σ where all the processes (including q) are still in their remainder sections. Thus, by Lemma 6 and Lemma 11, there is an extension of the current run which involves only p_1 and p_2, in which p_1 and p_2 enter and then exit their critical sections once, and in that extension, each one of the anonymous registers is written (by either p_1 or p_2) at least once. We slightly modify this extension of p_1 and p_2 by stopping them just before writing into register r_1; let q complete a write operation into r_1; then let p_1 and p_2 overwrite the value written by q, and continue until p_1 and p_2 return to their remainder sections.

Since the write of q into r_1 was immediately overwritten, q is hidden. Hence all the processes except q cannot again distinguish the current state from the state σ where all the processes (including q) are still in their remainder sections. Notice that here we use the assumption that the algorithm is memoryless, and this will enable us to use Lemma 11 repeatedly. It is important to understand that since we have already used Lemma 11 once in the proof, the naming assignments for processes p_1 and p_2 have been fixed, and these assignments can not be changed after that. However, because of the memoryless assumption, this does not prevent us from applying Lemma 11 again to σ using processes p_1 and p_2.

We notice that *without* the memoryless assumption, after p_1 and p_2 return to their remainder sections, it is no longer necessarily true that all the processes except q cannot again distinguish the current state from the state σ. This is so because there might be another state σ' where all the processes are in their remainder sections. Hence, in such a case (without the memoryless assumption), it would not be possible to apply Lemma 11 again.

By Lemma 7, at this point, since q is hidden, q must write again before it may enter its critical section. So, we run q alone until it is about to execute its second write operation. Let's call the register that q is about to write register r_2. (Register $r2$ might denote a different or the same register as r_1.) As before, all the processes except q cannot distinguish the current state from the state σ. Thus, again by Lemma 11, there is an extension of the current run that involves only process p_1 and p_2, in which p_1 and p_2 enter and then exit their critical sections once, and in that run each one of the anonymous registers is written at least once. We slightly modify this extension of p_1 and p_2 by stopping them just before writing register r_2; let q complete a write operation into r_2; then let p_1 and p_2 overwrite the value written by q, and continue until they return to their remainder sections. Since the write of q into r_2 was also immediately overwritten, q is still hidden, and thus all the processes except q cannot distinguish the current state from σ.

We can apply the above procedure as often as necessary, hiding all the write operations of q, which by Lemma 7, prevents q from ever entering its critical section. Thus, algorithm A does not satisfy starvation-freedom as promised. A contradiction. ◀

4.6 A generalization

For a non-memoryless algorithm, except for the initial state, the other quiescent states are not necessarily symmetric. Under the assumption that all the quiescent states are symmetric, it is possible to prove the following generalization of Theorem 5.

▶ **Theorem 13.** *There is no symmetric starvation-free mutual exclusion algorithm, with at most s quiescent states, for $n \geq 2s+1$ processes using (any number of) anonymous registers, assuming all the quiescent states are symmetric.*

Proof. We slightly modify the proof of the impossibility result in Subsection 4.5. We assume to the contrary that there is a symmetric *starvation-free* mutual exclusion algorithm for $n \geq 2s+1$ processes using anonymous registers. With each quiescent state σ_i we associate two processes p_1^i and p_2^i. Let q be a process not associated with any quiescent state. Using the above-modified version of Lemma 11, we can reach a contradiction by hiding all the write operations of q, which, by Lemma 7, will prevent q from ever entering its critical section.

This is done as follows. Assume that all the processes are in their remainder sections, and the current state is σ_i which is assumed to be symmetric (this is required for applying the modified version of Lemma 11). Now, process q tries to enter its critical section. Before doing so, q must execute at least one write operation. So, we run q alone until it is about to execute a write operation. Let's call the register that q is about to write, register r. Since q has not modified r yet, all the processes except q cannot distinguish the current state from the quiescent state σ_i. Thus, there is an extension of the current run, which involves only p_1^i and p_2^i, in which p_1^i and p_2^i enter and then exit their critical sections once, and in that extension, each one of the anonymous registers is written at least once. We slightly modify this extension by stopping p_1^i and p_2^i just before writing into register r; let q complete a write operation into r; then let p_1^i and p_2^i overwrite the value written by q, and continue until p_1^i and p_2^i return to their remainder sections.

Since the write of q into r was immediately overwritten, q is still hidden, and thus all the processes except q cannot distinguish the current state from one of the s quiescent states. We can apply the above procedure as often as necessary, hiding all the write operations of q, which by Lemma 7, prevents q from ever entering its critical section. Thus, the algorithm does not satisfy starvation-freedom as promised – a contradiction. ◀

5 Discussion

We have shown that, while for two processes, it is possible to design a symmetric memoryless starvation-free mutual exclusion algorithm using anonymous registers, this is impossible for three or more processes. These results imply that, while there is no separation between deadlock-freedom and starvation-freedom for two processes, such a separation between deadlock-freedom and starvation-freedom exists for three or more processes. Thus, in anonymous shared memory systems where process symmetry is assumed, it is impossible always to ensure individual progress in situations where global progress is possible. This is the first known case of fault-free systems, demonstrating a separation between individual and global progress.

References

1 Z. Aghazadeh, D. Imbs, M. Raynal, G. Taubenfeld, and Ph. Woelfel. Optimal memory-anonymous symmetric deadlock-free mutual exclusion. In *Proceedings of the 38th ACM Symposium on Principles of Distributed Computing*, PODC '19, pages 157–166, 2019.

2 H. Attiya, A. Castañeda, D. Hendler, and M. Perrin. Separating lock-freedom from wait-freedom at every level of the consensus hierarchy. *Journal of Parallel and Distributed Computing*, 163:181–197, 2022. Conf. version appeared in PODC 2018.

3 J. N. Burns and N. A. Lynch. Bounds on shared-memory for mutual exclusion. *Information and Computation*, 107(2):171–184, December 1993.

4 E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

5 M. Herlihy. Impossibility results for asynchronous PRAM. In *Proc. of the 3rd Annual ACM Symp. on Parallel Algorithms and Architectures*, pages 327–336, 1991.

6 M. P. Herlihy. Wait-free synchronization. *ACM Trans. on Programming Languages and Systems*, 13(1):124–149, January 1991.

7 M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, 1990.

8 L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.

9 S. Rashid, G. Taubenfeld, and Z. Bar-Joseph. The epigenetic consensus problem. In *28nd International Colloquium on Structural Information and Communication Complexity (SIROCCO 2021)*, pages 146–163, June 2021. *LNCS 12810.*

10 M. Raynal and G. Taubenfeld. Symmetry and anonymity in shared memory concurrent systems. *Bulletin of the European Association of Theoretical Computer Science (EATCS)*, 136, 2022. 17 pages.

11 M. Raynal and G. Taubenfeld. A visit to mutual exclusion in seven dates. *Theoretical Computer Science*, 919:47–65, June 2022.

12 E. Styer and G. L. Peterson. Tight bounds for shared memory symmetric mutual exclusion problems. In *Proc. 8th ACM Symp. on Principles of Distributed Computing*, pages 177–191, August 1989.

13 G. Taubenfeld. *Synchronization Algorithms and Concurrent Programming.* Pearson / Prentice-Hall, 2006. ISBN 0-131-97259-6, 423 pages.

14 G. Taubenfeld. Coordination without prior agreement. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC '17, pages 325–334, 2017.

15 G. Taubenfeld. Anonymous shared memory. *J. ACM*, 69(4):1?30, August 2022.

Improved and Partially-Tight Lower Bounds for Message-Passing Implementations of Multiplicity Queues

Anh Tran
Bucknell University, Lewisburg, PA, USA

Edward Talmage
Bucknell University, Lewisburg, PA, USA

Abstract

A *multiplicity queue* is a concurrently-defined data type which relaxes the conditions of a linearizable FIFO queue to allow concurrent *Dequeue* instances to return the same value. It would seem that this should allow faster message-passing implementations, as processes should not need to wait as long to learn about concurrent operations at remote processes and previous work has shown that multiplicity queues are computationally less complex than the unrelaxed version. Intriguingly, other work has shown that there is, in fact, not much speedup possible versus an unrelaxed queue implementation. Seeking to understand this difference between intuition and real behavior, we improve the existing lower bound for uniform algorithms. We also give an upper bound for a special case to show that our bound is tight at that point. To achieve our lower bounds, we use extended shifting arguments, which are rarely used. We use these techniques in series of inductive indistinguishability proofs, extending our proofs beyond the usual limitations of traditional shifting arguments. This proof structure is an interesting contribution independently of the main result, as new lower bound proof techniques may have many uses in future work.

2012 ACM Subject Classification Software and its engineering → Abstract data types; Theory of computation → Distributed algorithms; Computing methodologies → Distributed algorithms

Keywords and phrases Distributed Data Structures, ADTs, Lower Bounds, Shifting Arguments, Multiplicity Queues

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.34

Related Version *Short Version*: https://doi.org/10.1145/3583668.3594602 [17]
Preliminary Version: https://arxiv.org/abs/2305.11286 [18]

Funding *Anh Tran*: Bucknell University Emerging Scholars Program.
Edward Talmage: Bucknell University.

1 Introduction

In the search for efficient structured access to shared data, *relaxed data types* [5] have risen as an efficient way to trade off some of the precise guarantees of an ordered data type for more performance [14]. Multiplicity queues are a recently-developed relaxation of queues [4] which allow concurrent *Dequeue* instances to return the same value. Since they cannot have a sequential specification (being defined in terms of concurrency), previous results on relaxed queues do not apply to multiplicity queues.

Multiplicity queues are particularly interesting due to the implications for the computational power of the type. In [4], Castañeda et al. implement multiplicity queues from *Read*/*Write* registers, which is impossible for FIFO queues. This means that it is possible to have queue-like semantics without the cost of strong primitive operations like *Read*-

Modify-Write. Further work showed that this allows interesting applications in areas such as work-stealing [3] and more efficient implementations in shared memory systems with strong primitives than the best known algorithms for FIFO queues [7].

We are interested in message-passing implementations of data types, which provide the simplicity and well-defined semantics of a shared memory system in the message passing models inherent to geographically distributed systems [2]. Specifically, we want to implement shared structures efficiently in terms of the time between when a user invokes an operation and when the algorithm generates the operation's response. In queue implementations, the need for concurrent *Dequeue* instances to wait long enough to learn about each other, so that they can be sure to return different values, is one of the primary reasons that *Dequeue* is expensive to implement [19]. Between the higher performance multiplicity queues achieve in shared memory models and the intuitive notion that concurrent *Dequeue* instances need not learn about each other, it seems intuitive that multiplicity queues should have efficient implementations in message-passing systems.

However, recent work [13] showed the possible performance gains are limited. In a partially-synchronous system with maximum message delay d and delay uncertainty u, that work showed that the worst-case invocation-return delay for *Dequeue* is at least $\min\left\{\frac{2d}{3}, \frac{d+u}{2}\right\}$, which is within roughly a factor of 2 of a known unrelaxed queue implementation, where $|Dequeue| = d + \epsilon$ [19]. We here extend the work in [13], increasing the lower bound for the return time of *Dequeue* in uniform algorithms (those whose behavior does not depend on the number of participating processes) to $\min\left\{\frac{3d+2u}{5}, \frac{d}{2} + u\right\}$. Intuitively, while a particular *Dequeue* instance may not need to know about other, concurrent instances, determining which instances are concurrent is expensive in its own right.

This suggests new insights into fundamental properties of message passing implementations of shared data structures, showing that differentiating previous from concurrent instances is responsible for much of the time operation instances require. This could help develop more efficient algorithms or new relaxations with minimal weakening while providing maximum performance improvements. The piecewise nature of our bound also provides potential insight into what the optimal lower bound may be. We show that the $\frac{d}{2} + u$ portion of the bound, which has lower slope, is actually tight in the case when all messages have equal delay ($u = 0$). However, for larger uncertainties in message delay ($u > d/6$), the $\frac{3d+2u}{5}$ portion of the bound is higher. In fact, if we could strengthen the base case of our induction, it appears our argument would give larger bounds than $\frac{3d+2u}{5}$ for large values of u. That base case is already the most complex portion of our proof, so such strengthening and finding an optimal lower bound remain as future work. The two parts of the bound we show here relate to different constraints, with $\frac{d}{2} + u$ relating to admissibility of runs in our proof and $\frac{3d+2u}{5}$ relating to how fast information can travel through the system and when processes can make conclusions about the ordering of *Dequeue* instances at other processes.

Except for the edge case of $u = 0$, where they match and which we show is tight, our new bound is larger than the previous state of the art, and further demonstrates better tools for larger lower bound proofs in general. The proofs in [13] relied on shifting and other indistinguishability arguments among three or fewer processes, and the bounds were limited by the number of processes. In this paper, we develop more complex indistinguishability arguments, using inductive definitions of different runs of the algorithm on many processes. This requires using advanced shifting and indistinguishability tools, similar to those developed in [19]. These stronger tools allow us to prove larger bounds, and are interesting in their own right as a hint to how to prove larger lower bounds on other problems, as well.

1.1 Related Work

The idea of relaxing data types grew out of the study of consistency conditions weaker than linearizability. Afek et al. proposed Quasi-Linearizability in [1], which requires linearizations to be within a certain distance of a legal sequence, instead of themselves legal. From another perspective, this is just an expansion of the set of legal sequences on the data type. In [5], Henzinger et al. formalized these relaxations of abstract data type specifications by increasing the set of legal sequences and defined several parameterized ways to do so.

These relaxations and other work which followed [14, 12, 15] concentrated on relaxing sequential data type specifications and showed that they have more efficient implementations in a message passing system than an unrelaxed queue does. This approach cannot consider concurrency, which is simply not defined in the sequential space, so Castañeda et al. [4] defined multiplicity queues, which allow different behavior in the presence of concurrent operations than during sequential operation. They also replaced linearizability with set-linearizability as the consistency condition to accommodate non-sequential data types definitions.

In shared memory models, multiplicity queues have a number of advantages over unrelaxed queues, and even other, sequential relaxations. Castañeda et al. showed that multiplicity queues can be implemented purely from *Read*/*Write* registers, which is impossible for FIFO queues [6] and most previously-considered relaxed queues [12, 16], as they have consensus number 2. This suggests that they may be a practical way to get queue-like behavior cheaply in shared memory systems. Castañeda and Piña [3] use multiplicity queues to provide the first work-stealing algorithms without strong synchronization requirements. Johnen et al. [7] considered the time complexity of shared memory implementations of queues, implementing multiplicity queues in $O(\log n)$ steps for each of *Enqueue* and *Dequeue*, while the best previous algorithm for unrelaxed queues took $O(\sqrt{n})$ steps [8].

2 Model and Definitions

2.1 System Model

To have parameters we can use to prove lower bounds, we work in a partially synchronous model of computation. Lower bounds in this model also apply in asynchronous models, so a high lower bound here is still meaningful. We work in the same system model as [13] and its precedents, a partially-synchronous, message passing model without process failures used in the literature for various shared data type implementation algorithms and lower bounds.[1] There are n processes, $\{p_0, \ldots, p_{n-1}\}$, participating in an algorithm implementing a shared memory object. Each process allows a user to invoke operations on the simulated shared memory object and generates responses to those invocations. Each user can invoke operations at any time when their process does not have a *pending* operation–an invocation which does not yet have a matching response. Processes have local clocks running at the same rate as real time, but each potentially offset from real time, and can use these clocks to set timers.

Processes are state machines, where operation invocations, message arrivals and timer expirations trigger steps, which may perform local computation, set timers, send messages, and generate operation responses. A *run* of an algorithm is a set of sequences of state machine steps, one sequence for each process. Each sequence in a run is a valid state machine history

[1] There are several distinct models which the literature calls partially synchronous. We consider a model that is never totally synchronous or totally asynchronous, but always has some uncertainty in message delay, and thus cannot perfectly synchronize processes [10].

with a real time for each step, and is either infinite or ends in a state with no unexpired timers and no messages sent to that process but not received. A run is *admissible* if every message send step has a uniquely corresponding message receive step with the *delay* between send and receive at least $d - u$ and at most d real time and the skew, or maximum difference between local clocks, is at most $\varepsilon := (1 - 1/n)u$ [10]. We assume d and $u \leq d$ are known system parameters.

An implementation must provide *liveness*: every operation invocation must have a matching response. We call this invocation and corresponding response pair an *operation instance*. We are exploring the time cost of implementations, as measured by the worst-case delay between an instance's invocation and response. For operation OP, $|OP|$ denotes the maximum, over all admissible runs of the algorithm, of the real time between the invocation and response of any instance of OP. We measure communication cost, so we assume local computation is instantaneous. We also restrict ourselves to *eventually quiescent* implementations, requiring that if there are a finite number of operation invocations in a run, there is a finite time after the last invocation by which processes reach and stay in a *quiescent* state with no messages in transit and no timers set. A *uniform* algorithm is independent of the number of processes, with each process' state machine is identical for all n.[2]

A sequential data type specification gives a set of operations the user may invoke, with argument and return types, and the set of legal sequences of instances of those operations. We are interested in data types whose behavior may depend on concurrency in a distributed system, so we consider *set-sequential data type specifications*. A set-sequential data type specification similarly defines the set of operations the user may invoke, but instead of a set of legal sequences of instances, specifies a set of legal sequences of *sets of instances*. Thus, not all instances in a run must be totally ordered relative to each other, but each set of instances must be totally ordered relative to other sets.

To determine whether an algorithm implementing a set-sequential data type is correct, we require it to be *set-linearizable*, as defined in [11] and [4]. Set-linearizability requires that for every admissible run of the algorithm, there is a total order of sets of operation instances which contains every instance in the run, is legal by the set-sequential data type specification, and respects the real-time order of non-overlapping instances. That is, there must be a way to place all operation instances in the run in sets and order those sets into a legal sequence such that for every pair of instances where op_1 returns before op_2's invocation, op_1 is in a set which precedes the set containing op_2. The classic notion of linearizability is a special case of set-linearizability where all sets are required to have cardinality 1.

2.2 Multiplicity Queues

A *queue* is a First-In, First-Out data type providing operations $Enqueue(arg)$ which returns nothing and $Dequeue()$ which returns a data value, where any sequence of operations instances is legal iff each *Dequeue* instance returns the argument of the earliest preceding *Enqueue* instance whose argument has not already been returned by a *Dequeue*, or $\perp$ (which cannot be an *Enqueue* instance's argument) if there is no such *Enqueue* instance. We consider a related data type called a *multiplicity queue*, defined in [4], with the same operations but defined set-sequentially.

[2] It may seem that broadcasting requires knowledge of n, but since each process can simply iterate across all neighbors, no logic changes for different n.

▶ **Definition 1.** *A* multiplicity queue *over value set V is a data type with two operations: Enqueue(arg) takes one parameter $arg \in V$ and returns nothing. Dequeue$()$ takes no parameter and returns one value in $V \cup \{\perp\}$, where $\perp$ is special "empty" value. A sequence of sets of Enqueue and Dequeue instances is legal if (i) every Enqueue instance is in a singleton set, (ii) all Dequeue instances in a set return the same value, and (iii) each Dequeue instance deq returns the argument of the earliest Enqueue instance preceding deq in the sequence, which has not been returned by another Dequeue instance preceding deq. If there is no such Enqueue instance, deq returns $\perp$.*

This definition implies that concurrent *Dequeue* instances may, but do not necessarily, return the same value. If they do, they would set-linearize in the same set. If two *Dequeue* instances are not concurrent, then one must precede the other in the set linearization, so they must return different values. We assume all *Enqueue* arguments are unique, which is easily achieved by a higher abstraction layer timestamping the users' arguments.

2.3 Shifting Proofs

To prove our lower bounds, we will use indistinguishability arguments, where we argue that in a given time range in two runs, one or more processes have the same inputs (invocations, messages, timers) at the same local clock times. Since each process is a (deterministic) state machine, a process receiving the same inputs at the same times performs the same steps in the two runs. We will sometimes argue indistinguishability directly, but in some cases we will use *shifting* [10, 9, 19], a technique which mechanically changes the real time of events at one or more processes, while adjusting message delays and clock offsets such that each event happens at the same local time. Thus, if one run is a shift of another, they are indistinguishable. More formally, given run R and vector $\vec{v}$ of length n, we define $Shift(R, \vec{v})$ as a new run in which each event e at each $p_i, 0 \leq i < n$ which occurs at real time t in R occurs at real time $t + v[i]$. In $Shift(R, \vec{v})$ each local clock offset c_i becomes $c_i' = c_i - v[i]$. Finally, any message from p_i to p_j which had delay x in R has delay $x + v[j] - v[i]$, as the real times when it is sent and received change.

A challenge in using shifting arguments is that the shifted run must be admissible to require the algorithm to behave correctly. Great care is required to define a run's message delays and clock offsets so that the skew and message delays are admissible after shifting. We use an extended shifting technique by Wang et al. [19] which shows that if a shift is too large, making the shifted run inadmissible, it is in some cases possible to chop off each process' sequence of steps before a message arrives after an inadmissible delay, then extend the run from that collection of chop points with different message delays which are admissible. This extended run is not necessarily indistinguishable past the chop, but we can sometimes argue that the runs are indistinguishable long enough to form conclusions about the new run's behavior.

3 Lower Bound Proof Outline

Our primary result is a lower bound on the worst-case time of any uniform set-linearizable implementation of a multiplicity queue. For comparison, a linearizable implementation of an unrelaxed FIFO queue is possible with worst-case *Dequeue* cost $d + \varepsilon = d + (1 - 1/n)u$. Our lower bound is over half of that, indicating a limit on the performance gains of the multiplicity relaxation. Since our lower bound shows the impossibility of a more efficient multiplicity queue implementation in a relatively well-behaved partially synchronous model of computation, it follows that it is similarly impossible in more realistic, and less well-behaved, models, such as those with asynchrony or failures. It is possible that the cost to port a FIFO queue to a less well-behaved model is higher than the cost to port a multiplicity queue, but that remains an open question.

We prove our bound by building up two sets of runs. In both sets, each process invokes a single *Dequeue* instance. In the first set we show that each of these *Dequeue* instances, despite being concurrent with at least one other *Dequeue* instance, returns a unique value. In the second set, we show that there are fewer distinct return values than *Dequeue* instances, so there must be some *Dequeue* instances returning the same value. We then show that, for sufficiently large n, these two sets of runs eventually converge, in the sense that processes cannot distinguish which set they are in until after they choose return values for their *Dequeue* instances. This means they must have the same behavior in both, a contradiction which implies the worst-case cost of *Dequeue* must be higher. We need large n to ensure that the information about all of the *Dequeue* instances cannot reach the last process in time for it to distinguish which run it is in, so the lower bound on n depends on the relationship between d and u, increasing as u approaches d.

Both sets of runs are based on and building towards one simple run, which sequentially enqueues values $1..n$, then once the system is quiescent, has each process dequeue once, with invocation times staggered so that the *Dequeue* instances at different processes overlap slightly (unless $|Dequeue| < u$, in which case processes invoke *Dequeue* simultaneously, which the variable s below handles). Any set linearization of any of our runs will start with n singleton sets, enqueueing the values $1..n$ in order. Further operation instances will set-linearize after those *Enqueue* instances. In general, messages from lower-indexed processes to higher-indexed processes take $d - u$ time, while those from higher-indexed processes to lower-indexed processes take d time. The primary exception is that after a certain point, messages from p_{n-1} to p_n will also take d time. This prevents p_n from collecting complete information on other processes' actions, which is enough uncertainty to cause incorrect behavior. As we develop our proof, we will modify other delays, but start from this pattern.

Let $Q := \min\left\{\frac{3d+2u}{5}, \frac{d}{2} + u\right\}$ throughout the paper. To prove that $|Dequeue| \geq Q$, in the following we assume for the sake of contradiction that $|Dequeue| < Q \leq d$.

4 Distinct Return Values

For our first set of runs, we show that each *Dequeue* instance may return a distinct value, despite the fact that each is concurrent with at least one other instance. While this is the easier part of the proof, it is interesting as it shows that, under uncertainty in message delay, processes cannot tell whether their *Dequeue* instances are or are not concurrent, so the relaxation gives no advantage, as processes must spend time to choose distinct return values.

We denote this set of runs by $D_k, 1 \leq k \leq n$, where the first k processes invoke *Dequeue* instances slightly overlapped as discussed, and higher-indexed processes invoke *Dequeue* slightly later. We will inductively show that the *Dequeue* at p_k must return a different value from those at $p_0, \ldots, p_{k-1}$, then shift the run to obtain D_{k+1}, which is indistinguishable. When the inductive chain of shifts is complete, we will see that all n *Dequeue* instances in D_n must return different values.

▶ **Construction 2.** *Define run D_k (D for D̲istinct) as follows, for each $1 \leq k < n$:*

- *p_0 invokes Enqueue$(1) \cdots$ Enqueue(n) in order. Let t be any arbitrary time after Enqueue(n) returns at which the system is quiescent.*
- *$\forall 0 \leq i < k$, process p_i invokes Dequeue at time $t + i \times s$, where $s = \max\{0, Q - u\}$.*
- *$\forall k \leq j < n$, process p_j invokes Dequeue at time $t + (j-1)s + (s+u)$.*

- *Process p_0 has local clock offset $c_0 = 0$.*
- *$\forall 0 < i < k$, process p_i has local clock offset $c_i = \left(\frac{i}{n}\right) u$.*
- *$\forall k \le j < n$, process p_j has local clock offset $c_j = \left(\frac{j-n}{n}\right) u$.*
- *$\forall 0 \le i < k \le j < n$, messages from p_i to p_j have delay d, from p_j to p_i have delay $d-u$.*
- *$\forall 0 \le a < b < k$, messages from p_a to p_b have delay $d-u$, from p_b to p_a have delay d.*
- *$\forall k \le c < d < n$, messages from p_c to p_d have delay $d-u$, from p_d to p_c have delay d.*

Define run D_n similarly, but setting clock offsets, invocations, and message delays only for processes p_i with $i < n$. Since p_n does not exist, it does not invoke Dequeue, send or receive messages, or have a local clock offset.

Since all local clock offsets for processes p_i with $0 < i < k$ are positive and increase with i and all offsets for processes p_j with $k \le j < n$ are negative and increase with j, the maximum skew between processes is $|c_{k-1} - c_k| = \left|\frac{k-1}{n} - \frac{k-n}{n}u\right| = \frac{n-1}{n}u = \varepsilon$, except when $k = n$, when no such p_j exists and the maximum skew is $\left|0 - \frac{n-1}{n}u\right| = \varepsilon$. With this fact and since all message delays are in the range $[d-u, d]$, we see that each D_k is an admissible run.

Our first step is to note that each D_k is a shifted version of D_{k-1}, which implies that they are all indistinguishable. The proof is a straightforward application of classic shifting, adjusting real times, clock offsets, and message delays, and is included in the appendix. Then we can prove that each *Dequeue* instance in D_n must return a different value.

▶ **Lemma 3.** *For all $2 \le k < n$, $D_k = Shift(D_{k-1}, \overrightarrow{s_{k-1}})$, where $\overrightarrow{s_{k-1}}$'s only non-zero component is $-u$ at index $k-1$: $\overrightarrow{s_{k-1}} = \langle 0, \ldots, 0, -u, 0, \ldots, 0\rangle$.*

▶ **Lemma 4.** *In run D_k, $1 \le k \le n$, every Dequeue instance returns a distinct value. Specifically, for each $0 \le i < n$, the Dequeue instance at p_i returns $i+1$.*

Proof. Consider D_1. Here, p_0 invokes *Dequeue* at time t, which must return by time $t + |Dequeue|$. p_1 invokes *Dequeue* at time $t + (1-1)s + (s+u) > t + |Dequeue|$, which is after p_0's *Dequeue* instance returns. Every process p_i with $i \ge 1$ invokes *Dequeue* no earlier than p_1, so no other *Dequeue* instance is concurrent with p_0's, and thus that one must set-linearize before any other. This means that p_0 returns 1 to its *Dequeue* instance and all other processes return values in the set $\{2, \ldots, n\}$ to their *Dequeue* instances.

Assume that for some arbitrary $0 \le k < n-1$, each process p_i, $0 \le i < k$ returns $i+1$ to its *Dequeue* instance. We will then show that process p_k returns $k+1$ to its *Dequeue* instance. First, note that in D_k, p_k invokes *Dequeue* at time $t + (k-1)s + (s+u)$, while every $p_i, 0 \le i < k$ has its *Dequeue* instance return no later than $t + (i-1)s + |Dequeue| \le t + ((k-1)-1)s + |Dequeue| < t + (k-2)s + (s+u)$. Since $s \ge 0$, this is before p_k invokes *Dequeue*, so p_k's *Dequeue* instance must set-linearize strictly after all of those at any lower-indexed p_i. By the inductive hypothesis, each of those k processes returns $i+1$, so p_k must return a value larger than k.

Now, consider D_{k+1}. Since D_{k+1} is a shifted version of D_k, no process can distinguish the two runs, so all behave the same way in both. Specifically, p_k will return the same value to its *Dequeue* instance. But in D_{k+1}, by an identical argument to that in the previous paragraph, each $p_j, k < j < n$ invokes *Dequeue* after the *Dequeue* instance at p_k returns, so they must all set-linearize strictly after p_k's *Dequeue* instance, and those of each $p_i, 0 \le i \le k$. Since there are only $k+1$ *Dequeue* instances set-linearized with or before that at p_k, these must return values from the set $\{1, \ldots, k+1\}$. But we know that those at processes with indices in $\{0, \ldots, k-1\}$ all return values from $\{1, \ldots, k\}$, and the *Dequeue* instance at p_k returns a value distinct from any of these, so it must return $k+1$, and we have the claim. ◀

5 Repeated Return Values

For the second set of runs, we will show that n processes, each invoking one *Dequeue* instance in our same partially-overlapping pattern, will not return all different values to those *Dequeue* instances. To do this, we first show that if only three processes invoke *Dequeue*, then they will only return two distinct values. We then inductively construct more and more complex runs, with one more process joining the pattern and invoking *Dequeue* in each successive pair of runs. When the induction reaches n, we will show that we have a run indistinguishable from the D_n we constructed in the previous section. Since each of the *Dequeue* instances in that run returns a distinct value, and those in the run we construct here do not all return distinct values, we have a contradiction, proving that the assumed algorithm cannot exist.

First, we define the family of runs S_k, in each of which only $k \leq n$ processes invoke *Dequeue*. We inductively show that each of these has some pair of *Dequeue* instances which return the same value, eventually showing that not all *Dequeue* instances in S_n return distinct values. Then, to show the chain of indistinguishabilities in our induction, we will need another set of runs, which are intermediate steps.

▶ **Construction 5.** *Define run S_k (S for Same) as follows:*

- *p_0 invokes Enqueue$(1) \cdots$ Enqueue(n) in order. Let t be the same arbitrary time after Enqueue(n) returns at which the system is quiescent as in the definition of D_k.*
- *$\forall 0 \leq i < k$, process p_i invokes Dequeue at time $t + i \times s$, where $s = \max\{0, Q - u\}$.*
- *Process p_0 has local clock offset $c_0 = 0$, and $\forall 0 < i < n$, process p_i has $c_i = \left(\frac{i}{n}\right) u$.*
- *$\forall 0 \leq i < j < n$, messages from p_j to p_i have delay d and from p_i to p_j have delay $d - u$, except for those from p_{k-2} to p_{k-1} sent after $t^*_{k-2} = t + (k-2)(d-u)$, which have delay d.*

▶ **Construction 6.** *For $1 \leq k < n$, define run S'_k from run S_{k-1} by additionally having p_{k-1} invoke Dequeue at time $t + (k-1)s$. Adjust the delay of all messages from p_{k-2} to p_{k-1} sent at or after $t^*_{k-2} = t + (k-2)(d-u)$ to d.*

In S'_k, we have added the next *Dequeue* instance, but have two processes' messages (p_{k-3}'s and p_{k-2}'s) to the next, larger-indexed, process delayed. We can show that processes p_0 through p_{k-2} cannot distinguish S_{k-1} from S'_k before generating return values for their *Dequeue* instances, so they must return the same values, which gives us information about what p_{k-1} must return to its *Dequeue* instance. We then show that S'_k and S_k are indistinguishable to p_{k-1} until after it has generated a return value for its *Dequeue* instance, telling us what values it could return. The prof is by mathematical induction on k, from 3 to n.

▶ **Lemma 7.** *For sufficiently large n, all Dequeue instances in S'_n and S_n return values from $\{1, \ldots, n-1\}$.*

Proof. We proceed by induction on k.

Base Case. We proceed by induction on k, with base case $k = 3$, when only the first three of our n processes invoking *Dequeue*, which is run S_3. We show that all *Dequeue* instances return values from $\{1, 2\}$, for sufficiently large n. Due to higher-indexed processes invoking *Dequeue* later than lower-indexed processes, and the way we will set message delays, the first *Dequeue* instance will behave as if it were running alone, returning 1. We will then shift run S_3, using a technique like that in [19] that allows us to over-shift and break some message delays, then re-insert those messages with new, admissible delays. We can then show that the resulting patched run is still indistinguishable from the starting run for long enough. In this run, we will argue that the second process does not learn about the first process' *Dequeue* instance until after its own returns, and thus cannot distinguish this run from one in which it is running alone, so it must also return 1. Given these two return values, set-linearizability implies that the third process' *Dequeue* instance must return 2. We will then show that the third process cannot distinguish between the original and shifted runs before choosing its return value, so will return 2 in S_3.

First, observe that p_0 cannot learn about the *Dequeue* instances at p_1 and p_2 until after its own *Dequeue* instance has returned. Since all messages from a higher-indexed process to a lower-indexed process have delay d, the earliest p_0 will learn about the other *Dequeue* instances is at time $t + s + d$, since $t + s$ is when p_1 invokes *Dequeue*, and any message indicating that this has happened will take d time to reach p_0. Since p_2 invokes its *Dequeue* instance at time $t + 2s \geq t + s$, the same logic will imply that p_0 will also not learn about that instance until after its own *Dequeue* instance has returned. p_0's *Dequeue* instance returns no later than time $t + |Dequeue|$, by definition, which is strictly less than $t + d$. Together, we see that p_0 learns about a remote *Dequeue* invocation no sooner than $t + s + d \geq t + d > t + |Dequeue|$, so through the return of its *Dequeue* instance, p_0 cannot distinguish S_3 from a run in which that is the only *Dequeue* instance. Thus, it returns the same value, which by set-linearization is necessarily 1. Similarly, p_1 must return a value in $\{1, 2\}$, since it cannot learn about the *Dequeue* instance at p_2 until time $t + 2s + d$, which is larger than when its own *Dequeue* instance returns by $t + s + |Dequeue|$.

Next, we want to show that p_2 will also return a value from $\{1, 2\}$ to its *Dequeue* instance. We cannot directly argue this, since if p_2 learns about both the *Dequeue* instances at p_0 and p_1 before it generates a return value for its own, it may decide to return a different value than either. Instead, we will shift events at p_1 earlier, then argue that in this run, the information about p_0's *Dequeue* instance does not arrive at p_1 until after it has generated its *Dequeue* return value, forcing p_1 to return 1 to its *Dequeue* instance. Now, while p_1 may be able to distinguish this new run from S_3, we will argue that p_2 will not be able to distinguish them until after it generates its *Dequeue* return, so must return the same value in both. In the shifted run, p_0 and p_1 will both return 1, which means that p_2 must return either 1 or 2 to satisfy set-linearizability.

We will shift S_3 by the vector $\langle 0, -X, 0, \ldots, 0\rangle$, where X is a value we will determine shortly. Next, we will alter message delays, both to delay p_1 from learning about p_0's *Dequeue* instance and to make the shifted run admissible, yielding run S_3^X.

Our first step is to find what shift amounts X for p_1 will make S_3^X admissible, then argue the behavior of each process. First, note that this shift will increase the local clock offset of p_1 by X. In S_3, $c_1 = \frac{1}{n}u$, the smallest clock offset is $c_0 = 0$ and the largest is $c_{n-1} = \frac{n-1}{n}u$. To keep the run admissible, we must have $X \leq \left(\frac{n-1}{n}u - \frac{1}{n}u\right) = \frac{n-2}{n}u$, since we are not changing the smallest clock offset, so must keep c_1 within ε of that offset.

Next, as shown in Table 1, we see that for a non-negative value of X, we will have some inadmissible message delays in $\mathit{Shift}(S_3, \langle 0, -X, 0, \ldots, 0\rangle)$ (highlighted in the $\mathit{Shift}(S_3)$ column). To correct these, we trim the run before any of the inadmissible messages would arrive, then extend the run with other, admissible message delays (highlighted in the S_3^X column), following the technique introduced in [19]. Unlike a shift, this may change the behavior of the run, so we must argue what each process does in run S_3^X. We chose these new delays to delay processes from learning about remote actions, setting all of the adjusted delays to the maximum, d. Since $X \leq \frac{n-2}{n}u < u$, then the delays not highlighted in the S_3^X column are in the range $[d - u, d]$, and we conclude that if $0 \leq X \leq \frac{n-2}{n}u$, then S_3^X is admissible.

Table 1 Table showing message delays to and from p_1 in runs for base case in repeated *Dequeue* return values proof. Delays for other processes are unchanged across the three runs.

Message Path	S_3	$Shift(S_3)$	Adjusted: S_3^X
$p_0 \to p_1$	$d-u$	$d-u-X$	d
$p_1 \to p_0$	d	$d+X$	d
$p_1 \to p_2$ (initially)	$d-u$	$d-u+X$	$d-u+X$
$p_1 \to p_2$ (after t_1^*)	d	$d+X$	d
$p_1 \to p_{\geq 3}$	$d-u$	$d-u+X$	$d-u+X$
$p_{\geq 2} \to p_1$	d	$d-X$	$d-X$

Now that we know what values of X make S_3^X an admissible run, we will find which of those values of X will make all three *Dequeue* instances return values from $\{1,2\}$ in S_3^X.

p_0 will not learn about p_2's *Dequeue* instance until after its own returns, by the same argument as in S_3. We want p_0 to also not learn about p_1's *Dequeue* instance until after its own returns. p_1 invokes *Dequeue* at $t+s-X$ in S_3^X, since we shifted events at p_1 earlier by X. A message sent at this time will arrive at p_0 at time $t+s-X+d$, and we want to argue that this will be after $t+|Dequeue|$, and thus after p_0's *Dequeue* instance returns. This happens if and only if $d+s-X>|Dequeue|$, or $X<d+s-|Dequeue|$. Since $s\geq 0$, it is sufficient to require that $X<d-|Dequeue|$ to ensure that p_0's *Dequeue* instance returns 1.

To force p_1's *Dequeue* instance to return 1, we want information about p_0's invocation of *Dequeue* to arrive after p_1 generates its *Dequeue* return value. Thus, we want to have time $t+d$ (since messages from p_0 to p_1 have delay d in S_3^X) later than when p_1 generates a return value. p_1 invokes *Dequeue* at time $t+s-X$ and the *Dequeue* instance returns at most $|Dequeue|$ time after invocation, so we want to have $t+d>t+s-X+|Dequeue|$. Solving for X, we find that this is true iff $X>|Dequeue|+s-d$. Here, we split into cases depending on the value of s: If $s=0$, we want $X>|Dequeue|-d$, but we assumed that $|Dequeue|<d$, so any non-negative value of X is sufficient. If $s=Q-u$, we want $X>|Dequeue|+(Q-u)-d=|Dequeue|+Q-(d+u)$.

Similarly to previous arguments, since p_2 invokes *Dequeue* at least X after p_1 does (p_2 invokes *Dequeue* s after p_1 in S_3, which means $s+X$ after in S_3^X), and message delays from p_2 to p_1 are $d-X$, p_1 cannot learn about p_2's *Dequeue* invocation until at least $X+(d-X)=d>|Dequeue|$ after p_1 invokes *Dequeue*. This is after p_1 generates its *Dequeue* return value. Combining this with the previous conclusion that p_1 is unaware of p_0's *Dequeue* invocation until after it chooses a return value, we conclude that p_1 will return the same value as in a run where neither p_0 nor p_2 invoked *Dequeue*. The only legal set-linearization of such a run requires that p_1 return 1.

We can now reason about p_2's behavior. Since both p_0 and p_1 must return 1 to their *Dequeue* instances in S_3^X, we conclude that p_2 must return either 1 or 2, as there is no legal set-linearization of any other return value. We will thus argue that p_2 cannot distinguish S_3^X from S_3 until after it generates its *Dequeue* return value, concluding that p_2 will return either 1 or 2 to its *Dequeue* instance in S_3 as well.

Consider when each process in S_3 can first distinguish that it is not in S_3^X. These differences correspond to the adjusted message delays highlighted in the final column of Table 1. p_0 can distinguish the runs when it does not receive a message p_1 may have sent at its *Dequeue* invocation as soon as it would have received it in S_3^X, since in S_3^X we reduced the delay on messages from p_1 to p_0. This detection would occur at time $t+s+(d-X)$, when that message does not arrive. p_1 can first distinguish the runs at time $t+(d-u)$, when it can receive a message p_0 sent at its *Dequeue* invocation but which arrives later in S_3^X, where we increased the delay on messages from p_0 to p_1. Note $t+s+(d-X)+(d-u) > t+(d-u)$ and $t+s+(d-X) \leq t+(d-u)+d$, so neither process can send a message after it detects the difference which will arrive before the recipient detects the difference itself.

Finally, p_2 can distinguish the runs either by receiving a message p_0 or p_1 sends after distinguishing the runs or directly from adjusted message delays. We argue that each of these must occur after the *Dequeue* instance at p_2 returns, so p_2 cannot distinguish S_3 from S_3^X until after that *Dequeue* instance's return value is set, so the value must be the same in both runs.

Consider when p_2 can receive a forwarded detection of a difference in the runs:

- p_0 can send this information no sooner than $t+s+(d-X)$, and the message would take d time to arrive, meaning that the earliest p_2 could distinguish the runs based on this information is $t+s+2d-X$. We want to show that this is greater than $t+2s+|Dequeue|$, and thus after p_2's *Dequeue* instance returns. This is true iff $2d-X-u > s+|Dequeue|$. Consider cases for the value of s:
  - $s=0$: We want to show that $2d-X-u > |Dequeue|$. This is true if and only if $X < (d-|Dequeue|)+(d-u)$, but we know that $d \geq u$ so this holds if $X < d-|Dequeue|$.
  - $s=Q-u$: We want to show that $2d-X-u > |Dequeue|+Q-u$. This is true if and only if $X < (d-Q)+(d-|Dequeue|)$. Since $Q \leq d$ and we are already assuming $X < d-|Dequeue|$, this inequality holds.
- p_1 can send a message informing p_2 that it is in S_3, not S_3^X, no sooner than $t+(d-u)$. Since $t+(d-u)=t_2^*$, this message will take d time to arrive at p_2. We want to show that this is after p_2's *Dequeue* instance returns, which happens at $t+2s+|Dequeue|$. Thus, we want $t+(d-u)+d > t+2s+|Dequeue|$, or $2d-u > 2s+|Dequeue|$. Solving for $|Dequeue|$, this is equivalent to $|Dequeue| < 2d-2s-u$. Consider the possible values of s:
  - $s=0$: We want to show that $|Dequeue| < 2d-u$. But we know that $d \geq u$, so $d-u \geq 0$ and $|Dequeue| < d$, so this inequality holds.
  - $s=Q-u$: We want to show that $|Dequeue| < 2d-2(Q-u)-u = 2d-2Q+u$. But $|Dequeue| < Q$, so it is sufficient to show that $Q \leq 2d-2Q+u$. This holds iff $Q \leq \frac{2d+u}{3}$. But we assumed $Q \leq \frac{3d+2u}{5} \leq \frac{2d+u}{3}$, so we have the desired relationship.

  Thus, p_2 cannot learn from p_1 that it is in S_3^X before it generates a return value for its *Dequeue* instance.

Finally, we show that p_2 cannot directly differentiate S_3 from S_3^X based on the altered message delays in S_3^X before its *Dequeue* instance returns. At the earliest, this can happen at t_2^*+d-X, when p_2 does not receive a message in S_3 that it may have in S_3^X, since in S_3^X we decreased the delay of messages p_1 sends to p_2 at or after time t_2^*-X. We again want to show that this is after p_2's *Dequeue* instance returns which happens no later than $t+2s+|Dequeue|$. That is, we want $t_2^*+d-X = t+(d-u)+d-X > t+2s+|Dequeue|$. Equivalently, we want $2d-u-X > 2s+|Dequeue|$. Consider cases for s:

- $s=0$: In this case, we want $2d-u-X > |Dequeue|$, which is true iff $X \leq (d-|Dequeue|)+(d-u)$. We already have the constraint that $X < d-|Dequeue|$ and $u \leq d$.
- $s=Q-u$: Here, we want $2d-u-X > 2(Q-u)+|Dequeue|$, which is true if $X < 2d+u-2Q-|Dequeue|$. This is a new constraint on X which we must meet.

Thus, in all cases (if X meets all our constraints simultaneously), p_2 cannot distinguish S_3 from S_3^X until after its *Dequeue* instance returns. This means it returns the same value in both runs, and since we proved it returns a value from $\{1,2\}$ in S_3^X, it also does in S_3.

Our last step is to verify that our constraints on X are compatible–that there is a value of X which will make S_3^k admissible and give the behavior we want. Our constraints are

- $X \geq 0$ and $X > |Dequeue| + Q - (d+u)$
- $X < d - |Dequeue|$, $X < 2d + u - 2Q - |Dequeue|$, and $X \leq \frac{n-2}{n}u$

These three upper bounds and two lower bounds lead to six cases to check to show that there exists a value of X which satisfies all of our constraints.

- $d - |Dequeue| > 0$: By assumption, $|Dequeue| < d$, so $d - |Dequeue| > 0$.
- $d - |Dequeue| > |Dequeue| + Q - (d+u)$: This is true iff $2d + u > 2|Dequeue| + Q$. Since $|Dequeue| < Q$, it is sufficient to show that $2d + u \geq 3Q$, or $Q \leq \frac{2d+u}{3}$, but we assumed that $Q \leq \frac{3d+2u}{5} \leq \frac{2d+u}{3}$, so this relationship holds.
- $2d + u - 2Q - |Dequeue| > 0$: This is equivalent to the previous case.
- $2d+u-2Q-|Dequeue| > |Dequeue|+Q-(d+u)$: This is true iff $3d+2u > 3Q+2|Dequeue|$, but it is sufficient to show that $3d + 2u \geq 5Q$, and we assumed that $Q \leq \frac{3d+2u}{5}$, so this relationship holds.
- $\frac{n-2}{n}u \geq 0$: $n > 2$ and $u \geq 0$, and a positive fraction of u will thus be non-negative.
- $\frac{n-2}{n}u > |Dequeue| + Q - (d+u)$: Solving for $|Dequeue|$ and Q, this is true iff $Q + |Dequeue| < d + \frac{n-2}{n}u + u$. Since $|Dequeue| < \frac{d}{2} + u$, there is some N_0 s.t. for all $n \geq N_0$, $|Dequeue| < \frac{d}{2} + \frac{n-2}{n}u$. Further $|Q| \leq \frac{d}{2} + u$, so combining these, the inequality holds for $n \geq N_0$.

Thus, since every upper bound is larger than every lower bound, for sufficiently large n ($n \geq N_0$), there exists at least one X such that S_3^X is admissible and p_0, p_1, and p_2 all return values from $\{1, 2\}$ to their *Dequeue* instances, and we have the claim.

Inductive Case. Assume that for some arbitrary $4 \leq k \leq n$, all *Dequeue* instances in S_{k-1} return values from the set $\{1, \dots, k-2\}$. We will show that in S_k and S'_k, all *Dequeue* instances return values from the set $\{1, \dots, k-1\}$. First, we will use S_{k-1} to argue the behavior of S'_k, then use that behavior to prove the behavior of S_k.

To show that in S'_k, all processes return values from the set $\{1, \dots, k-1\}$ to their *Dequeue* instances, we argue that processes $p_0, \dots, p_{k-1}$ cannot distinguish S_{k-1} from S'_k until after they have all generated their *Dequeue* return values. Thus, they will return the same values as in S_{k-1}, which are all in $\{1, \dots k-2\}$ by the inductive hypothesis. We can then conclude that p_{k-1}, which invokes a *Dequeue* instance in S'_k but not in S_{k-1} must return a value in the set $\{1, \dots, k-1\}$ to satisfy set-linearizability.

Recall that S'_k differs from S_{k-1} in two ways: First, p_{k-1} invokes *Dequeue* at time $t + (k-1)s$. Second, messages from p_{k-2} to p_{k-1} sent at or after $t^*_{k-2} = t + (k-2)(d-u)$ have delay d instead of $d - u$. Thus, the first point at which any process can discern that it is in S'_k instead of S_{k-1} is p_{k-1} at whichever of these events happens first. For any other process, the first point where it can distinguish the runs is when it can receive a message p_{k-1} sends after it discerns the difference. We will argue that such a message arrives at any of $p_0, \dots, p_{k-2}$ after it has chosen a return value for its *Dequeue* instance. Note that we need only prove that such a message arrives at p_{k-2} more than $|Dequeue|$ after it invokes *Dequeue*, since each process with a lower index invokes *Dequeue* at the same time or sooner, and the message delay from p_{k-1} to any lower-index process is the same. We proceed by cases on which distinguishing event at p_{k-1} occurs first.

- p_{k-1} first distinguishes the runs when it invokes *Dequeue*: The message delay from p_{k-1} to p_{k-2} is d, and any indirect path would take even longer, since any such path must have some message from a higher-indexed to lower-indexed process, which has delay d. Thus, the earliest p_{k-2} can distinguish the runs is $t + (k-1)s + d$. We want to show

that this is later than the return time of p_{k-2}'s *Dequeue* instance, which must return by $t+(d-2)s+|Dequeue|$. This inequality is true iff $t+(k-1)s+d > t+(k-2)s+|Dequeue|$, which reduces to $s+d > |Dequeue|$.
Since $d > Q$ and $s \geq 0$, this inequality holds, which means that p_{k-2} (and similarly $p_0, \ldots, p_{k-3}$) cannot use the extra *Dequeue* invocation at p_{k-1} to distinguish S'_k from S_{k-1} until after their *Dequeue* instances have returned.

- p_{k-1} first distinguishes the runs when it fails to receive a message whose delay was increased: The earliest possible sending time of such a message is $t^*_{k-2} = t+(k-2)(d-u)$. p_{k-1} can detect that it has not arrived $d-u$ later (when it would have arrived in S_{k-1}), and then the earliest it can get information about the differentiation to a lower-indexed process is another d after that. We similarly want to show that this is after the *Dequeue* instance at p_{k-2} returns, which is true iff $t+(k-2)(d-u)+(d-u)+d > t+(k-2)s+|Dequeue|$, which reduces to $(k-1)(d-u)+d > (k-2)s+|Dequeue|$.
Since $d > |Dequeue|$, $d \geq u$, and $s = \max\{0, Q-u\}$, we see that $d-u \geq s$, so the inequality holds. Thus, in this case no process in $p_0, \ldots, p_{k-2}$ can distinguish S'_k from S_{k-1} until after it has generated a return value for its *Dequeue* instance.

Since in neither case can $p_0, \ldots, p_{k-2}$ distinguish S'_k from S_{k-1} until after its *Dequeue* instance returns, all their *Dequeue* instances return the same values in both runs. Specifically, by the inductive hypothesis they all return values from the set $\{1, \ldots, k-2\}$. The *Dequeue* instance at p_{k-1} must then return a value in the set $\{1, \ldots, k-1\}$, as any larger value would violate set-linearizability, since there would be no *Dequeue* instance returning $k-1$.

Now, having determined the behavior of S'_k, we use it to show that S_k will behave similarly. This is another indistinguishability argument, showing that p_{k-1} cannot distinguish S_k from S'_k, until after it has generated a return value for its *Dequeue* instances. Recall that the difference between S_k and S'_k is that in R_k all messages from p_{k-3} to p_{k-2} have delay $d-u$, while in S'_k, those sent at or after t^*_{k-3} have delay d.

Before we start the indistinguishability argument, note that if p_k did not invoke *Dequeue* in S_k, the remaining $k-1$ *Dequeue* instances must return values from the set $\{1, \ldots, k-1\}$, since there would only be $k-1$ instances, so there would be no way to set-linearize an instance that returned a larger value. These processes must behave the same way in S_k as in this run, since the first point where any could detect a difference would be d after p_k's invocation, which is after all other *Dequeue* instances have returned, similar to prior arguments. Thus, we need only concern ourselves with showing that p_{k-1} cannot distinguish S_k from S'_k before its *Dequeue* instance returns, so that it will return a value in $\{1, \ldots, k-1\}$, as we proved it does in S'_k.

The only process which can directly detect a difference between S_k and S'_k is p_{k-2}, when it receives a message in S_k which arrives sooner than it could in S'_k. This occurs $d-u$ after time t^*_{k-3}, when the message delays changed. The soonest p_{k-1} can learn about the difference is when a message from p_{k-2}, sent after it detected the difference, could arrive. But $t^*_{k-3} + (d-u) = t^*_{k-2}$, so any message p_{k-2} sends to p_{k-1} after this point has delay d. Thus, the soonest p_{k-1} can distinguish S_k from S'_k is $t^*_{k-2} + d = t+(k-2)(d-u)+d$. We argue that this is after p_{k-1} generates its *Dequeue* return value, which occurs no later than $t+(k-1)s+|Dequeue|$. We thus want to show that $t+(k-2)(d-u)+d > t+(k-1)s+|Dequeue|$. Consider the cases for s:

If $s = 0$: The inequality holds iff $(k-2)(d-u)+d > |Dequeue|$, which is true because $d \geq u$ and $d > |Dequeue|$. If $s = Q-u$: The inequality holds if $(k-2)(d-u)+d > (k-1)(Q-u)+|Dequeue|$, or $(k-1)d > (k-1)Q - u + |Dequeue|$. Since $|Dequeue| < Q$, it is sufficient to show that $(k-1)d \geq kQ - u$, or $Q \leq \frac{(k-1)d+u}{k}$.

To prove this final inequality, recall that $Q \leq \frac{3d+2u}{5}$ and that $k \geq 4$. For all $k \geq 4$, $\frac{(k-1)d+u}{k} \geq \frac{3d+u}{4}$, so it suffices to show that $\frac{3d+2u}{5} \leq \frac{3d+u}{4}$. This follows because $\frac{3d+2u}{5} = \left(\frac{3d+u}{4}\right)\left(\frac{4}{5}\right) + \frac{u}{5}$, and $\frac{u}{5} \leq \left(\frac{1}{5}\right)\left(\frac{3d+u}{4}\right)$, as that inequality reduces to $u \leq d$, which is true.

We conclude that p_k cannot distinguish S_k from S'_k until after it generates a return value for its *Dequeue* instance, so it must return the same value in both runs, which we previously proved was in the set $\{1, \ldots, k-1\}$. Thus, by mathematical induction, when $k = n$, all *Dequeue* instances in S_n return values from the set $\{1, \ldots, n-1\}$, and we have the claim. ◀

6 Contradiction

Let us quickly recap what we have shown so far. First, we showed that there is a run D_n with n overlapping *Dequeue* instances each returning a different value. Then, we (somewhat laboriously) showed that there is a run S_n with n overlapping *Dequeue* instances in which two *Dequeue* instances return the same value. Now, we want to show that these runs are indistinguishable, a contradiction, as processes must return the same values in indistinguishable runs.

▶ **Theorem 8.** *There is no uniform, set-linearizable implementation of a multiplicity queue with* $|Dequeue| < \min\left\{\frac{d}{2} + u, \frac{3d+2u}{5}\right\}$.

Proof. Assume, in contradiction, that there is such an algorithm. Then the conditions for Lemma 4 and Lemma 7 are satisfied, so we know that S_n and D_n exist, where all *Dequeue* instances in S_n return values from $\{1, \ldots, n-1\}$ and the *Dequeue* instance at p_i in D_n returns $i+1$, for all $0 \leq i < n$. Recall that S_n requires that $n \geq N_0$, defined in Section 5 s.t. for all $n \geq N_0$, $|Dequeue| < \frac{d}{2} + \frac{n-2}{n}u$.

Note that S_n and D_n are nearly identical–they have the same initial sequence of *Enqueue* instances at p_0, the same clock offsets ($c_0 = 0$, $c_i = \frac{i}{n}u, 1 \leq i < n$), and the same *Dequeue* invocations (p_i invokes *Dequeue* at time $t + (i-1)s$). The two runs also have nearly identical message delays, where if $0 \leq i < j < n$, messages from p_j to p_i have delay d and those from p_i to p_j have delay $d - u$, except that in S_n, messages from p_{n-2} to p_{n-1} sent at or after time t^*_{n-2} have delay d. Thus, if we extend those message delays in D_n, we will have the same run. We will argue that we will still have D_n's behavior, which differs from S_n's, in the same run, which is a contradiction.

Suppose first that $u < d$. Construct D^* from D_n by delaying all messages from p_{n-2} to p_{n-1} sent at or after t^*_{n-2} by d. We argue that no process can distinguish that it is in D^* instead of D_n before its *Dequeue* instance returns. The first point where any process could distinguish the two runs is when a message p_{n-2} sends at t^*_{n-2} does not arrive at p_{n-1} at the same time in D^* it would in D_n, because we extended its delay. Thus, the first time a process can distinguish the two runs is $t^*_{n-2} + d - u = t + (n-2)(d-u) + (d-u) = t + (n-1)(d-u)$. We argue that, for sufficiently large n, this is after p_{n-1}'s *Dequeue* instance returns. That happens at or before $t + (n-1)s + Q$. We thus want $t + (n-1)(d-u) > t + (n-1)s + Q$, which is true iff $(n-1)(d-u) > (n-1)s + Q$. Consider the possible values of s by cases:

- $s = 0$: We want to show that $(n-1)(d-u) > Q$. This is true when $n > \frac{Q}{d-u} + 1$. Since $d > u$, this is true for sufficiently large n. Let N_1 be such that for all $n \geq N_1, n > \frac{Q}{d-u} + 1$.
- $s = D - u$: We want to show that $(n-1)(d-u) > (n-1)(Q-u) + Q$. This is true when $(n-1)d - (n-1)u > nQ - (n-1)u$, or $(n-1)d > nQ$. If we solve for n, we have $n(d-Q) - d > 0$, or $n > \frac{d}{d-Q}$, since $d - Q > 0$. Again, this is true for sufficiently large n, so let N_2 be such that for all $n \geq N_2$, $n > \frac{d}{d-Q}$.

Thus, in runs with sufficiently large n (at least $\max\{N_0, N_1, N_2\}$), p_{n-1} cannot distinguish that it is in D^*, not D_n, until after its *Dequeue* instance has returned. Similarly, no other process can distinguish the runs before its *Dequeue* instance returns, as those returns occur by $t + (i)s + Q \leq t + (n-1)s + Q$ for $0 \leq i < n$, so there is not time for p_{n-1} to inform any other process of the discrepancy since by the time p_{n-1} discovers it, all other processes' *Dequeue* instances have already returned.

Next, we have the case where $u = d$. In this case, observe that $t^*_{n-2} = t+(n-2)(d-u) = t$, so all messages from p_{n-2} to p_{n-1} starting at t have delay d. Thus, p_{n-1} can distinguish the runs at $t + (d - u) = t$, which is before its *Dequeue* instance returns.

Instead, we can use a reduction argument to disprove the existence of an algorithm performing better than our bound. Choose a new message uncertainty $u' = \frac{d+|Dequeue|}{2}$, noting that this gives $0 < u' < d$. Now, since our assumed algorithm correctly implements a multiplicity queue in a system with message delays in the range $[0, d]$ with $|Dequeue| < \min\left\{\frac{3d+2u}{5}, \frac{d}{2} + u\right\} = d$, it must also correctly implement that multiplicity queue in a system with message delays $[d - u', d]$, since any run possible in that system is possible in the system where $d = u$ since the range of possible message delays is completely contained in $[d - u, d]$. It thus implements multiplicity queues in a system with message uncertainty u' with $|Dequeue| < d = 2u' - |Dequeue|$. Then $|Dequeue| < u' < \min\left\{\frac{3d+2u'}{5}, \frac{d}{2} + u'\right\}$ because $d > u'$. But this contradicts the impossibility of such an algorithm as proved in the $u < d$ case above, so our assumed algorithm cannot exist. ◀

Note that the $n \geq \max\{N_0, N_1, N_2\}$ constraint is the only place we need the assumption of uniform algorithms. This shows that our proof applies not only to uniform algorithms, but to any algorithm on at least that many processes. However, we state the result as for uniform algorithms to get a result applicable to any system, as we have not excluded higher performance of non-uniform algorithms in small systems.

Finally, we note that our result is an improvement over the previously best known bound of $|Dequeue| \geq \min\left\{\frac{2d}{3}, \frac{d+u}{2}\right\}$ [13], with the added restriction to uniform algorithms. This claim follows from elementary algebra, as $\frac{d+u}{2} = \frac{d}{2} + \frac{u}{2} < \frac{d}{2} + u$ and $\frac{d+u}{2} \leq \frac{2.5d+2.5u}{5} \leq \frac{3d+2u}{5}$, since $u \leq d$.

▶ **Corollary 9.** *Any uniform, set-linearizable implementation of a multiplicity queue must have* $|Dequeue| \geq \frac{d+u}{2} \geq \min\left\{\frac{2d}{3}, \frac{d+u}{2}\right\}$.

7 Partial Tightness

While it may seem that the $\frac{d}{2} + u$ term in the lower bound is an artifact of our limited proof techniques for lower bounds, and future work may increase the bound to $\frac{3d+2u}{5}$ or better for all values of u, we here outline an algorithm for the special case where $u = 0$ which matches the $\frac{d}{2} + u = \frac{d}{2}$ lower bound, beating $\frac{3d}{5}$. This suggests $\frac{d}{2} + u$ may be somehow fundamental, despite not holding everywhere.

The idea of the algorithm is to have all processes maintain a local copy of the queue, which they update based on messages about operation invocations. For every operation invocation, the invoking process broadcasts operation and arguments immediately, then returns after $d/2$ time. Thus, if two instances are concurrent, neither can learn about the other, since messages take d time to arrive. If they are non-concurrent, then there is more than d time from the invocation of the first instance to the return of the second instance, so at the end of a *Dequeue* instance the invoking process must know about any strictly-preceding instances. Each process will execute every *Enqueue* instance on its local copy, $d/2$ after invocation at the invoking process and d after invocation at every other process, when it receives the message. Non-invoking processes will only locally execute *Dequeue* instances if they have not already seen another *Dequeue* instance concurrent with it. If they have, detected by timestamps, then they have already removed the return value from their local copy, so there is no further work to do. Full pseudocode for the algorithm appears in the appendix, along with the proof of the following theorem.

▶ **Theorem 10.** *If $u = 0$, there is a uniform, set-linearizable implementation of a multiplicity queue with $|Dequeue| = d/2$.*

8 Conclusion

We developed a new combination of shifting and other indistinguishability arguments to prove a larger lower bound of $|Dequeue| \geq \min\left\{\frac{3d+2u}{5}, \frac{d}{2} + u\right\}$ in uniform multiplicity queue implementations. This both improves the state of the art and suggests ways to improve the bound further. For example, strengthening the base case for Lemma 7 in Section 5 should improve the $\frac{3d+2u}{5}$ portion of the lower bound. We hypothesize that this may increase to approach a limit of $|Dequeue| \geq d$ for all non-zero values of u, which seems an intuitive value. If that is true, our tightness result that $|Dequeue| = d/2$ is possible when $u = 0$ is more interesting, as it suggests the bounds may be discontinuous. We continue exploring these bounds to understand multiplicity queues, and then use that understanding to design and understand other data type relaxations.

References

1 Yehuda Afek, Guy Korland, and Eitan Yanovsky. Quasi-linearizability: Relaxed consistency for improved concurrency. In Chenyang Lu, Toshimitsu Masuzawa, and Mohamed Mosbah, editors, *Principles of Distributed Systems - 14th International Conference, OPODIS 2010, Tozeur, Tunisia, December 14-17, 2010. Proceedings*, volume 6490 of *Lecture Notes in Computer Science*, pages 395–410. Springer, 2010. doi:10.1007/978-3-642-17653-1_29.

2 Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, 1995. doi:10.1145/200836.200869.

3 Armando Castañeda and Miguel Piña. Fully read/write fence-free work-stealing with multiplicity. In Seth Gilbert, editor, *35th International Symposium on Distributed Computing, DISC 2021, October 4-8, 2021, Freiburg, Germany (Virtual Conference)*, volume 209 of *LIPIcs*, pages 16:1–16:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.DISC.2021.16.

4 Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. Relaxed queues and stacks from read/write operations. In Quentin Bramas, Rotem Oshman, and Paolo Romano, editors, *24th International Conference on Principles of Distributed Systems, OPODIS 2020, December 14-16, 2020, Strasbourg, France (Virtual Conference)*, volume 184 of *LIPIcs*, pages 13:1–13:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.OPODIS.2020.13.

5 Thomas A. Henzinger, Christoph M. Kirsch, Hannes Payer, Ali Sezgin, and Ana Sokolova. Quantitative relaxation of concurrent data structures. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 317–328. ACM, 2013. doi:10.1145/2429069.2429109.

6 Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991. doi:10.1145/114005.102808.

7 Colette Johnen, Adnane Khattabi, and Alessia Milani. Efficient wait-free queue algorithms with multiple enqueuers and multiple dequeuers. In Eshcar Hillel, Roberto Palmieri, and Etienne Rivière, editors, *26th International Conference on Principles of Distributed Systems, OPODIS 2022, December 13-15, 2022, Brussels, Belgium*, volume 253 of *LIPIcs*, pages 4:1–4:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.OPODIS.2022.4.

8 Pankaj Khanchandani and Roger Wattenhofer. On the importance of synchronization primitives with low consensus numbers. In Paolo Bellavista and Vijay K. Garg, editors, *Proceedings of the 19th International Conference on Distributed Computing and Networking, ICDCN 2018, Varanasi, India, January 4-7, 2018*, pages 18:1–18:10. ACM, 2018. doi:10.1145/3154273.3154306.

9 Martha J. Kosa. Time bounds for strong and hybrid consistency for arbitrary abstract data types. *Chic. J. Theor. Comput. Sci.*, 1999, 1999. URL: http://cjtcs.cs.uchicago.edu/articles/1999/9/contents.html.

10 Jennifer Lundelius and Nancy A. Lynch. An upper and lower bound for clock synchronization. *Information and Control*, 62(2/3):190–204, 1984. doi:10.1016/S0019-9958(84)80033-9.

11 Gil Neiger. Set-linearizability. In James H. Anderson, David Peleg, and Elizabeth Borowsky, editors, *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing, Los Angeles, California, USA, August 14-17, 1994*, page 396. ACM, 1994. doi:10.1145/197917.198176.

12 Nir Shavit and Gadi Taubenfeld. The computability of relaxed data structures: queues and stacks as examples. *Distributed Comput.*, 29(5):395–407, 2016. doi:10.1007/s00446-016-0272-0.

13 Edward Talmage. Lower bounds on message passing implementations of multiplicity-relaxed queues and stacks. In Merav Parter, editor, *Structural Information and Communication Complexity - 29th International Colloquium, SIROCCO 2022, Paderborn, Germany, June 27-29, 2022, Proceedings*, volume 13298 of *Lecture Notes in Computer Science*, pages 253–264. Springer, 2022. doi:10.1007/978-3-031-09993-9_14.

14 Edward Talmage and Jennifer L. Welch. Improving average performance by relaxing distributed data structures. In Fabian Kuhn, editor, *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings*, volume 8784 of *Lecture Notes in Computer Science*, pages 421–438. Springer, 2014. doi:10.1007/978-3-662-45174-8_29.

15 Edward Talmage and Jennifer L. Welch. Relaxed data types as consistency conditions. *Algorithms*, 11(5):61, 2018. doi:10.3390/a11050061.

16 Edward Talmage and Jennifer L. Welch. Anomalies and similarities among consensus numbers of variously-relaxed queues. *Computing*, 101(9):1349–1368, 2019. doi:10.1007/s00607-018-0661-2.

17 Anh Tran and Edward Talmage. Brief announcement: Improved, partially-tight multiplicity queue lower bounds. In Rotem Oshman, Alexandre Nolin, Magnús M. Halldórsson, and Alkida Balliu, editors, *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing, PODC 2023, Orlando, FL, USA, June 19-23, 2023*, pages 370–373. ACM, 2023. doi:10.1145/3583668.3594602.

18 Anh Tran and Edward Talmage. Improved and partially-tight lower bounds for message-passing implementations of multiplicity queues, 2023. doi:10.48550/arXiv.2305.11286.

19 Jiaqi Wang, Edward Talmage, Hyunyoung Lee, and Jennifer L. Welch. Improved time bounds for linearizable implementations of abstract data types. *Inf. Comput.*, 263:1–30, 2018. doi:10.1016/j.ic.2018.08.004.

A Appendix

A.1 Proofs Omitted from Paper Body

▶ **Lemma 11.** *For all $2 \leq k < n$, $D_k = Shift(D_{k-1}, \overrightarrow{s_{k-1}})$, where $\overrightarrow{s_{k-1}}$'s only non-zero component is $-u$ at index $k-1$: $\overrightarrow{s_{k-1}} = \langle 0, \dots, 0, -u, 0, \dots, 0\rangle$.*

Proof. Let k be an arbitrary value with $2 \leq k < n$. Consider what happens when we shift D_{k-1} by $\overrightarrow{s_{k-1}}$. All events at p_{k-1} occur u earlier in real time, so p_{k-1} invokes *Dequeue* at time $t + ((k-1)-1)s + (s+u) - u = t + (k-2)s + s = t + (k-1)s$, which matches the definition of D_k. Let $0 \leq i < k-1 < j < n$. Message delays in D_{k-1} from p_{k-1} to p_i were $d-u$, and from p_i to p_{k-1} were d. In the other direction, messages delays from p_{k-1} to p_j were $d-u$ and from p_j to p_{k-1} were d. When we shift the send and receive events at p_{k-1} earlier, messages from p_{k-1} have a longer delay by u and messages to p_{k-1} have a shorter delay u. We see that this leaves all delays from p_{k-1} to another process at d and all delays to p_{k-1} at $d-u$, which are admissible. Since we only shifted one process, messages between other processes are unchanged.

Finally, we consider clock offsets. c_{k-1} is $\left(\frac{(k-1)-n}{n}\right)u$ in D_{k-1}, and must increase by u to hide the difference in real time when we shift. Thus, in $Shift(D_{k-1}, \overrightarrow{s_{k-1}})$, $c_{i-1} = \left(1 + \frac{(k-1)-n}{n}\right)u = \left(\frac{(k-1)}{n}\right)u$, matching the specification for D_k. ◀

A.2 Partial Tightness: Special Case Upper Bound

The algorithm is event-driven, where each process can react to operation invocations, message receptions, and expiration of local timers it sets. Because $u = 0$, every message takes exactly d time to arrive. Thus, since the algorithm broadcasts every message, when any process receives a message, it knows all other processes receive the same message at the same time. Further, since there is no uncertainty, the maximum clock skew is $(1 - 1/n)0 = 0$, so every process' local clock (read by the function *localClock*()) is equal to real time. We thus let every operation instance take $d/2$ time. By the message delay and operation instance duration, a process learns about an instance at another process before it returns to an instance at itself if and only if that remote instance returned before the local one's invocation, so applying remote operations to the local copy of the structure immediately upon receipt and choosing *Dequeue* return values $d/2$ after invocation together keep the local copies synchronized and choose correct values.

Let R be an arbitrary run of Algorithm 1. Observe that every invocation in R either has a matching response, $d/2$ after invocation. We define a set-linearization of R and prove that π respects real time order and is legal.

▶ **Construction 12.** *Place each Enqueue instance in a singleton set and define the set's timestamp as the pair of the invoking process' local clock read on line 3 plus $d/2$ and the invoking process' id. For each Dequeue return value x, place all Dequeue instances which return x in a set, and define the set's timestamp as the smallest timestamp of any instance in the set, where a Dequeue instance's timestamp is the pair of the local clock read in line 5 and the invoking process' id, with the id breaking ties between clock values. Let π be the sequence of these sets ordered by increasing timestamps (break ties by process id).*

▶ **Lemma 13.** *π respects the order of non-overlapping operation instances.*

Proof. Let op_1 and op_2 be any two non-overlapping operation instances, with op_1 invoked at p_i and returning before op_2's invocation at p_j. Since local clocks are exactly real time, and all instances have duration $d/2$, then op_1's timestamp will be more than $d/2$ smaller than op_2's. Thus, the only way that op_1 would not strictly precede op_2 in π is if they were in the same set, which could happen if they are both *Dequeue* instances which returned the same value x. But in that case, since op_1 returned before op_2's invocation and each of op_1 and op_2 took $d/2$ time between invocation and response, then p_j would receive the message p_i sent on line 5 at op_1's invocation before op_2 returns. This should have removed x from p_j's local copy of the queue, unless there were another element preceding x in p_j's local queue when op_2 returned. By the FIFO ordering of the multiplicity queue, this can only happen if there is a *Dequeue* instance which p_i applied before op_1 returned but p_j did not apply before op_2

Algorithm 1 Set-linearizable implementation of a multiplicity queue with $u = 0$. Code for each p_i.

```
Initially: $localQueue$ is an empty FIFO queue, $mostRecentDequeue = 0$
1: HandleInvocation ENQUEUE($arg$)
2:     send $\langle enq, arg\rangle$ to all other processes
3:     $setTimer(d/2, \langle enq, arg, \langle localClock(), i\rangle, return\rangle)$
4: HandleInvocation DEQUEUE
5:     send $\langle deq, ts = \langle localClock(), i\rangle\rangle$ to all other processes
6:     $setTimer(d/2, \langle deq, ts\rangle)$
7: HandleTimer EXPIRE($\langle enq, arg, ts, return\rangle$)
8:     Generate $Enqueue$ response to user
9:     $setTimer(d/2, \langle enq, arg, apply\rangle)$
10: HandleTimer EXPIRE($\langle enq, arg, apply\rangle$)
11:     $localQueue.enqueue(arg)$
12: HandleTimer EXPIRE($\langle deq, \langle clockVal, i\rangle\rangle$)
13:     Generate $Dequeue$ response to user with return value $localQueue.dequeue()$
14:     $mostRecentDequeue = clockVal$
15: HandleReceive $\langle enq, arg\rangle$
16:     $localQueue.enqueue(arg)$
17: HandleReceive $\langle deq, \langle clockVal, j\rangle\rangle$
18:     if $clockVal > mostRecentDequeue + d/2$ then
19:         $localQueue.dequeue()$
20:         $mostRecentDequeue = clockVal$
```

returned. Any *Dequeue* instance which p_i has applied before op_1 returns was either delivered to p_j at the same time as to p_i, and thus applied to p_j's local copy or invoked at p_i before op_1, but then by the time op_1 returns, by the fact that every *Dequeue* returns $d/2$ time after invocation, p_j would also receive and apply that *Dequeue* instance before op_2's invocation. Thus, there cannot be an element in p_j's local queue preceding x when it applies op_1, and op_2 cannot return x. ◀

▶ **Lemma 14.** *π is legal by the specification of a multiplicity queue.*

Proof. We proceed by induction on σ, a prefix of π. If σ is empty, then it is legal, as the empty sequence is always legal.

Suppose that $\sigma = \rho \cdot S$, where S is a set of operation instances. Assume that ρ is legal. We will show that σ is also legal by cases on S.

If $S = Enqueue(x)$, then σ is necessarily legal, as *Enqueue* does not return a value, so cannot be illegal.

If S is a set of *Dequeue* instances returning x, then we need to argue that the algorithm chose x correctly. Each invoking process chose the oldest value in its local copy of the queue as a return value, in line 13, so we merely need to argue that the local copy of the queue contains the elements enqueued and not dequeued in ρ, in order. Consider the *Dequeue* instance in S with the smallest timestamp, and call it d and its invoking process p_i. When p_i executes line 13 to generate d's return, it will have received every *Enqueue* invocation in ρ, as those were invoked at least $d/2$ before than this *Dequeue*, and added them to its local queue. The order of *Enqueue* instances in ρ matches their timestamp order, which is the order in which they are locally applied, since every process adds each *Enqueue* argument d

time after its invocation. When any other process p_j which has a *Dequeue* instance return the same value as d executes line 13 for that instance, it will have locally applied all *Enqueue* instances p_i has, and possible more. But any additional *Enqueue* instances will have larger timestamps, and thus follow *Enqueue*(x) in π, so would not be the correct return value for this *Dequeue* instance.

Thus, each process chooses x as the oldest-enqueued value in ρ which it has not already removed for another *Dequeue* instance. Such an instance must be in ρ, as another *Dequeue* instance at the same process would have a smaller timestamp and one at another process would not remove a value from the local queue until d after its invocation, which means it would have a smaller timestamp than this *Dequeue* instance which returns x.

Further, each process only removes values from its local queue when there is a *Dequeue* instance returning it. Suppose this were not so. Then some process p_k must have received a *Dequeue* instance which returned y and executed line 19 when it had already removed y from its local queue. But p_k could only remove y when it either returned y to its own *Dequeue* instance or received a message about another *Dequeue* instance. But either of those cases would update *mostRecentDequeue*, so the check on line 18 means that the two *Dequeue* instances which returned y had timestamps more than $d/2$ apart, which implies they were not concurrent, so they could not have returned the same value as that would imply they are in the same set in π, which is not possible by Lemma 13.

Finally, there cannot be a *Dequeue* instance returning x in ρ, as all instances returning x are in the set S. Thus, x is the argument of the first *Enqueue* instance in ρ which is not returned by a *Dequeue* instance in ρ. ◀

▶ **Theorem 15.** *If $u = 0$, Algorithm 1 is a uniform, set-linearizable implementation of a multiplicity queue with $|Dequeue| = d/2$.*

Proof. By Lemma 13, the sequence π we defined in Construction 12 respects the real-time order of non-overlapping instances. Lemma 14 proves that π is legal, so it is a legal set-linearization, proving by construction that Algorithm 1 is a set-linearizable implementation of a multiplicity queue. By lines 6 and 13, every *Dequeue* instance returns $d/2$ time after invocation, so $|Dequeue| = d/2$. Finally, the code for Algorithm 1 does not depend on n, so it is a uniform algorithm. ◀

Since this matches our lower bound of $|Dequeue| \geq \min\left\{\frac{3d+2u}{5}, \frac{d}{2} + u\right\} = \frac{d}{2}$ when $u = 0$, this algorithm is optimal and proves the bound is tight in this case.

Brief Announcement: BatchBoost: Universal Batching for Concurrent Data Structures

Vitaly Aksenov
City, University of London, UK
ITMO University, St. Petersburg, Russia

Michael Anoprenko
Institut Polytechnique de Paris, Palaiseau, France

Alexander Fedorov
IST Austria, Klosterneuburg, Austria

Michael Spear
Lehigh University, Betlehem, PA, USA

Abstract

Batching is a technique that stores multiple keys/values in each node of a data structure. In sequential search data structures, batching reduces latency by reducing the number of cache misses and shortening the chain of pointers to dereference. Applying batching to concurrent data structures is challenging, because it is difficult to maintain the search property and keep contention low in the presence of batching.

In this paper, we present a general methodology for leveraging batching in concurrent search data structures, called BatchBoost. BatchBoost builds a search data structure from distinct "data" and "index" layers. The data layer's purpose is to store a batch of key/value pairs in each of its nodes. The index layer uses an unmodified concurrent search data structure to route operations to a position in the data layer that is "close" to where the corresponding key should exist. The requirements on the index and data layers are low: with minimal effort, we were able to compose three highly scalable concurrent search data structures based on three original data structures as the index layers with a batched version of the Lazy List as the data layer. The resulting BatchBoost data structures provide significant performance improvements over their original counterparts.

2012 ACM Subject Classification Computing methodologies → Concurrent algorithms

Keywords and phrases Concurrency, Synchronization, Locality

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.35

1 Motivation and Background

Batching is an increasingly important technique for maximizing the performance of concurrent data structures. Briefly, batching is the technique by which a linked data structure stores multiple elements in a single data node. The most well-known batched data structure is the B-tree [4], but batching has been applied to a variety of trees [17, 23], lists [5], and skip lists [3, 5]. The benefit of batching is that it co-locates multiple elements in a contiguous region of memory (e.g., a cache line). While batching typically does not improve asymptotic guarantees, it can reduce the total number of cache lines accessed by an operation.

The latency reductions that stem from batching are broadly beneficial. In data structures that provide `scan` operations and `range` queries [2, 3, 8, 12, 24], batching coarsens the granularity of synchronization metadata, so that it can be accessed less frequently. In data structures that use remote direct memory access (RDMA), Non-Uniform Memory Access (NUMA), or non-volatile byte-addressable memory (NVM), batching reduces the number of accesses to a memory that is slower than local DRAM. Batching can also benefit algorithms for GPUs [16] and emerging near-memory computing paradigms [11], where careful consideration of data placement is paramount.

Batching is not without its downsides, for both sequential and concurrent programs. For example, consider an ordered map implemented as a batched linked list (i.e., each node uses a sorted vector to represent a batch of N key/value pairs). While lookup operations within a batch take $O(\log N)$ time, it takes $O(N)$ work to insert or remove an element in a batch, in order to preserve sorting. If instead we used an unsorted batch, each operation would cost $O(N)$, but with lower constants. Similarly, if each batch is protected by a coarse lock, then when keys K_1 and K_2 are stored in the same batch, threads operating on those keys would not be able to proceed in parallel.

While it may seem difficult to find an ideal batch implementation, recent work has shown that it is not too difficult, especially for workloads that deal with large volumes of data and low rates of skew, so long as batch sizes remain modest. Examples of scalable, low-latency batched data structures include maps (e.g., Kiwi [3], CUSL [19], Skip Vector [21], OCC (a, b)-tree [22], Lock-Free B+Tree [6]), and queues [10, 20, 25]. These works tended to treat batching as a first-class design consideration, raising the question of whether it is possible to build a general methodology for adding batching to an existing concurrent data structure. We propose the BatchBoost methodology as a step toward this goal. BatchBoost is designed specifically for ordered maps. It provides programmers with a scalable batched doubly-linked list. The original data structure is then treated as an index to some node in the list. The key innovation is that an out-of-date index will always return a valid node, from which the "correct" node can be found by moving through the links of our doubly-linked links. In this way, BatchBoost lets programmers keep their existing, scalable index, while still benefitting from batching of key/value pairs.

2 Requirements and the BatchBoost Construction

Our goal is to emphasize orthogonality. It should be possible for a programmer to think of a data structure as consisting of an *index layer* and a *data layer*. The data layer should be batched, with as few configuration knobs as possible. The index layer should be decoupled from the data layer, and chosen based on workload and machine characteristics. At any time, it should be trivial to replace the index or data layer with a more suitable data structure, without changing the other layer's implementation.

In BatchBoost, data structure operations always linearize in the data layer. The index layer can be thought of as providing routing "hints." Given relatively straightforward requirements on the data layer, an operation proceeds in three steps. First, it queries the index layer to find a good starting position in the data layer. Second, it operates on the data layer. Finally, it might update the index. A key point is that the index layer need not be kept consistent with the data layer, so long as (1) data layer operations can recover from bad hints, and (2) the index and data layers agree on how to achieve safe memory reclamation.

Listing 1 Composition of index and data layer operations into BatchBoost operations.

```
1 fn lookup(IndexLayer I, Key K) -> Option<V>
2     at = I.findApprox(K)
3     <ret, val, node> = at.lookup(K)
4     if ret == Found: return Some(val)†
5     if ret == NotFound: return None()†
6     if ret == DeletedNode: I.remove(node.key); goto 2
```

```
7
8  fn insert(IndexLayer I, Key K, Value V) -> bool
9      at = I.findApprox(K)
10     <ret, node> = at.insert(K, V)
11     if ret == InsertSuccess: return true†
12     if ret == AlreadyExists: return false†
13     if ret == DeletedNode: I.remove(node.key); goto 9
14     assert(ret == InsertSuccessAndSplit)
15     I.insert(node.key, node)
16     if node.deleted: I.remove(node.key)
17     return true
18
19 fn remove(IndexLayer I, Key K) -> bool
20     at = I.findApprox(K)
21     <ret, node> = at.remove(K)
22     if ret == RemoveSuccess: return true†
23     if ret == NotPresent: return false†
24     if ret == DeletedNode: I.remove(node.key); goto 20
25     assert(ret == RemoveSuccessAndMerge)
26     I.remove(node.key)
27     return true
```

Listing 1 presents a general BatchBoosted data structure. We model the `DataLayer` type as a collection of nodes, each of which stores a tuple $\langle pairs, lower, upper, size, capacity\rangle$, as well as links to other nodes. *pairs* is a collection of *size* key/value pairs ($size \leq capacity$), whose keys are in the range $[lower, upper)$. The range of the `DataLayer` is from $\bot$ to $\top$, which is also the union of all nodes' ranges. We require that from any node, there is a way to reach any other node (perhaps because nodes have predecessor and successor pointers, or because everything is reachable from some sentinel node). We also require that the node include a field indicating if it has been removed from the data layer (a `mark` or `deleted` bit). Each node in `DataLayer` supports three operations with a key argument: 1) `lookup` operation (line 3) traverses the doubly-linked list and returns the node that should contain the key; 2) `insert` operation (line 10) traverses the doubly-linked list, finds the node where the key should be inserted, and inserts there; 3) `remove` operation (line 21) traverses the doubly-linked list, finds the node where the key should be, and removes it from there.

The `IndexLayer` type is an ordered map from keys to `DataLayer::Node` objects. We do not specify its implementation, only that it allows the creation and removal of mappings, and supports some suitable `findApprox(k)` function that returns a value mapped to a key which is likely to be close to `k`. The precision of `findApprox()` does not affect correctness, but the performance of BatchBoosted data structures is likely to correlate with the precision of the index's `findApprox()` implementation.

Initially the data layer contains a single node, which is mapped to the index with key $\bot$. The index may store references to logically deleted nodes; it can also lack references to nodes that are in the data layer. `IndexLayer::findApprox(key)` represents these possibilities: when queried with a key, there is no guarantee that the returned node contains it or even be somewhere close. Note that for an ordered map, `findApprox(key)` can be implemented in many ways, including `ceil(key)` and `floor(key)`.

The index is updated lazily. Insertion of a key/value pair into a node may result in the creation of a new node in the data layer; removal of a pair may result in a node becoming "too small", in which case it can be unlinked once its contents are merged into an adjacent node. These conditions are returned on lines 10 and 21, respectively. If a node becomes deleted between when it is created and when it is added to the index, an `insert` operation is responsible for removing it (line 16). Coupled with standard assumptions about safe memory reclamation, this ensures a node pointed to by the index is still safe to access, even if it has been unlinked from the data layer. Similarly, removal of a merged node from the index layer can delay (line 24), in which case some other thread may remove it (e.g., line 6), and a subsequent insertion can put a different key/node mapping into the index. When this happens, the removal of a valid node is possible. Lines marked with † represent places where an operation may choose to remedy this situation by trying to insert `node` if `node` $\neq$ `at`.

For clarity, the code in Listing 1 skips other optimizations. We do not describe the exact implementation of the data layer because there are lots of them. For example, some data layer implementations may allow `lookup` to succeed even when the node returned by `findApprox` has been unlinked, avoiding the need for line 6.

3 Performance Evaluation

Description. We implemented BatchBoost in C++. We use three non-batched search structures as index layers: Fraser's skip list [13] and trees by Bronson et al. [7] and Natarajan et al. [18]. For all index layers we use the existing `floor` method for `findApprox`. The skip-list code is from SynchroBench [14], the trees are from SetBench [9]. For the data layer, we created a batched, doubly linked list based on the Lazy List [15]. While many configurations of the data layer are possible, we only consider a fixed-capacity array storing its key/value pairs in ascending order. We use epoch-based memory reclamation; threads enter the epoch at the beginning of an operation in Listing 1, and exit the epoch immediately before the operation returns.

All experiments were conducted on a machine with two Intel Xeon Gold 5218 CPUs at 2.30GHz (32 total cores / 64 threads), running Ubuntu 22.04 (Linux Kernel 5.15). We compiled all code with clang 15 (–O3 optimizations). Each data point is the average of five 5-seconds trials. Variance was typically low, and is indicated via error bars.

Experiments are parameterized by lookup ratio R and key range K. Each operation type is chosen randomly and is a lookup with R% probability, with remaining operations split equally between insert and remove. Data structures are pre-filled with 50% of keys, so that the data structure size stays roughly constant. Integer keys are chosen with uniform probability from $[1, K]$.

Sensitivity to Batch Size. The batch size is a critical configuration parameter. If it is too small, batching might increase latency. If it is too large, then contention on batches will be too high, hindering scalability. Figure 1 measures throughput at 32 threads as we vary the batch size ($K = 10^7$). We consider lookup ratios of 34% and 90%. The labels **sl**, **bro**, and **nat** refer to Fraser's skip list [13], Bronson's tree [7], and Natarajan's tree [18], respectively. The **_bb** suffix refers to a BatchBoost data structure composing the corresponding index with our doubly-linked list.

While the results confirm that there is a sensitivity to batch size, the expected performance plateau is surprisingly wide. Thus while there is more than 2× difference between good and bad batch sizes, the exact size does not seem to be particularly significant. We observe that sensitivity is lower than in nonblocking batched data structures [22]. This is due to our use of a lock-based list, which allows in-place modification instead of copy-on-write. Since the drop-off is worse when the batch size gets too large, we conservatively chose a batch size of 100 for all subsequent experiments.

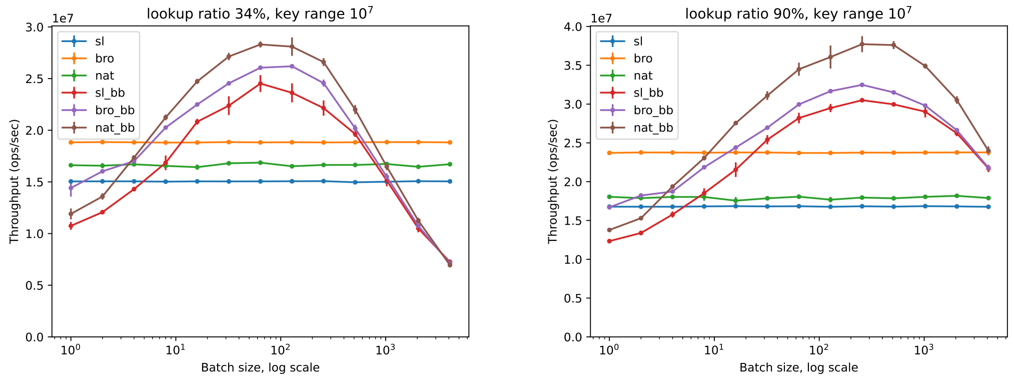


Figure 1 Impact of batch size on throughput at 32 threads.

Table 1 Impact of batch size on cache miss ratio at 16 threads.

	4	64	1024
bro_bb	44.43	30.22	36.68
sl_bb	29.07	22.27	35.58
nat_bb	37.14	36.32	40.71

Using the Linux `perf` tool, we were able to attribute these results directly to a reduction of cache misses. Table 1 shows cache miss ratio against the total number of cache loads for different batch sizes. In effect, BatchBoost shrinks the size of the index, thereby reducing pointer chasing. While the data layer has more cache accesses than a leaf of the unmodified data structure, the increase is less than the savings in the index layer. However, with the increasing batch the ratio of cache misses also increases, thus, we need to choose some ideal batch size.

Throughput and Scalability. Figure 2 measures throughput of our BatchBoost data structures with a fixed batch size as we vary the thread count. BatchBoost consistently improves the performance. The peak speedup depends on workload parameters and varies from $5-10\%$ to almost $2\times$.

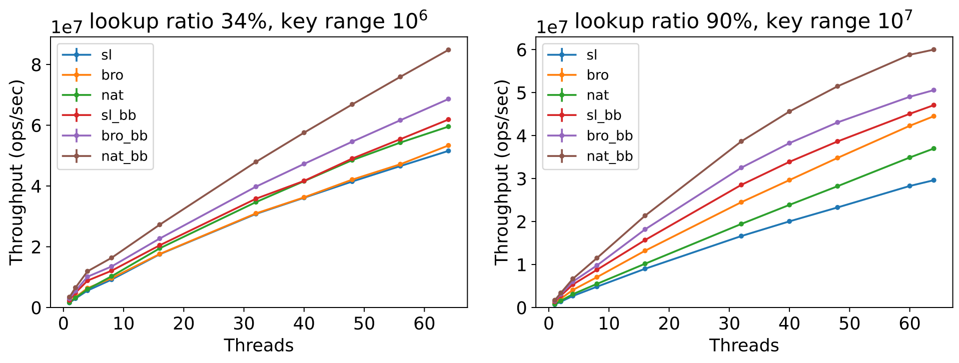


Figure 2 BatchBoost throughput and scalability for varied R and K.

Furthermore, we do not observe significant cache traffic due to contention. By the time threads reach the data layer, the index has dispersed them, reducing the likelihood of contention. Thus as long as the data layer has low latency, the window of contention is low, and threads are not likely to interfere with each other. Additionally, the data layer hides most mutations (insertions and removals) from the index layer. A smaller index, with fewer writes, is more likely to remain resident in most CPUs' caches. In essence, BatchBoost increases the likelihood that the index stays in its common (read-only) case.

4 Conclusions and Future Work

In this paper we introduced the BatchBoost methodology, and demonstrated that it simplifies the creation of scalable data structures with good locality. As discussed in Section 1, batching has broad potential. An important future research direction is to apply our BatchBoost construction in additional domains, as well as on more complex benchmarks. We also intend to compare against other batching techniques. Another important research question pertains to the data layer: We demonstrated that BatchBoost worked well with different index layer implementations, but what about alternate data layer implementations (especially nonblocking)? Further afield, our evaluation showed that BatchBoost amplified the "common case" in the index layer. This may motivate designing new index layers with an explicit and highly optimized `findApprox` operations. For example, we are interested whether we can use a fast sequential index data structure, e.g., Abseil B-trees [1], protected by a scalable readers/writer lock. This could allow concurrent updates and reads, since even under concurrent rebalancing, index lookup operations will give a good enough approximation in our doubly-linked list.

References

1 Abseil b-tree containers. URL: `https://abseil.io/about/design/btree`.

2 Maya Arbel-Raviv and Trevor Brown. Harnessing epoch-based reclamation for efficient range queries. *ACM SIGPLAN Notices*, 53(1):14–27, 2018.

3 Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Guy Golan-Gueta, Eshcar Hillel, Idit Keidar, and Moshe Sulamy. Kiwi: A key-value map for scalable real-time analytics. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 357–369, 2017.

4 Rudolf Bayer and Edward McCreight. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, pages 107–141, 1970.

5 Anastasia Braginsky and Erez Petrank. Locality-conscious lock-free linked lists. In *Distributed Computing and Networking: 12th International Conference, ICDCN 2011, Bangalore, India, January 2-5, 2011. Proceedings 12*, pages 107–118. Springer, 2011.

6 Anastasia Braginsky and Erez Petrank. A Lock-Free B+tree. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*, Pittsburgh, PA, June 2012.

7 Nathan G Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. *ACM Sigplan Notices*, 45(5):257–268, 2010.

8 Trevor Brown and Hillel Avni. Range queries in non-blocking k-ary search trees. In *Principles of Distributed Systems: 16th International Conference, OPODIS 2012, Rome, Italy, December 18-20, 2012. Proceedings 16*, pages 31–45. Springer, 2012.

9 Trevor Brown, Aleksandar Prokopec, and Dan Alistarh. Non-blocking interpolation search trees with doubly-logarithmic running time. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 276–291, 2020.

10 Irina Calciu, Hammurabi Mendes, and Maurice Herlihy. The adaptive priority queue with elimination and combining. In *Distributed Computing: 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings 28*, pages 406–420. Springer, 2014.

11 Jiwon Choe, Andrew Crotty, Tali Moreshet, Maurice Herlihy, and R Iris Bahar. Hybrids: Cache-conscious concurrent data structures for near-memory processing architectures. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 321–332, 2022.

12 Panagiota Fatourou, Elias Papavasileiou, and Eric Ruppert. Persistent non-blocking binary search trees supporting wait-free range queries. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, pages 275–286, 2019.

13 Keir Fraser. Practical lock-freedom. Technical report, University of Cambridge, Computer Laboratory, 2004.

14 Vincent Gramoli. More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–10, 2015.

15 Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N Scherer, and Nir Shavit. A lazy concurrent list-based set algorithm. In *Principles of Distributed Systems: 9th International Conference, OPODIS 2005, Pisa, Italy, December 12-14, 2005, Revised Selected Papers 9*, pages 3–16. Springer, 2006.

16 Kevin Hsieh, Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen W Keckler. Transparent offloading and mapping (tom) enabling programmer-transparent near-data processing in gpu systems. *ACM SIGARCH Computer Architecture News*, 44(3):204–216, 2016.

17 Justin J Levandoski, David B Lomet, and Sudipta Sengupta. The bw-tree: A b-tree for new hardware platforms. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 302–313. IEEE, 2013.

18 Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 317–328, 2014.

19 Kenneth Platz, Neeraj Mittal, and S. Venkatesan. Concurrent Unrolled Skiplist. In *Proceedings of the 39th IEEE International Conference on Distributed Computing Systems*, Dallas, TX, July 2019.

20 Anastasiia Postnikova, Nikita Koval, Giorgi Nadiradze, and Dan Alistarh. Multi-queues can be state-of-the-art priority schedulers. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 353–367, 2022.

21 Matthew Rodriguez, Ahmed Hassan, and Michael Spear. Exploiting locality in scalable ordered maps. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 998–1008. IEEE, 2021.

22 Anubhav Srivastava and Trevor Brown. Elimination (a, b)-trees with fast, durable updates. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 416–430, 2022.

23 Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G Andersen. Building a bw-tree takes more than just buzz words. In *Proceedings of the 2018 International Conference on Management of Data*, pages 473–488, 2018.

24 Yuanhao Wei, Naama Ben-David, Guy E Blelloch, Panagiota Fatourou, Eric Ruppert, and Yihan Sun. Constant-time snapshots with applications to concurrent data structures. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 31–46, 2021.

25 Chaoran Yang and John Mellor-Crummey. A Wait-Free Queue as Fast as Fetch-and-Add. *SIGPLAN Notices*, 51(8), February 2016. doi:10.1145/3016078.2851168.

Brief Announcement: Multi-Valued Connected Consensus: A New Perspective on Crusader Agreement and Adopt-Commit

Hagit Attiya
Department of Computer Science, Technion, Haifa, Israel

Jennifer L. Welch
Department of Computer Science and Engineering, Texas A&M University, College Station, TX, USA

Abstract

Algorithms to solve fault-tolerant consensus in asynchronous systems often rely on primitives such as crusader agreement, adopt-commit, and graded broadcast, which provide weaker agreement properties than consensus. Although these primitives have a similar flavor, they have been defined and implemented separately in ad hoc ways. We propose a new problem called *connected consensus* that has as special cases crusader agreement, adopt-commit, and graded broadcast, and generalizes them to handle multi-valued (non-binary) inputs. The generalization is accomplished by relating the problem to approximate agreement on graphs.

We present three algorithms for multi-valued connected consensus in asynchronous message-passing systems, one tolerating crash failures and two tolerating malicious (unauthenticated Byzantine) failures. We extend the definition of *binding*, a desirable property recently identified as supporting binary consensus algorithms that are correct against adaptive adversaries, to the multi-valued input case and show that all our algorithms satisfy the property. Our crash-resilient algorithm has failure-resilience and time complexity that we show are optimal. When restricted to the case of binary inputs, the algorithm has improved time complexity over prior algorithms. Our two algorithms for malicious failures trade off failure resilience and time complexity. The first algorithm has time complexity that we prove is optimal but worse failure-resilience, while the second has failure-resilience that we prove is optimal but worse time complexity. When restricted to the case of binary inputs, the time complexity (as well as resilience) of the second algorithm matches that of prior algorithms.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases graded broadcast, gradecast, binding, approximate agreement

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.36

Related Version *Full Version*: `https://arxiv.org/abs/2308.04646`

Funding *Hagit Attiya*: partially supported by the Israel Science Foundation (grants 380/18 and 22/1425).

1 Introduction

One way to address the impossibility of solving consensus in asynchronous systems is to employ unreliable *failure detectors* [6]. Several algorithms in this class (e.g., [4, 14]) combine a failure detector with a mechanism for detecting whether processes have reached unanimity, in the form of an *adopt-commit* protocol [21]. In such a protocol, each process starts with a binary input value and returns a pair (v, g) where v is one of the input values and g is either 1 or 2. The process is said to pick v as its output value; furthermore, if $g = 2$, then it *commits* to v, and if $g = 1$, then it *adopts* v. In addition to the standard validity property that the output value is the input of some correct process, an adopt-commit protocol ensures that processes commit to at most one value, and if any process commits to a value, then no process adopts the other value.

Another way to address the impossibility of consensus is to use randomization and provide only probabilistic termination. Some algorithms in this class (e.g., [20]) rely on a mechanism called *crusader agreement* [9]: Roughly, if all processes start with the same value v, they must decide on this value, and otherwise, they may pick an *undecided* value, denoted $\perp$. Other algorithms in this class (e.g., [7]) rely on *graded broadcast* [12], also called *graded crusader agreement*, *graded consensus*, or just *gradecast*. In a sense, graded broadcast is a combination of adopt-commit with crusader agreement: the decisions are either (v, g), where v is a binary value and g is either 1 or 2, or $\perp$ (also denoted $(\perp, 0)$). As in adopt-commit, the requirement is that processes commit to at most one value, but in addition, if any process *adopts* a value, then no process adopts the other value. In a sense, the $\perp$ value allows a separation between adopting one value and adopting a different value.

The relation between crusader agreement, adopt-commit and graded broadcast becomes apparent when they are pictorially represented, as in Figure 1, with the possible decisions represented by vertices on a path. The different "convergence" requirements all boil down to ensuring that processes decide on *the same or adjacent vertices* on the path.

With binary inputs, this description of the problems resembles *approximate agreement on the* $[0, 1]$ *real interval with parameter* ϵ [10]: processes start at the two extreme points of the interval, 0 or 1, and must decide on values that are at most ϵ apart from each other. Decisions must also be valid, i.e., contained in the interval of the inputs.

Indeed, crusader agreement reduces to approximate agreement with $\epsilon = \frac{1}{2}$: Run approximate agreement with your input (0 or 1) to get some output y, then choose the value in $\{0, \frac{1}{2}, 1\}$ that is closest to y (taking the smaller one if there are two such values, e.g., for $y = \frac{1}{4}$). Finally, return $\perp$ if $\frac{1}{2}$ is chosen. (A similar observation is noted in [11, 16].) Likewise, adopt-commit reduces to approximate agreement with $\epsilon = \frac{1}{3}$, and graded consensus to taking $\epsilon = \frac{1}{4}$. This connection makes it clear why *binary* crusader agreement, adopt-commit and graded broadcast can be solved in an asynchronous message-passing system, in the presence of crash and malicious (unauthenticated Byzantine) failures, within a small number of communication rounds.

In some circumstances, agreement must be reached on a *non-binary* value, e.g., the identity of a leader, or the next operation to apply in state machine replication. To handle *multi-valued* inputs, where processes can start with an input from some set V with $|V| \geq 2$, we define a new problem, *connected consensus*. Connected consensus elegantly unifies seemingly-diverse problems, including crusader agreement, graded broadcast, and adopt-commit, and generalizes them to accept multi-valued inputs. The definition takes inspiration from *approximate agreement on graphs* [5], in which each process starts with a vertex of a graph as its input and must decide on a vertex such that all decisions are within distance one of each other and within the convex hull of the inputs.

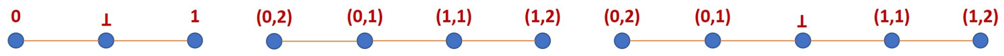


Figure 1 Left: crusader agreement. Center: adopt-commit. Right: graded broadcast.

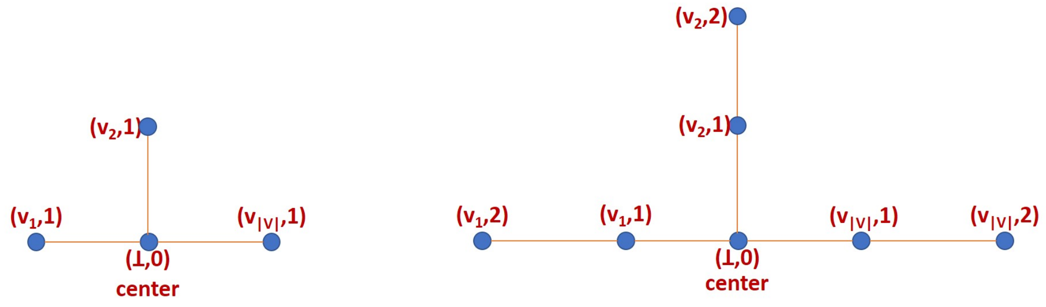


Figure 2 Spider graphs: $R = 1$ (left) and $R = 2$ (right).

2 Connected Consensus and Related Problems

Connected consensus can be viewed as approximate agreement on a restricted class of graphs, called *spider graphs* [15]. These graphs consist of a central clique (which could be a single vertex) to which are attached $|V|$ paths ("branches") of length R, the *refinement parameter*.

More formally, let V be a finite, totally-ordered set of values; assume $\perp \notin V$. Given a positive integer R, let $G_S(V, R)$ be the "spider" graph consisting of a central vertex labeled $(\perp, 0)$ that has $|V|$ paths extending from it, with one path ("branch") associated with each $v \in V$. The path for each v has R vertices on it, not counting $(\perp, 0)$, labeled $(v, 1)$ through (v, R), with (v, R) being the leaf. (See Figure 2.) Given a subset V' of V, we denote by $T(V, R, V')$ the minimal subtree of $G_S(V, R)$ that connects the set of leaves $\{(v, R)|v \in V'\}$; note that when V' is a singleton set $\{v\}$ then $T(V, R, \{v\})$ is the single (leaf) vertex (v, R).

In the *connected consensus problem for V and R*, each process has an input from V. The requirements are:

Termination: Each correct process must decide on a vertex of $G_S(V, R)$, namely, an element of $\{(v, r)|v \in V, 1 \le r \le R\} \cup \{(\perp, 0)\}$.

Validity: Let $I = \{(v, R)|v$ is the input of a (correct)[1] process$\}$. The output of each (correct) process must be a vertex in $T(V, R, I)$. In particular, if all (correct) processes start with the same input v, then (v, R) must be decided.

Agreement: The distance between the vertices labeled by the decisions of all (correct) processes is at most one.

Setting $R = 1$ gives *crusader agreement* [9]. Setting $R = 2$ gives *graded broadcast* [13], also called *adopt-commit-abort* [8].

Recently, the definition of binary (graded) crusader agreement was extended to include a *binding* property [1]: "before the first non-faulty party terminates, there is a value $v \in \{0, 1\}$ such that no non-faulty party can output the value v in any continuation of the execution." That paper demonstrates that this property facilitates the modular design of randomized consensus algorithms that tolerate an *adaptive* adversary. We refer to [1] for an excellent description of the usage, and its pitfalls, of (graded) crusader agreement, together with common coin protocols, in randomized consensus; they show how faster (graded) crusader agreement algorithms lead to faster randomized consensus algorithms.

[1] When "correct" is in parentheses, it only applies for the case of malicious failures.

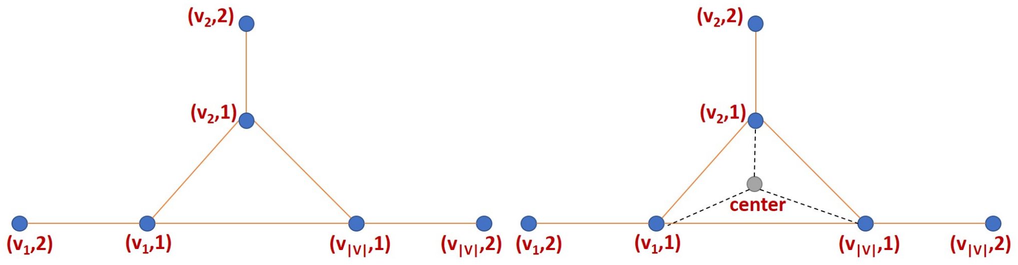


Figure 3 Centerless spider graph with $R = 2$ (left) and its reduction to a (centered) graph (right).

We generalize the binding property to hold for multi-valued inputs: once the first process decides, one value is "locked", so that in all possible extensions, the decisions are on the same branch of the spider graph. Formally:

Binding: In every execution prefix that ends with the first (correct) process deciding, one value is "locked", meaning that in every extension of the execution prefix, the decision of every (correct) process must be on the same branch of the spider graph.

If the first decision is not $(\bot, 0)$, then this condition follows from Agreement. More interestingly, if the first decision is $(\bot, 0)$, then there are many choices as to which branch is locked but the choice must be the same in every extension. Note that when $|V| = 2$, our definition is equivalent to the original one [1], but for larger V, our definition is stronger – the original definition only excludes one value, leaving $|V| - 1$ possible decision values, while ours excludes $|V| - 1$ values, leaving only one possible decision value.

When $R = 1$, there are only two vertices on any given branch of the spider graph, $(v, 1)$ and $(\bot, 0)$. Thus, the Binding property implies the Agreement property. If $R = 2$, though, the Binding property only restricts the branch of the spider graph on which decisions can be made; both $(\bot, 0)$ and $(v, 2)$ are on the same branch, but Agreement does not permit them to both be decided.

Recall that in *adopt-commit* [14, 21], processes return a pair (v, g) where v is one of the input values and g is either 1 (adopt) or 2 (commit). Thus, there is no analog of the "center" vertex. We model this with a *centerless* spider graph (see left side of Figure 3). Here, $G_S(V, R)$ is the graph consisting of a clique on the vertices $(v, 1)$ for all $v \in V$, each with a path extending from it, with $R - 1$ vertices on it, not counting $(\bot, 0)$, labeled $(v, 2)$ through (v, R), with (v, R) being the leaf. Decisions must satisfy Termination, Validity and Agreement as specified for the variant with a center. Since the graph has no center, binding cannot be defined; indeed, when a process returns $(v, 1)$, other processes might return $(v', 1)$, for $v \neq v'$.

The centerless problem can be reduced to the centered problem with the same refinement parameter: Call the algorithm for the centered problem with your input u. If the return value is (v, g) with $g > 0$, then decide this value for the centerless problem; when the return value is $(\bot, 0)$, decide $(u, 1)$ for the centerless problem. (See right side of Figure 3.)

In the *vacillate-adopt-commit* (VAC) problem [2], the possible output values are (v,commit), (v,adopt), and (v,vacillate), where v is any value. If any output is (v,commit), then every other output is either (v,commit) or (v,adopt), for the same v. Furthermore, if there is no commit output and there is at least one (v,adopt) output, then every other output is either (v,adopt), with the same value v, or (w,vacillate), where w can be any value. VAC corresponds to a centerless spider graph with refinement parameter $R = 3$. However, a closer look at the usage of VAC suggests that the return value of vacillate is irrelevant and the problem could be represented as a centered spider graph with $R = 2$.

3 New Algorithms for Connected Consensus

With these definitions at hand, we turn to designing algorithms for connected consensus in asynchronous message-passing systems that tolerate crash or malicious failures. There is an algorithm for approximate agreement on general graphs in the presence of malicious failures [19]. However, it requires exponential local computation and does not satisfy the Binding property. We are interested in special-case spider graphs, as described above; furthermore, we focus on the cases when the refinement parameter R equals either 1 or 2, which captures the applications of interest. Thus we exploit opportunities for optimizations to obtain better algorithms.

For *communication complexity*, we count the maximum, over all executions, of the number of messages sent by all the (correct) processes. We adopt the definition in [3] for *time complexity* in an asynchronous message-passing system. We start by defining a timed execution as an execution in which nondecreasing nonnegative integers ("times") are assigned to the events, with no two events by the same process having the same time. For each timed execution, we consider the prefix ending when the last correct process decides, and then scale the times so that the *maximum time* that elapses between the sending and receipt of any message between correct processes is 1. We define the time complexity as the maximum, over all such scaled timed execution prefixes, of the time assigned to the last event. (For simplicity, we assume all processes start at time 0.) This definition of time complexity is analogous to that in [17, 18], which measures the length of the longest sequence of causally related messages.

We present an algorithm for $R = 1$ and $R = 2$ with the Binding property that tolerates crash failures assuming $n > 2f$, where n is the number of processes and f is the maximum number of faulty processes, which is optimal. Its time complexity is R and its message complexity is $O(n^2)$. The message complexity is optimal and the time complexity is optimal for reasonable resiliencies. The best previous algorithms, in [1], have slightly worse time complexity: 2 for $R = 1$ (crusader agreement) and 3 for $R = 2$ (graded crusader agreement). Furthermore, both of these previous algorithms are for the binary case ($|V| = 2$) only.

For malicious failures, we first present a simple algorithm with Binding for $R = 1$ and $R = 2$, that assumes $n > 5f$. Like the crash-tolerant algorithm, its time complexity is R and its message complexity is $O(n^2)$. The message complexity is optimal and the time complexity is optimal for reasonable resiliencies. Both this algorithm and our crash-tolerant one derive the Binding property from the inputs of the processes. That is, the assignment of input values to the processes uniquely determines which non-$\bot$ value, if any, can be decided in any execution with that input assignment. The fact that Binding is determined solely by the inputs is conducive to the development of simple and efficient algorithms. However, we show that in the presence of malicious failures Binding cannot be determined solely by the inputs when $n < 5f$, even if faulty processes do not equivocate.

Our main algorithmic contribution is a connected consensus algorithm for $R = 1$ and $R = 2$ with Binding that tolerates f malicious failures, where $n > 3f$. A simple proof shows that this is the optimal resilience. Its time complexity is 5 for $R = 1$ and 7 for $R = 2$, and its message complexity is $O(|V| \cdot n^2)$, where V is the set of input values. The message complexity can be reduced to $O(n^2)$, at the cost of increasing the time complexity by 2, using techniques of [18].

The upper bounds of 5 and 7 on the time complexity are tight for our algorithm, as shown by giving a concrete execution. The execution uses $V = \{0, 1\}$ and it is also an execution of the crusader agreement algorithm in [1], implying that the tight time complexity of the

Table 1 Summary of connected consensus algorithms for $R = 1$ (crusader agreement) and $R = 2$ (graded broadcast) with input set V; all algorithms satisfy Binding.

failure type	crash		malicious			
algorithm	this paper	[1] ($\|V\| = 2$)	this paper	this paper	this paper + [18]	[1] ($\|V\| = 2$)
resilience	$n > 2f$	$n > 2f$	$n > 5f$	$n > 3f$	$n > 3f$	$n > 3f$
messages	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(\|V\| \cdot n^2)$	$O(n^2)$	$O(n^2)$
time $R = 1$	1	2	1	5	7	5
time $R = 2$	2	3	2	7	9	7

latter algorithm is also 5, and that of the graded broadcast algorithm in [1] is 7. This is in contrast to the *round complexities* of 4 and 6 calculated in [1] for their algorithms. The round complexity counts the number of broadcasts performed by an algorithm. However, in their algorithms (as well as in ours), waiting conditions are imposed before performing the next broadcast. If the condition is simply to receive enough messages from the previous broadcast, then at most one time unit elapses per broadcast. But when there is an additional condition, then the condition may take more than one time unit to become true.

4 Discussion

This paper presents the *connected consensus* problem. A numeric *refinement* parameter, R, allows connected consensus to generalize a number of primitives used to solve consensus, including crusader agreement, graded broadcast, and adopt-commit. The problem can be reduced to real-valued approximate agreement when the input set is binary and and to approximate agreement on a specific class of *spider* graphs for multi-valued input sets (with two or more inputs). We define the *Binding property* for the multi-valued case, which previously was only defined for the binary case.

We design efficient message-passing algorithms for connected consensus when R is 1 (corresponding to crusader agreement) or 2 (corresponding to graded broadcast), in the presence of crash and malicious failures, for arbitrarily large input sets. The algorithms are modular in that the $R = 2$ case is obtained by appending more communication exchanges to the $R = 1$ case. (Table 1 summarizes our algorithms and relates them to prior work.)

Our algorithm for crash failures has optimal resilience and message complexity. Its time complexity is optimal for reasonable resiliencies and improves on the best previously known algorithms, which only handled binary inputs. We provide two algorithms for malicious failures: One algorithm has time complexity 1 or 2 (for $R = 1$ or $R = 2$) and sends $O(n^2)$ messages, but requires $n > 5f$. The other algorithm only requires $n > 3f$, but has time complexity 5 or 7 (for $R = 1$ or $R = 2$) and sends $O(|V| \cdot n^2)$ messages. This is the same performance as the algorithms in [1] which are only for the case when $|V| = 2$.

An intriguing open question is whether there is some measure, perhaps time, in which solving connected consensus without Binding is more efficient than solving it with Binding?

References

1 Ittai Abraham, Naama Ben-David, and Sravya Yandamuri. Efficient and adaptively secure asynchronous binary agreement via binding crusader agreement. In *41st ACM Symposium on Principles of Distributed Computing*, pages 381–391, 2022.

2 Yehuda Afek, James Aspnes, Edo Cohen, and Danny Vainstein. Brief announcement: Object oriented consensus. In *36th ACM Symposium on Principles of Distributed Computing*, pages 367–369, 2017. Full version in https://www.cs.yale.edu/homes/aspnes/papers/vac-abstract.html.

3 Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. McGraw-Hill Publishing Company, 1st edition, 1998.

4 Zohir Bouzid, Achour Mostefaoui, and Michel Raynal. Minimal synchrony for Byzantine consensus. In *34th ACM Symposium on Principles of Distributed Computing*, pages 461–470, 2015.

5 Armando Castañeda, Sergio Rajsbaum, and Matthieu Roy. Convergence and covering on graphs for wait-free robots. *Journal of the Brazilian Computer Society*, 24:1–15, 2018.

6 Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

7 Giovanni Deligios, Martin Hirt, and Chen-Da Liu-Zhang. Round-efficient Byzantine agreement and multi-party computation with asynchronous fallback. In *19th International Conference on Theory of Cryptography, TCC*, pages 623–653, 2021.

8 Carole Delporte-Gallet, Hugues Fauconnier, and Michel Raynal. On the weakest information on failures to solve mutual exclusion and consensus in asynchronous crash-prone read/write systems. *Journal of Parallel and Distributed Computing*, 153:110–118, 2021.

9 Danny Dolev. The Byzantine generals strike again. *Journal of Algorithms*, 3(1):14–30, 1982.

10 Danny Dolev, Nancy A. Lynch, Shlomit S. Pinter, Eugene W. Stark, and William E. Weihl. Reaching approximate agreement in the presence of faults. *J. ACM*, 33(3):499–516, 1986.

11 Alan David Fekete. Asymptotically optimal algorithms for approximate agreement. *Distributed Computing*, 4:9–29, 1990.

12 Paul Feldman and Silvio Micali. Optimal algorithms for Byzantine agreement. In *12th Annual ACM Symposium on Theory of Computing*, pages 148–161, 1988.

13 Pesech Feldman and Silvio Micali. An optimal probabilistic protocol for synchronous Byzantine agreement. *SIAM J. Comput.*, 26(4):873–933, 1997.

14 Eli Gafni. Round-by-round fault detectors: unifying synchrony and asynchrony. In *17th ACM Symposium on Principles of Distributed Computing*, pages 143–152, 1998.

15 Manfred Koebe. On a new class of intersection graphs. In *Annals of Discrete Mathematics*, volume 51, pages 141–143. Elsevier, 1992.

16 Stephen R Mahaney and Fred B Schneider. Inexact agreement: Accuracy, precision, and graceful degradation. In *4th ACM Symposium on Principles of Distributed Computing*, pages 237–249, 1985.

17 Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. Signature-free asynchronous binary Byzantine consensus with $t < n/3$, $O(n^2)$ messages, and $O(1)$ expected time. *J. ACM*, 62(4):31:1–31:21, 2015.

18 Achour Mostéfaoui and Michel Raynal. Signature-free asynchronous Byzantine systems: from multivalued to binary consensus with $t < n/3$, $O(n^2)$ messages, and constant time. *Acta Informatica*, 54(5):501–520, 2017.

19 Thomas Nowak and Joel Rybicki. Byzantine approximate agreement on graphs. In *33rd International Symposium on Distributed Computing*, pages 29:1–29:17, 2019.

20 Sam Toueg. Randomized Byzantine agreements. In *3rd ACM Symposium on Principles of Distributed Computing*, pages 163–178, 1984.

21 Jiong Yang, Gil Neiger, and Eli Gafni. Structured derivations of consensus algorithms for failure detectors. In *17th ACM Symposium on Principles of Distributed Computing*, pages 297–306, 1998.

Brief Announcement: Relations Between Space-Bounded and Adaptive Massively Parallel Computations

Michael Chen
Iowa State University, Ames, IA, USA

A. Pavan
Iowa State University, Ames, IA, USA

N. V. Vinodchandran
University of Nebraska–Lincoln, NE, USA

Abstract

In this work, we study the class of problems solvable by (deterministic) Adaptive Massively Parallel Computations in constant rounds from a computational complexity theory perspective. A language L is in the class AMPC^0 if, for every $\varepsilon > 0$, there is a deterministic AMPC algorithm running in constant rounds with a polynomial number of processors, where the local memory of each machine $s = O(N^\varepsilon)$. We prove that the space-bounded complexity class $\mathsf{ReachUL}$ is a *proper subclass* of AMPC^0. The complexity class $\mathsf{ReachUL}$ lies between the well-known space-bounded complexity classes Deterministic Logspace (DLOG) and Nondeterministic Logspace (NLOG). In contrast, we establish that it is unlikely that PSPACE admits AMPC algorithms, even with polynomially many rounds. We also establish that showing PSPACE is a subclass of *nonuniform*-AMPC with polynomially many rounds leads to a significant separation result in complexity theory, namely PSPACE is a proper subclass of $\mathsf{EXP}^{\Sigma_2^{\mathsf{P}}}$.

2012 ACM Subject Classification Computing methodologies → Massively parallel algorithms; Theory of computation → Complexity classes

Keywords and phrases Massively Parallel Computation, AMPC, Complexity Classes, LogSpace, NL, PSPACE

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.37

Funding This work is supported in part by NSF grants 1934884, 2130536, and 2130608.

Acknowledgements The authors thank Sriram Pemmaraju and Meena Mahajan for their helpful discussions. We thank anonymous reviewers for their valuable comments and pointers to some critical references.

1 Introduction

The *Massively Parallel Computation* (MPC) model is widely accepted as the standard theoretical model for distributed computation frameworks such as MapReduce, Spark, Hadoop, FlumeJava, Beame, Pregel, and Gigraph [7, 9]. It was defined in [5], and it captures computation on large data: data is adversarially distributed to processors, and each processor has local memory $s = O(N^\varepsilon)$ $(0 < \varepsilon < 1$ where N is the input size. Computation occurs in rounds, and in each round, every machine performs computation based on its local data and then communicates with other machines with the constraint that the amount of communication by a process is equal to that of its local memory s. A salient feature of the MPC model is that no computational restriction is placed on the processor, except that each processor has local memory s, and a key objective is to minimize the number of rounds.

Ideally, one would like to design an algorithm with constant rounds with a small number of processors. The MPC model has been extensively studied in the context of designing algorithms as well as its relationship with complexity classes [2, 3, 4, 7, 5, 6, 16].

Recent work of [7] introduced an adaptive extension of the MPC model called *Adaptive Massively Parallel Computation* model (AMPC). In the AMPC model, the processors communicate via a shared memory called Distributed Data Stores (DDS) by reading from and writing to the DDS. In a single round, a machine can adaptively query the DDS to obtain s words and write at most s words, and as in the case of MPC, s is $O(N^{\varepsilon})$. In [7], authors designed a constant round randomized AMPC algorithm for 1v2-Cycle as well as a few other graph problems.

In this work, we report progress on the power and limitation of the AMPC model from a computational complexity theory viewpoint. Towards this, we define a robust complexity class which we denote by AMPC^0. A language L is in AMPC^0 if for every ε there is a constant round (depending on ε) AMPC algorithm with $P = p(N)$ many processors (where $p(\cdot)$ is a polynomial) each with $s = O(N^{\varepsilon})$ memory. We define a similar complexity class $\mathsf{AMPC}^{\mathsf{poly}}$ where the number of rounds is polynomial. We study the relationship of these AMPC complexity classes with respect to the standard space-bounded complexity classes DLOG, NLOG, and PSPACE. The starting point of our work is that the ideas from the randomized AMPC algorithm for 1v2-Cycle from [7] can be used to show that the complexity class DLOG is a subset of (uniform) AMPC^0. Motivated by this, we explore whether NLOG is a subset of AMPC^0. We make progress toward this question by studying a complexity class $\mathsf{ReachUL}$ [8, 1, 11]. This is a natural complexity class that lies between DLOG and NLOG and has been studied earlier in the context of designing space-efficient algorithms for reachability that beat the Savitch's bound [1]. We prove that $\mathsf{ReachUL}$ is a subset of (uniform) AMPC^0, More interestingly, we show that $\mathsf{ReachUL}$ is a *proper subset* of (uniform) AMPC^0. On the contrary, we observe that it is unlikely that the whole of PSPACE (or even NP) can be solved even in $\mathsf{AMPC}^{\mathsf{poly}}$. This is because every language that admits (uniform) $\mathsf{AMPC}^{\mathsf{poly}}$ algorithm can be solved in subexponential time. Since we do not believe that PSPACE can be solved in subexponential time, we obtain that it is unlikely that $\mathsf{PSPACE} \subseteq \mathsf{AMPC}^{\mathsf{poly}}$. We also consider the limitation of nonuniform $\mathsf{AMPC}^{\mathsf{poly}}$. We unconditionally show that there exist languages in $\mathsf{E}^{\Sigma_2^{\mathsf{P}}}$ that are not in $\mathsf{AMPC}^{\mathsf{poly}}$. We note that the work reported in [15] also considered the relations of complexity classes such as DLOG and NLOG to MPC model.

▶ Remark. In an algorithmic setting, it is typically desired that the total memory of an AMPC algorithm $P \cdot s$ to be $N \cdot \operatorname{poly}\log(N)$. However, to define a robust complexity class (closed under reductions), we allow P to be polynomial and require that for every $0 < \varepsilon < 1$, there is an algorithm with s local memory per processor.

2 Preliminaries

We now give the formal description of the AMPC model [7, 9]. Let $p(\cdot)$, $s(\cdot)$ and $r(\cdot)$ are functions from $\mathbb{N}$ to $\mathbb{N}$. An $\mathsf{AMPC}[p(N), s(N), r(N)]$ algorithm for length N, is a collection of processors $M_{i,j}$, $1 \leq i \leq p(N)$ and $1 \leq j \leq r(N)$ where each processor has a memory bound of $s(N)$. In addition to the processors there is a collection of *Distributed Data Stores* (DDS) denoted by $\mathcal{D}_0, \mathcal{D}_1, \mathcal{D}_2, \ldots \mathcal{D}_{r(N)}$. For each DDS, the data is stored in a bit addressable manner (as done in [9]) i.e., a collection of key-value pairs in the form of $(i, i^{th}$ bit of DDS). The input string $x = x_1 \ldots x_N$ is stored in $\mathcal{D}_0$ in the form of $\{(i, x_i)\}_{i=1}^{N}$. The computation occurs in rounds. The processors $M_{i,j}$, $1 \leq i \leq p(N)$ participate in the jth round. In the jth round, each of these processors is allowed to make $s(N)$ adaptive queries to read from

$\mathcal{D}_{j-1}$ and each processor is allowed to can write up to $s(N)$ bits to $\mathcal{D}_j$. The computation stops after $r(N)$ rounds, and we say that the algorithm accepts string x if the value of key 1 in $\mathcal{D}_{r(N)}$ is 1.

Inherently this is a nonuniform model of computation. A language L is in the class (nonuniform) AMPC^0 if for every $0 < \varepsilon < 1$, there exists a polynomial $p(\cdot)$, and a constant $r = r(N) > 0$ such that for every input length $N \geq 0$, there is a $\mathsf{AMPC}[p(N), N^{\varepsilon}, r]$ algorithm that accepts L on strings at length N. We define the uniform AMPC model. This definition is similar to the uniform MRC model as defined in [10]. For an algorithm P, we use $P_{i,j}$ to denote a processor whose behavior is the same as P on inputs i and j. A language L is in the class (uniform) AMPC^0 if for every $\varepsilon > 0$, there exists a polynomial $p(\cdot)$ and a constant $r = r(N)$, and a logspace bounded algorithm U that on input 1^N outputs the code of a processor P with the following properties: the processors $P_{i,j}$ $1, \leq i \leq p(N)$, $1 \leq j \leq r$ constitute a $\mathsf{AMPC}[p(N), N^{\varepsilon}, r]$ algorithm that accepts L at strings of length N. Analogously we define uniform and nonuniform versions of the class $\mathsf{AMPC}^{\mathsf{poly}}$ where the number of rounds is allowed to be a polynomial. In the rest of the document, we write AMPC^0 to denote (uniform) AMPC^0.

We use DLOG (resp. PSPACE) to denote the class of languages accepted by deterministic logspace (resp. polynomial-space) Turing machines. The complexity class E is the class of languages that are accepted by deterministic $2^{O(N)}$-time bounded machines, and Σ_2^{P} denote the class of languages in the second level of the polynomial-hierarchy. A language L is in the class SubEXP, if for every $\varepsilon > 0$, there is a $O(2^{N^{\varepsilon}})$-time bounded machine that accepts L. A language L is in NC^1 if L can be decided by a family of circuits $\{C_N\}_{N\in\mathbb{N}}$ where C_N has poly(N) size and $O(\log N)$ depth.

▶ **Definition 1** ([8, 1]). *A nondeterministic machine is called reach-unambiguous if for every configuration C, there is at most one path from the start configuration to C. The class* $\mathsf{ReachUL}$ *is the class of languages that are accepted by $O(\log N)$-space-bounded reach-unambiguous machines.*

▶ **Definition 2** (Reach-Unambiguous). Reach-Unambiguous *is the language consisting of tuples $\langle G, a, b\rangle$ such that (1) $G = (V, E)$ is a directed graph, (2) for all $u \in V$ there exists at most 1 directed path from a to u and, (3) there exists a directed path from a to b.*

It is known that Reach-Unambiguous is complete for $\mathsf{ReachUL}$ with respect to logspace reductions [13, 8, 1].

3 Results

3.1 ReachUL in AMPC^0

We show that $\mathsf{ReachUL}$ is a proper subset of AMPC^0. We start with the following theorem.

▶ **Theorem 3.** $\mathsf{DLOG} \subsetneq \mathsf{AMPC}^0$. *That is,* DLOG *is a proper subset of* AMPC^0.

▶ **Corollary 4.** AMPC^0 *is closed under logspace reductions.*

Inclusion of Theorem 3 follows from [7]. The authors showed a randomized constant round AMPC algorithm for the DLOG-complete problem, 1v2-Cycle. Their algorithm can be modified to obtain a deterministic algorithm by allowing for $O(N)$ processors and using $O(N^{1+\varepsilon})$ total memory for any ε. The strictness of inclusion follows from Theorem 8 which we prove.

Let $\langle G, a, b\rangle$ be an input instance of REACH-UNAMBIGUOUS where $G = (V, E)$ is a reach-unambiguous graph such that $V = \{v_1, \ldots, v_N\}$ and $a, b \in V$. Without loss of generality, we assume that the out-degree of each vertex is at most 2. For $u \in V$ and $s \in \mathbb{N}$, define $T_s(u)$ to be the tree resulting from a *Breadth First Search* (BFS) starting from u in G upto s nodes such that no node is partially visited, i.e., either all the children of any vertex are in the tree, or none of them are. We shall call every node other than u in $T_s(u)$ a descendent of u. The main ingredient in our proof is Algorithm 1 that constructs a compressed version of $T_s(u)$. This algorithm is based on the tree contraction idea [1]. We note that recently tree contraction has been studied in the context AMPC in [12]. For any graph G, we often overload the notation G to also refer to the vertex set of the graph.

▶ **Definition 5.** *Let $u \in V$, and v be a descendant of u in $T_s(u)$. v is said to be an intermediate vertex for $T_s(u)$ if there exists an edge $(v, w) \in E$ such that $w \notin T_s(u)$. Define $I_s(u) \subseteq T_s(u)$ as the set of vertices that are intermediate for $T_s(u)$. We say that $T_s(u)$ is* complete *if $I_s(u) = \emptyset$, otherwise $T_s(u)$ is incomplete.*

Intermediate vertices capture the idea of vertices that can still be explored. If v is an intermediate vertex for $T_s(u)$, that means v can still be further explored. But due to the BFS parameter s, it could not explore v any further. We shall assume for simplicity of the analysis that if a tree $T_s(u)$ is incomplete, the tree has exactly $s+1$ vertices (in general, such a tree could have either s or $s+1$ vertices). Thus the condition $|T_s(u)| < s+1$ denotes the condition that $T_s(u)$ is complete.

Algorithm 1 Construct Algorithm.

```
Function Construct$(u, s)$:
    Compute $T_s(u)$ using at most $O(s)$ queries.
    if $b \in T_s(u)$ then
        // $b$ can be reached from $u$ within $s$ queries
        $T'_s(u) \leftarrow (\{b\}, \emptyset)$
    else if $|T_s(u)| < s+1$ then
        // $b$ cannot be reached from $u$
        $T'_s(u) \leftarrow (\{u\}, \emptyset)$
    else
        // $b$ cannot be reached from $u$ within $s$ queries, need to explore
        Compute $I_s(u)$ using at most $O(s)$ queries
        $T'_s(u) \leftarrow$ A complete binary tree whose leaves are exactly $I_s(u)$
    Write $T'_s(u)$ to the DDS
```

Let $T'_s(u)$ be the output of *Construct*(u, s) in Algorithm 1. $T'_s(u)$ is a contracted version of $T_s(u)$. If $b \in T_s(u)$ or $|T_s(u)| < s+1$, the search from u is completed, and we can contract the tree to a single node. Otherwise, the tree is contracted to a complete binary tree whose leaves are $I_s(u)$, which are precisely the candidates that can lead to b. Locally, it is possible that $T'_s(u)$ does not contract. Claim 6 shows that globally the contraction will occur.

Define the tree T' generated by starting with $T'_s(a)$, and recursively substituting every leaf $l \in T'$ with $T'_s(l)$. Continuing the process until substituting leaves does not change the tree. This graph has the property that $\langle T', a, b\rangle \in$ REACH-UNAMBIGUOUS $\iff$ $\langle G, a, b\rangle \in$ REACH-UNAMBIGUOUS since the only vertices that remain are those vertices that have the potential to reach b. For an AMPC model, T' need not be explicitly constructed since each tree is locally computed and is then updated in the DDS. We shall now show that the graph size reduces by a factor of $s/2$, which is sufficient to get the algorithm to halt in constant rounds by setting $s = O(N^\varepsilon)$.

▷ Claim 6. $|T'| \leq 2N/s$

Given $u \in V$, for analysis' sake construct $H_s(u)$ by making every descendant in $T_s(u)$ a child of u, i.e. $H_s(u)$ is a re-arranged version of $T_s(u)$ such that edges go from u to the descendants of u in $T_s(u)$.

We now construct an s-ary tree, H, such that it is always full (vertices either have out degree 0 or s). Start with $H_s(a)$, then for every leaf l whose parent is p such that $l \in I_s(p)$ substitute l with $H_s(l)$ if $T_s(p) = s + 1$. If the BFS search was incomplete, substitute it with b if $b \in H_s(l)$, otherwise, do nothing. Repeat the process until no more substitutions can be done. This construction leads to a full s-ary tree H, such that $|H| \leq N$. H represents a BFS traversal done in "batches" of size s, where only intermediate nodes are substituted with another s-ary tree. Exploring non-intermediate nodes would be redundant. Let i denote the number of internal nodes of this s-ary tree.

Proof of Claim 6. Since H is a full s-ary tree, $i = |H|/s \leq N/s$. And H is essentially a rearrangement of T' such that internal vertices in H correspond to intermediate vertices in T'. However, due to line 9 of Algorithm 1, we may be adding more vertices, but however, it is no more than twice. i.e. we have $|T'| \leq 2i$. Therefore we have $|T'| \leq 2N/s$ ◁

▶ **Theorem 7.** Reach-Unambiguous $\in$ AMPC^0

Proof. Let $\langle G, a, b\rangle$ be a problem instance of Reach-Unambiguous with $G = (V, E)$ such that $V = \{v_1, \ldots, v_N\}$. Fix $\varepsilon \in (0, 1)$. Define the AMPC algorithm with $s = O(N^\varepsilon)$ local memory. Assign $G_1 \leftarrow G$ and $R \leftarrow O(1/\varepsilon)$, for $i = 1, \ldots, R$ rounds do the following, for each $v \in G_i$, a machine executes *Construct*(v, s) for G_i to get a new graph T' as described earlier. Assign $G_{i+1} \leftarrow T'$. After the rounds are complete, $|G_R| = \frac{N}{(s/2)^R} = O(1)$. Then a single machine can perform normal reachability on G_R, accepting the input if and only if there is a path from a to b. ◀

▶ **Theorem 8.** *For every $c \in \mathbb{N}$. There exists a problem in* AMPC^0 *that is not in* DSPACE$(\log^c N)$

Proof. Let $A \in$ DSPACE(N^2) but $A \notin$ DSPACE(N). We know such a problem exists due to the space hierarchy theorem. Consider a padded language B defined as $B = \{\langle x, y\rangle \mid x \in A, |x| = M, |y| = 2^{M^{1/c}} - M\}$.

We claim that $B \notin$ DSPACE$(\log^c N)$. Assume by contradiction, $B \in$ DSPACE$(\log^c N)$. We show that in that case, $A \in$ DSPACE(N). Let x be an input instance of A of length M; we shall solve it by reducing it to an instance of B, by generating $z = \langle x, y\rangle$, where $y = 0^{2^{M^{1/c}} - M}$, we have $|z| = N = 2^{M^{1/c}}$. The instance z need not be explicitly stored; every bit can be computed on the fly. Thus the space used is $O(M)$. Then use algorithm for B to solve z using space $O(\log^c N) = O(\log^c 2^{M^{1/c}}) = O(M)$. Thus $A \in$ DSPACE(N) a contradiction. Therefore, $B \notin$ DSPACE$(\log^c N)$.

Now we show that B is in AMPC^0. Let $z = \langle x, y\rangle$ be a problem instance of B of length N. The crucial point to note is that the membership of z in B depends only on x, which has length $O(\log^c N)$. If x does not have this length, we can safely reject it. Fix an arbitrary $\varepsilon \in (0, 1)$. Consider the AMPC algorithm with just one machine and one round. Let $z = \langle x, y\rangle$ be an input of length N. The machine $\mathcal{M}$ reads x which has length $M = O(\log^c N) \subseteq O(N^\varepsilon)$, then checks if $x \in A$ using space $O(M^2) = O(\log^{2c} N) \subseteq O(N^{\varepsilon})$, via our assumption that $A \in \mathsf{DSPACE}(N^2)$. If $x \in A$ then $\mathcal{M}$ accepts otherwise rejects. Thus we have exhibited a language B such that $B \in \mathsf{AMPC}^0$ but $B \notin \mathsf{DSPACE}(\log^c N)$. ◀

▶ **Theorem 9.** $\mathsf{ReachUL} \subsetneq \mathsf{AMPC}^0$. *That is,* ReachUL *is a proper subset of* AMPC^0.

Proof. The containment follows since REACH-UNAMBIGUOUS is complete for ReachUL under logspace reductions and by Corollary 4, AMPC^0 is closed under logspace reductions. The strict containment follows from the fact that $\mathsf{ReachUL} \subseteq \mathsf{NLOG} \subseteq \mathsf{DSPACE}(\log^2 N)$. Hence by Theorem 8, there is a language in AMPC^0 that is not in ReachUL. ◀

3.2 Limitations

This section discusses the limitations of the AMPC model in relation to well-known complexity classes.

Uniform Model. Since each processor in each round has a memory bound of $O(N^{\varepsilon})$, the number of configurations of each processor is $\mathsf{poly}(2^{N^{\varepsilon}})$ and hence runs in $O(2^{N^{\varepsilon'}})$ for some $0 < \varepsilon' < 1$ (since the processors are halting). Thus it is clear that uniform $\mathsf{AMPC}^{\mathsf{poly}}$ is in SubEXP. Thus by time-hierarchy theorem, there is a language in EXP that is not in $\mathsf{AMPC}^{\mathsf{poly}}$. This also establishes that it is unlikely that PSPACE is in $\mathsf{AMPC}^{\mathsf{poly}}$ as this will imply PSPACE is a subset of SubEXP. Moreover, no NP-complete problem (under logspace reduction) is in $\mathsf{AMPC}^{\mathsf{poly}}$ unless $\mathsf{NP} \subseteq \mathsf{SubEXP}$.

Non-uniform Model. In the case of non-uniform AMPC computations, we can argue that any language accepted by a polynomial round AMPC algorithm can be simulated by a Boolean circuit of size $\mathsf{poly}(2^{N^{\varepsilon}})$. This is because every bit computed by a processor is a decision tree of size $O(2^{N^{\varepsilon}})$ and hence has a Boolean circuit of size $O(2^{N^{\varepsilon}})$. Since the number of bits written by all machines overall (polynomial) rounds is bounded by a polynomial, the size of the Boolean circuit simulating the whole computation is $\mathsf{poly}(2^{N^{\varepsilon}})$. It is known that there is a language L in $\mathsf{E}^{\Sigma_2^{\mathsf{P}}}$ that has maximum circuit complexity [14], it follows that L is not in non-uniform $\mathsf{AMPC}^{\mathsf{poly}}$. This lower bound establishes that showing PSPACE is in non-uniform $\mathsf{AMPC}^{\mathsf{poly}}$ is difficult as this will imply an unknown complexity theory separation that PSPACE is a proper subset of $\mathsf{EXP}^{\Sigma_2^{\mathsf{P}}}$.

References

1 E. Allender and K.-J. Lange. RUSPACE($\log n$) $\subseteq$ DSPACE($\log^2 n/\log\log n$). *Theory of Computing Systems*, 31(5):539–550, October 1998. doi:10.1007/s002240000102.

2 Alexandr Andoni, Zhao Song, Clifford Stein, Zhengyu Wang, and Peilin Zhong. Parallel graph connectivity in log diameter rounds. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, 2018.

3 Alexandr Andoni, Clifford Stein, and Peilin Zhong. Log Diameter Rounds Algorithms for 2-Vertex and 2-Edge Connectivity. In *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*, Leibniz International Proceedings in Informatics (LIPIcs), pages 14:1–14:16, 2019.

4 Sepehr Assadi, Xiaorui Sun, and Omri Weinstein. Massively parallel algorithms for finding well-connected components in sparse graphs. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, 2019. doi:10.1145/3293611.3331596.

5 Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. *J. ACM*, 64(6), October 2017. doi:10.1145/3125644.

6 Soheil Behnezhad, Laxman Dhulipala, Hossein Esfandiari, Jakub Lacki, and Vahab Mirrokni. Near-optimal massively parallel graph connectivity. In *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1615–1636, 2019. `doi:10.1109/FOCS.2019.00095`.

7 Soheil Behnezhad, Laxman Dhulipala, Hossein Esfandiari, Jakub Lacki, Vahab Mirrokni, and Warren Schudy. Massively parallel computation via remote memory access. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '19, pages 59–68, New York, NY, USA, 2019. Association for Computing Machinery. `doi:10.1145/3323165.3323208`.

8 Gerhard Buntrock, Birgit Jenner, Klaus-Jörn Lange, and Peter Rossmanith. Unambiguity and fewness for logarithmic space. In L. Budach, editor, *Fundamentals of Computation Theory*, pages 168–179, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

9 Moses Charikar, Weiyun Ma, and Li-Yang Tan. Unconditional lower bounds for adaptive massively parallel computation. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '20, pages 141–151, New York, NY, USA, 2020. Association for Computing Machinery. `doi:10.1145/3350755.3400230`.

10 Benjamin Fish, Jeremy Kun, Ádám D. Lelkes, Lev Reyzin, and György Turán. On the computational complexity of mapreduce. In Yoram Moses, editor, *Distributed Computing*, pages 1–15, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

11 Brady Garvin, Derrick Stolee, Raghunath Tewari, and N. V. Vinodchandran. Reachfewl = reachul. *computational complexity*, 23(1):85–98, March 2014. `doi:10.1007/s00037-012-0050-8`.

12 MohammadTaghi Hajiaghayi, Marina Knittel, Hamed Saleh, and Hsin-Hao Su. Adaptive Massively Parallel Constant-Round Tree Contraction. In *13th Innovations in Theoretical Computer Science Conference (ITCS 2022)*, Leibniz International Proceedings in Informatics (LIPIcs), pages 83:1–83:23, 2022. `doi:10.4230/LIPIcs.ITCS.2022.83`.

13 Klaus-Jörn Lange. An unambiguous class possessing a complete set. In Rüdiger Reischuk and Michel Morvan, editors, *STACS 97*, pages 339–350, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

14 Peter Bro Miltersen, N. V. Vinodchandran, and Osamu Watanabe. Super-polynomial versus half-exponential circuit size in the exponential hierarchy. In Takao Asano, Hiroshi Imai, D. T. Lee, Shin-Ichi Nakano, and Takeshi Tokuyama, editors, *Computing and Combinatorics, 5th Annual International Conference, COCOON '99, Proceedings*, Lecture Notes in Computer Science, 1999.

15 Danupon Nanongkai and Michele Scquizzato. Equivalence classes and conditional hardness in massively parallel computations. *Distributed Computing*, 35(2):165–183, April 2022. `doi:10.1007/s00446-021-00418-2`.

16 Tim Roughgarden, Sergei Vassilvitskii, and Joshua R. Wang. Shuffles and circuits (on lower bounds for modern parallel computation). *J. ACM*, 65(6), November 2018. `doi:10.1145/3232536`.

Brief Announcement: On Implementing Wear Leveling in Persistent Synchronization Structures

Jakeb Chouinard
Department of Mechanical and Mechatronics Engineering, University of Waterloo, Canada

Kush Kansara
Department of Electrical and Computer Engineering, University of Waterloo, Canada

Xialin Liu
Department of Electrical and Computer Engineering, University of Waterloo, Canada

Nihal Potdar
Department of Electrical and Computer Engineering, University of Waterloo, Canada

Wojciech Golab
Department of Electrical and Computer Engineering, University of Waterloo, Canada

Abstract

The last decade has witnessed an explosion of research on persistent memory, which combines the low access latency of dynamic random access memory (DRAM) with the durability of secondary storage. Intel's implementation of persistent memory, called Optane, comes close to realizing the game-changing potential of persistent memory in terms of performance; however, it also suffers from limited endurance and relies on a proprietary wear leveling mechanism to mitigate memory cell wear-out. The traditional embedded approach to wear leveling, in which the storage device itself maps logical addresses to physical addresses, can be fast and energy-efficient, but it is also relatively inflexible and can lead to missed opportunities for optimization. An alternative school of thought, exemplified by "open channel" solid state drives (SSDs), delegates responsibility for wear leveling to software, where it can be tailored to specific applications. In this research, we consider a hypothetical hardware platform where the same paradigm is applied to the persistent memory device, and ask how the wear leveling mechanism can be co-designed with synchronization structures that generate highly skewed memory access patterns. Building on the recent work of Liu and Golab, we implement an improved wear leveling atomic counter by leveraging hardware transactional memory in a novel way. Our solution is close to optimal with respect to both space complexity and measured performance.

2012 ACM Subject Classification Theory of computation → Shared memory algorithms

Keywords and phrases persistent memory, transactional memory, wear leveling, atomic counter, concurrency, fault tolerance, theory

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.38

Funding This research was supported by an Ontario Early Researcher Award, a Google Faculty Research Award, as well as the Natural Sciences and Engineering Research Council (NSERC) of Canada.

Acknowledgements We thank the anonymous reviewers for their helpful feedback on this work and their insightful suggestions regarding future research directions.

1 Introduction

The last decade has witnessed an explosion of research on persistent memory. Research activities in this area are primarily driven by the performance benefits of persistent memory, which behaves like dynamic random access memory (DRAM) with respect to access latency and yet provides the durability of secondary storage. Thus, persistent memory can be used directly to store application state during a computation, and its use opens the door to recovering such state efficiently from the memory device after a power failure or system crash. Intel's implementation of persistent memory, called Optane, comes close to realizing the game-changing potential of persistent memory in terms of performance, but it suffers from limited endurance, meaning that the memory cells tend to wear out in response to repeated overwriting [3]. To prevent irrecoverable data loss during the product warranty period, Optane persistent memory modules use a proprietary wear leveling mechanism that remaps logical memory addresses to physical addresses somewhat similarly to a flash translation layer (FTL) in a solid state drive (SSD).

The traditional embedded approach to wear leveling, in which the storage device itself internally performs address remapping, can be fast and energy-efficient. However, this one-size-fits-all solution is relatively inflexible, and it can lead to missed opportunities for optimization when the workload (i.e., data access pattern) generated by an application deviates from the one anticipated by the hardware designer. An alternative school of thought, exemplified by "open channel" solid state drives (SSDs) [8], addresses this inherent limitation by delegating responsibility for wear leveling to software, where it can be tailored more effectively to specific applications. In this research, we consider a hypothetical hardware platform where the same paradigm is applied to the persistent memory device, and ask how the wear leveling mechanism can be co-designed with persistent data structures.

The case for application-managed wear leveling in the context of persistent memory is especially interesting due to stringent design constraints that limit the solution space. Specifically, the physical form factor of the persistent memory module limits how much logical-to-physical (L2P) address translation data can be stored on the device, and the translation algorithm must be extremely fast to enable memory access at DRAM-like latency. To operate within these constraints, the wear leveling algorithm cannot accurately account for the number of write cycles applied to every individual memory word, and so it must operate at a coarser granularity. Details of Intel's Optane persistent memory are not well documented, but it is known that these memory modules are internally organized into blocks of 256 bytes [5, 7]. Because of this, we speculate that wear leveling state is likely tracked on a per-block basis (or even more coarsely). While such a block-based wear leveling scheme could work effectively for workloads dominated by sequential writing, like storing append-only logs, it can lead to severe resource under-utilization in a scenario where a single memory word is repeatedly overwritten. This limitation is particularly relevant for a byte-addressable "write-in-place" storage medium like Intel's Optane memory, whereas a flash-based SSD's entire data block must always be erased before it can be overwritten.

This paper focuses on software-managed wear leveling for synchronization structures, such as shared counters, which generate precisely the kind of skewed memory access pattern that can delude a general-purpose embedded wear leveling solution. Building on the foundations established by Liu and Golab [6], we propose a novel software implementation of an atomic counter that internally harnesses together multiple words of persistent memory to distribute wear. Our implementation uses transactional memory in a new way and vastly outperforms Liu and Golab's algorithm, which is based on ordinary Compare-And-Swap.

2 The Wear Leveling Problem

For the purposes of this paper, wear leveling is the abstract problem of implementing a concurrent object that maintains correctness across many state changes while using base objects that may lose their correctness after relatively few state changes. Liu and Golab [6] formalized this notion as the following *endurance* property, where T can denote a constant or a function of some model-specific parameters like the number of concurrent threads:

▶ **Definition 1.** *An object has* endurance T *if it maintains its safety and liveness properties in all executions where at most* T *updates (i.e., operations other than reads) are invoked on the object, but not in some execution where* $T+1$ *updates are invoked.*

In general, the endurance of an implemented object (e.g., one that is strictly linearizable [1, 2] and lock-free) is limited by the endurance of the base objects from which the implemented object is constructed. We focus in this work on *endurance-oblivious* [6] implementations that treat the endurance of the base objects as an unknown.

3 The Transactional Counter Algorithm

Building on the work of Liu and Golab [6], we seek improved implementations of the atomic counter, also known as a *Fetch-And-Increment* object, in the system-wide crash-recover failure model with persistent main memory and a volatile cache. The abstract state of a counter object is an integer, typically initialized to zero. The object supports a single operation that retrieves the current value of the counter and also increases the value by one. As an example, a strictly linearizable [1, 2] lock-free implementation of an atomic counter using the `FetchAndIncrement` instruction is presented in Figure 1. Although the implementation lacks wear leveling, it illustrates our syntax conventions and the correct use of persistence instructions to manage the volatile cache.[1] The linearization point of the Increment operation is the first (process-initiated or environment-initiated) flush step that persists either the value of the counter established at line 1 or a larger value.

Persistent shared variables:
- B: base object supporting `FetchAndIncrement` operation, initially 0

Procedure Increment().

```
1 ret := FetchAndIncrement(&B)
2 Persist(&B)
3 return ret
```

Figure 1 Baseline counter implementation.

Following [6], we partition the state of the counter across a collection of k base objects $B_0 \dots B_{k-1}$ such that the value of the implemented object equals the sum of the values of the base objects. In theory, the endurance of the implemented object can be increased by a factor of k as long as two conditions are met. First, each implemented operation must correctly compute the fetched value, which amounts to obtaining a snapshot of the states of the base objects that appears to be atomic with respect to the increment. Second, each implemented operation must not only spread out wear evenly across the base objects, but also limit the number of updates applied to the base objects to avoid undesirable *write amplification*. Ideally, each implemented operation would increment only a single base object.

Both challenges are addressed by maintaining a particular state invariant over the base objects, as illustrated by way of example in Figure 2. The pattern is that the i'th Increment operation on the implemented object (counting starting at zero) updates base object number $\lfloor i/m \rfloor \bmod k$, where m is a parameter we call the *bin size*. Intuitively, m increments are

[1] The ampersand symbol (&) means "address of" as in C/C++. `Persist` represents a process-initiated flush step on a base object that is assumed to fit inside a single cache line. It can be implemented on the Intel platform using the function `pmem_persist` in the Persistent Memory Development Kit [9].

state of implemented object	base object B_0	base object B_1	base object B_2	...	base object B_{k-1}
0	0	0	0	...	0
1	1	0	0	...	0
2	2	0	0	...	0
m	m	0	0	...	0
$m+1$	m	1	0	...	0
$2m$	m	m	0	...	0
km	m	m	m	...	m
$km+1$	$m+1$	m	m	...	m
$km+m+1$	$2m$	$m+1$	m	...	m

Figure 2 State representation of wear leveling counter for bin size m and k base objects.

applied to base object B_0, then m to B_1, ..., m to B_{k-1}, then m more to B_0, etc., in round-robin fashion. As long as the base objects have endurance T such that T is a multiple of m, the implemented object can count up to kT, which improves on the baseline technique from Figure 1 by a factor of k. Not only does this strategy amplify endurance, but it also expands the counter's domain of values by the same factor k.

The central technical challenge in maintaining our state invariant under concurrent access is to apply the correct state transition to the correct base object each time the implemented counter is accessed. This is a non-trivial task since the correct base object and state transition depend on the position of an Increment operation in the linearization order, which is not known to processes ahead of time. Liu and Golab [6] solved the problem for bin size $m = 1$ using an algorithm based on the `CompareAndSwap` instruction, which is lock-free but inefficient under high contention. We improve upon this preliminary design by replacing the `CompareAndSwap` instruction with `FetchAndIncrement`. The immediate problem this strategy presents is that incrementing a base object unconditionally can increase its value beyond the threshold permitted by the invariant presented earlier in Figure 2, which we rely on crucially to correctly compute the response of an Increment operation. For example, if more than m processes attempt to increment base object B_0 starting from the initial state, then the final value of B_0 will exceed the value of B_1 by more than m, which violates the invariant. We address this problem by encapsulating the `FetchAndIncrement` instruction in a hardware transaction, and aborting the transaction whenever the invariant is violated. Secondly, we optimize the selection of the base object by introducing static variables that allow processes to remember which object was last accessed.[2] Assuming that processes access the counter frequently and that the bin size m is large relative to the number of processes, this second optimization mostly avoids the costly linear search in Liu and Golab's algorithm.

We present pseudo-code for the algorithm in Figure 3, which borrows syntax from the GCC transactional memory intrinsics [4]. At the beginning, process p computes the boundary between the current bin and the next bin at line 4 based on its recollection of the current bin obtained from static variable bin_p. The transaction then starts at line 6 inside the outer while loop, and its current status is determined at line 7 using the `_xbegin` intrinsic. For the reader unfamiliar with the GCC transactional intrinsics, the algorithm should be interpreted

[2] A static variable retains its value across calls to Increment. Our algorithms do not persist such variables, and function correctly (albeit more slowly) even if the variables hold stale values after a crash.

Persistent shared variables:

- $B[0..(k-1)]$: array of base objects, each element initially 0

Private static variables:

- $index_p$: integer in the interval $[0, k)$, initially 0
- bin_p: integer ≥ 0, initially 0

Private variables:

- $limit_p, prev_p, status_p, bumped_p, temp_p$: integers

Procedure Increment().

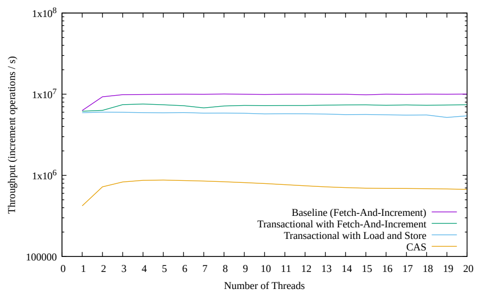
```
4  $limit_p := (bin_p + 1) \times m$
5  while true do
6      $status_p :=$ _xbegin()
7      if $status_p =$ _XBEGIN_STARTED then
8          $prev_p :=$ FetchAndIncrement(&$B[index_p]$)
9          if $prev_p \geq limit_p$ then
10             _xabort(_ABORT_BIN_EXCEEDED)
11         else
12             _xend()
13             Persist(&$B[index_p]$)
14             return $prev_p + m \times (bin_p \times (k-1) + index_p)$
15     else if _XABORT_CODE($status_p$) = _ABORT_BIN_EXCEEDED then
16         $bumped_p :=$ false
17         while true do
18             $temp_p := B[index_p]$
19             if $temp_p < limit_p$ then
20                 break
21             else
22                 $index_p := (index_p + 1) \bmod k$
23                 if $index_p = 0$ then
24                     $bin_p := \lfloor temp_p / m \rfloor$
25                     $limit_p := (bin_p + 1) \times m$
26                 $bumped_p :=$ true
27         if $bumped_p$ then
28             Persist(&$B[(index_p + k - 1) \bmod k]$)
```

Figure 3 Endurance-oblivious counter implementation using hardware transactions and FetchAndIncrement. Pseudo-code shown for process p, k base objects, and bin size m.

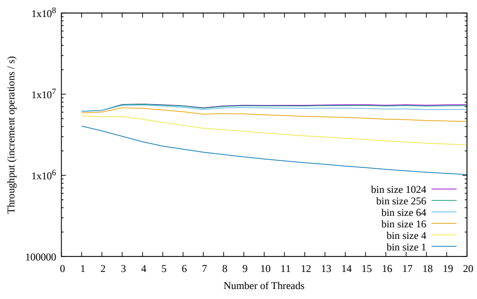
as returning a successful status (_XBEGIN_STARTED) when line 7 is first executed in an iteration of the outer while loop. It then proceeds with the FetchAndIncrement instruction at line 8 and continues onward to the commit point at line 12 and beyond (lines 13–14), unless the transaction aborts. The latter can occur due to an explicit abort at line 10 via the _xabort intrinsic or due to a spontaneous abort, and in either case, the algorithm is rolled back to line 6 where _xbegin is re-executed and returns a special status code different from _XBEGIN_STARTED. The _XABORT_CODE intrinsic (a GCC macro) at line 15 determines the user-defined code (if any) passed to _xabort at line 10. If the transaction aborted spontaneously then it is restarted at the next iteration of the while loop, otherwise the fallback execution path at lines 16–28 is executed to adjust the values of the static variables $index_p$ and bin_p, and another transaction is attempted.

4 Experiments

We implemented a collection of wear leveling counters in C++ and evaluated their performance on a 20-core Intel Xeon Gold 6230 platform with Optane persistent memory. The Intel Persistent Memory Development Kit (PMDK) [9] was used to access the Optane memory using memory-mapped files. The `Persist` operation featured in our pseudo-code was implemented using the `pmem_persist` function in the PMDK, which internally performs a cache line write-back (clwb) and store fence. Intel's Restricted Transactional Memory (RTM) was accessed using GCC intrinsics [4], which we explained earlier in Section 3. Persistent memory and hardware transactions are typically not used together as the transactions do not guarantee failure-atomicity, and persistence instructions inside a transaction can cause an abort on some platforms. However, the transaction used in our algorithm circumvents these drawbacks by accessing only a single memory word and persisting after committing.



(a) Comparison of counter implementations with cache line write-backs and store fences.



(b) Sensitivity of transactional `FetchAndIncrement` implementation to the bin size parameter.

Figure 4 Scalability experiments.

Figure 4a presents an experimental comparison of the baseline algorithm from Figure 1, our transactional counter algorithm from Figure 3, an alternative implementation of our algorithm that uses load and store instead of `FetchAndIncrement`, and the Liu-Golab algorithm (denoted CAS). The bin size parameter (m in Section 3) was 1 for the baseline and Liu-Golab algorithms, and 1024 for the two transactional algorithms. We observe that the transactional `FetchAndIncrement`-based algorithm is roughly 1.5× slower than the baseline, which lacks wear leveling, and outperforms the alternative transactional algorithm by roughly 2×. It also outperforms Liu-Golab by roughly 15×. Next, we consider the effect of the bin size on performance in Figure 4b, and find that a bin size of $256 \leq m \leq 1024$ works well.

References

1 Marcos K. Aguilera and S. Frølund. Strict linearizability and the power of aborting. Technical Report HPL-2003-241, Hewlett-Packard Labs, 2003.

2 Ryan Berryhill, Wojciech Golab, and Mahesh Tripunitara. Robust shared objects for non-volatile main memory. In *Proc. of the 19th International Conference on Principles of Distributed Systems (OPODIS)*, pages 20:1–20:17, 2016.

3 Frank Hady. Intel Optane technology delivers new levels of endurance, 2019. URL: `https://www.intel.com/content/www/us/en/architecture-and-technology/optane-technology/delivering-new-levels-of-endurance-article-brief.html`.

4 Free Software Foundation Inc. Transactional memory intrinsics. [last accessed 5/01/2023]. URL: `https://gcc.gnu.org/onlinedocs/gcc/x86-transactional-memory-intrinsics.html`.

5 Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the Intel Optane DC persistent memory module. *CoRR*, abs/1903.05714, 2019. `arXiv:1903.05714`.

6 Xialin Liu and Wojciech Golab. Brief announcement: Towards a theory of wear leveling in persistent data structures. In *Proc. of the 41st ACM Symposium on Principles of Distributed Computing (PODC)*, pages 220–223, 2022.

7 Ivy Bo Peng, Maya B. Gokhale, and Eric W. Green. System evaluation of the Intel Optane byte-addressable NVM. In *Proc. of the International Symposium on Memory Systems (MEMSYS)*, pages 304–315, 2019.

8 Ivan Luiz Picoli, Niclas Hedam, Philippe Bonnet, and Pinar Tözün. Open-channel SSD (what is it good for). In *In Proc. of the 10th Conference on Innovative Data Systems Research (CIDR)*, 2020.

9 Andy Rudoff and the Intel PMDK Team. Persistent memory development kit. [last accessed 5/01/2023]. URL: `https://pmem.io/pmdk/`.

Brief Announcement: Subquadratic Multivalued Asynchronous Byzantine Agreement WHP

Shir Cohen ✉
Technion, Haifa, Israel

Idit Keidar ✉
Technion, Haifa, Israel

Abstract

There have been several reductions from multivalued consensus to binary consensus over the past 20 years. To the best of our knowledge, none of them solved it for Byzantine asynchronous settings. In this short paper, we close this gap. Moreover, we do so in subquadratic communication, using newly developed subquadratic binary Byzantine Agreement techniques.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Theory of computation → Cryptographic primitives; Mathematics of computing → Probabilistic algorithms

Keywords and phrases Byzantine agreement, subquadratic communication, fault tolerance in distributed systems

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.39

Related Version *Full Version*: `https://arxiv.org/abs/2308.02927` [3]

Funding *Shir Cohen*: Supported by the Adams Fellowship Program of the Israel Academy of Sciences and Humanities.

1 Introduction

Byzantine Agreement (BA) is a well-studied problem where a set of correct processes have input values and aim to agree on a common decision despite the presence of malicious ones. This problem was first defined over 40 years ago [8]. However, in the past decade BA gained a renewed interest due to the emergence of blockchains as a new distributed and decentralized tool. Moreover, the scale of the systems in which this problem is solved is much larger than in the past. As a result, there is a constant effort to find new techniques that will enable the reduction of communication complexity of BA solutions.

A significant improvement in BA scalability was enabled by subquadratic solutions, circumventing Dolev and Reischuk's renown lower bound of $\Omega(n^2)$ messages [5]. This was done by King and Saia [7] in the synchronous model and later by Algorand [6] (first in the synchronous model and then with eventual synchrony). In their work, Algorand presented a validated committee sampling primitive, based on the idea of cryptographic sortition using *verifiable random functions* (VRF) [9]. This primitive allows different subsets of processes to execute different parts of the BA protocol. Each committee is used for sending exactly one protocol message and messages are sent only by committee members, thus reducing the communication cost.

In this paper, we tackle the asynchronous model, which best describes real-life settings. Importantly, subquadratic asynchronous BA was first introduced not so long ago by Cohen et. al [4] and Blum et. al [2]. The limitation of these results is that both solve a binary BA, where the inputs and outputs are in $\{0, 1\}$. We extend the binary results to multivalued BA, which is more suitable in real-world systems. Nowadays, perhaps the most extensive use of BA solution appears in blockchains to agree on the next block. These blocks carry multiple transactions (as well as additional metadata), which is clearly not a binary value. We note that a similar extension for multivalued consensus was presented in asynchrony by Mostefaoui, Raynal, and Tronel [11] but it cannot handle Byzantine failures. Another reduction by Turpin and Coan [12] is able to handle Byzantine failures, but only in a synchronous model. Despite not solving the multivalued case in the Byzantine asynchronous model, both of them have quadratic word complexity (Following the standard complexity notions [1, 10]).

We consider a system with a static set of n processes and an adversary, in the so-called "permissioned" setting, where the ids of all processes are well-known. The adversary may adaptively corrupt up to $f = (\frac{1}{3} - \epsilon)n$ processes in the course of a run, where $\frac{1}{2\ln n} < \epsilon < \frac{1}{3}$. In addition, we assume a trusted *public key infrastructure* (PKI) that allows us to use *verifiable random functions* (VRFs) [9]. Finally, we assume that once the adversary takes over a process, it cannot "'front run" messages that that process had already sent when it was correct, causing the correct messages to be supplanted. This assumption can be replaced if one assumes an erasure model as used in [2, 6]. That is, using a separate key to encrypt each message, and deleting the secret key immediately thereafter.

Let us first examine the "straightforward" (yet faulty) reduction from multivalued BA to binary BA, both satisfying the *strong unanimity* validity property. This property states that if all correct processes have the same input value, then this must be the decision they all output. To solve the multivalued BA, any process interprets its input as a binary string, and then all processes participate in a sequence of binary BA instances. The input for the i^{th} instance by process p is the i^{th} digit in the binary representation. It is easy to see, that if all correct processes share the same input value, they start each instance with the same binary value and by validity agree upon it. Hence, by the end of the last BA instance, they all reach the same (input) decision. Otherwise, they can agree on some arbitrary common value.

Unfortunately, the simple binary-to-multivalued reduction does not work when applied with existing asynchronous subquadratic solutions. Assume that in the multivalued version of BA, values are taken from a finite domain $\mathcal{V}$. If the size of $\mathcal{V}$ is in $O(n)$, then to represent the input value as a binary string we need $O(\log n)$ bits, and the same number of BA instances. Although this number keeps the overall complexity subquadratic, it breaks the probability arguments made in the existing solutions. Briefly, both works take advantage of logarithmic subsets of processes that drive the protocol progress. These so-called committees are elected uniformly such that with high probability (WHP) it contains "enough" correct processes, and not "too many" Byzantine ones. Since, WHP, both algorithms complete in a constant number of rounds, their safety and liveness are guaranteed WHP. However, once we apply these techniques and make the probability arguments more than a constant number of times (as we would if we were to apply the reduction), the high probability does not remain high at all.

To overcome this challenge, we take a different approach. We generalize the method in [12] to work with asynchronous committee sampling and solve for *weak unanimity* validity. Our algorithm requires only two additional committees, compared to any binary BA algorithm. Finally, we present the first multivalued BA with a word complexity of $\widetilde{O}(n)$.

Validated Committee Sampling

Using VRFs, it is possible to implement *validated committee sampling*, which is a primitive that allows processes to elect committees without communication and later prove their election. It provides every process p_i with a private function $sample_i(s, \lambda)$, which gets a string s and a threshold $1 \leq \lambda \leq n$ and returns a tuple $\langle v_i, \sigma_i \rangle$, where $v_i \in \{true, false\}$ and σ_i is a proof that $v_i = sample_i(s, \lambda)$. If $v_i = true$ we say that p_i is *sampled* to the committee for s and λ. The primitive ensures that p_i is sampled with probability $\frac{\lambda}{n}$. In addition, there is a public (known to all) function, *committee-val*$(s, \lambda, i, \sigma_i)$, which gets a string s, a threshold λ, a process identification i and a proof σ_i, and returns *true* or *false*.

Consider a string s. For every i, $1 \leq i \leq n$, let $\langle v_i, \sigma_i \rangle$ be the return value of *sample*$_i(s, \lambda)$. The following is satisfied for every p_i:

- *committee-val*$(s, \lambda, i, \sigma_i) = v_i$.
- If p_i is correct, then it is infeasible for the adversary to compute *sample*$_i(s, \lambda)$.
- It is infeasible for the adversary to find $\langle v, \sigma \rangle$ s.t. $v \neq v_i$ and *committee-val*$(s, \lambda, i, \sigma) =$ *true*.

Due to space limitations, we present here the parameters and guarantees as presented and proven in [4] using Chernoff bounds. For simplicity, we only state the claims we are using in this paper.

Committee Sampling Properties from [4]. *Let the set of processes sampled to the committee for s and λ be $C(s, \lambda)$, where λ is set to $8 \ln n$. Let d be a parameter of the system such that $\frac{1}{\lambda} < d < \frac{\epsilon}{3} - \frac{1}{3\lambda}$. We set $W \triangleq \lceil (\frac{2}{3} + 3d)\lambda \rceil$ and $B \triangleq \lfloor (\frac{1}{3} - d)\lambda \rfloor$. With high probability the following hold:*

(S3) *At least W processes in $C(s, \lambda)$ are correct.*
(S4) *At most B processes in $C(s, \lambda)$ are Byzantine.*
(S5) *Consider $C(s, \lambda)$ for some string s and two sets $P_1, P_2 \subset C(s, \lambda)$ s.t $|P_1| = |P_2| = W$. Then, $|P_1 \cap P_2| \geq B + 1$.*

2 From Binary BA to Multivalued BA

In the Byzantine Agreement (BA) problem, a set Π of n processes attempt to reach a common decision. In addition, the decided value must be "valid" in some sense which makes the problem non-trivial. We consider two standard variants of BA for asynchrony that differ in their validity condition and the domain of inputs by processes in the system. In the binary version, all inputs are taken from the domain $\{0, 1\}$, while in the multivalued case they can be any value from any finite domain $\mathcal{V}$. For the validity condition, in order to support a larger domain we weaken the validity condition. Instead of the known *strong unanimity* property, we opt for *weak unanimity* as defined below. In this work we show how to reduce a subquadratic weak multivalued BA to a subquadratic binary strong BA, both are solved WHP. That is, a probability that tends to 1 as n goes to infinity. Formally, we take a black-box solution to:

▶ **Definition 1** (Binary Strong Byzantine Agreement WHP). *In Binary Strong Byzantine Agreement WHP, each correct process $p_i \in \Pi$ proposes a binary input value v_i and decides on an output value decision$_i$ s.t. with high probability the following properties hold:*

- *Validity (Strong Unanimity). If all correct processes propose the same value v, then any correct process that decides, decides v.*
- *Agreement. No two correct processes decide differently.*
- *Termination. Every correct process eventually decides.*

And use it to solve:

▶ **Definition 2** (Multivalued Weak Byzantine Agreement WHP). *In Multivalued Weak Byzantine Agreement WHP, each correct process $p_i \in \Pi$ proposes an input value v_i and decides on an output value decision$_i$ s.t. with high probability the following properties hold:*

Algorithm 1 Multivalued Byzantine Agreement(v_i): code for p_i.

```
    local variables: $alert \in \{true, false\}$, initially $false$
    $count \in \mathbb{N}$, initially 0
    $init\text{-}set, init\text{-}values\text{-}set, converge\text{-}set \in \mathcal{P}(\Pi)$, initially 0
 1: if $sample_i(\textsc{init}, \lambda) = true$ then broadcast $\langle \textsc{init}, v_i \rangle_i$
 2: upon receiving $\langle \textsc{init}, v_j \rangle_j$ with valid $v_j$ from validly sampled $p_j$ do
 3:     $init\text{-}set \leftarrow init\text{-}set \cup \{j\}$
 4:     $init\text{-}values\text{-}set \leftarrow init\text{-}values\text{-}set \cup \{v_j\}$
 5:     if $sample_i(\textsc{converge}, \lambda) = true$ and $|init\text{-}set| = W$ for the first time then
 6:         if $init\text{-}values\text{-}set = \{v_i\}$ then        ▷ All received values are $p_i$'s initial value
 7:             batch the $W$ messages into $QC_{v_i}$
 8:             send $\langle \textsc{converge}, true, QC_{v_i} \rangle_i$ to all processes
 9:         else
10:             send $\langle \textsc{converge}, false, \perp \rangle_i$ to all processes
11: upon receiving $\langle \textsc{converge}, is\_content, QC_v \rangle_j$ from validly sampled $p_j$ do
12:     $converge\text{-}set \leftarrow converge\text{-}set \cup \{j\}$
13:     if $is\_content{=}true$ then
14:         $count{+}= 1$
15:     when $|converge\text{-}set| = W$ for the first time
16:         $alert \leftarrow count < B + 1$
17:         $binary\_decision_i \leftarrow Binary\ Byzantine\ Agreement(alert)$
18:         if $binary\_decision_i = true$ then
19:             $decision_i \leftarrow \perp$
            else
20:             wait for a message of the form $\langle \textsc{converge}, true, QC_v \rangle_j$ from validly sampled $p_j$ if
    such was not already received
21:             $decision_i \leftarrow v$
```

- *Validity (Weak Unanimity). If all processes are correct and propose the same value v, then any correct process that decides, decides v.*
- *Agreement. Same as above.*
- *Termination. Same as above.*

We employ committee sampling and a binary subquadratic strong BA to present a multivalued solution to the weak BA problem. That is, the processes' initial values are from an arbitrary domain $\mathcal{V}$. We follow the method presented in [12] and adjust it to work with an asynchronous environment and committee sampling to achieve a subquadratic solution. The algorithm, presented in Algorithm 1, consists of two communication phases followed by a binary BA execution. To reduce the communication costs of the algorithm, the two phases are being executed only by a subset of the processes that are elected uniformly in random by a committee sampling primitive.

The first step is an INIT step, in which all INIT committee members send their signed initial value to all other processes (line 1). The second is a CONVERGE step, during which all CONVERGE committee members aim to converge around one common value. To do so, CONVERGE processes are waiting to hear from sufficiently many processes in the INIT committee. Since committees are elected using randomization, it is impossible for processes to wait for all of the previous committee members, as the size of the committee is unknown. Instead, it is guaranteed that a process hears from at least W processes WHP.

For some process p in the CONVERGE committee, if all W INIT messages include the same value v, that is also p's initial value, then p is considered to be *content*. To inform all other processes, p sends a CONVERGE message claiming to be content, that also carries a quorum certificate (QC) on the value v containing all received messages (line 8). In the complementary case, where p knows of at least two different values by line 6, it sends a CONVERGE message with a false is_content flag (line 10).

In the third part of the algorithm, processes run a binary consensus whose target is deciding whether the system as a whole is content. To do so, processes update an alert flag that is determined according to the number of content processes in the CONVERGE committee (lines 13 – 16). It is designed such that if a correct process p hears from at least one non-content correct process, it sets its alert flag to true. To do so, we utilize the parameter B which is, according to the specification, an upper bound on the number of Byzantine processes in a committee.

After processes set their boolean *alert* flag, they make a binary decision on its values at line 17. If the output is *true*, then all correct processes output $\perp$ (line 19). Notice that by the strong unanimity property of the binary BA, it is impossible that all correct processes have had *alert=false* before the execution. Namely, there were many non-content processes in the converge committee. Otherwise, if the binary decision is *true*, then it must be the case that (i) all content processes are content with respect to the same value and (ii) at least one correct process was content in the converge committee (proof appears in the full version [3]). In this case, that process carries a quorum certificate with W different signatures on a value v, that can be safely decided upon (line 21) and is guaranteed to eventually arrive at all correct processes. This is mainly thanks to the fact that two subsets of the converge committee of size W intersect by at least one correct process.

Complexity

In Algorithm 1 all correct processes that are sampled to the two committees (lines 1,5) send messages to all other processes. Each of these messages contains a value from the finite domain, a VRF proof of the sender's election to the committee, and possibly a quorum certificate of W different signatures. Therefore, each message's size is either a constant number of words or W words. Thus, the total word complexity of a multivalued weak BA WHP is $O(nWC)$ where C is the number of processes that are sampled to the committees. Since each process is sampled to a committee with probability $\frac{\lambda}{n}$, we get a word complexity of $O(nW\lambda) = O(n\log^2 n) = \widetilde{O}(n)$ in expectation.

In the full version [3], we prove the following theorem:

▶ **Theorem 3.** *Algorithm 1 implements multivalued weak Byzantine Agreement WHP with a word complexity of $\widetilde{O}(n)$.*

3 Conclusions

Real-world systems are asynchronous and prone to Byzantine failures. This paper presents an algorithm that reduces the multivalued weak BA WHP to binary strong BA. Together with the binary BA presented in [4, 2] this paper yields that first subquadratic multivalued BA.

References

1 Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically optimal validated asynchronous byzantine agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 337–346, 2019.

2 Erica Blum, Jonathan Katz, Chen-Da Liu-Zhang, and Julian Loss. Asynchronous byzantine agreement with subquadratic communication. Cryptology ePrint Archive, Report 2020/851, 2020.

3 Shir Cohen and Idit Keidar. Subquadratic multivalued asynchronous byzantine agreement whp. *arXiv preprint arXiv:2308.02927*, 2023.

4 Shir Cohen, Idit Keidar, and Alexander Spiegelman. Not a coincidence: Sub-quadratic asynchronous byzantine agreement whp. In *34th International Symposium on Distributed Computing (DISC 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

5 Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for byzantine agreement. *J. ACM*, 32(1):191–204, January 1985.

6 Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 51–68, 2017.

7 Valerie King and Jared Saia. Breaking the $O(n^2)$ bit barrier: scalable byzantine agreement with an adaptive adversary. *Journal of the ACM (JACM)*, 58(4):1–24, 2011.

8 Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

9 Silvio Micali, Michael Rabin, and Salil Vadhan. Verifiable random functions. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 120–130. IEEE, 1999.

10 Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. Signature-free asynchronous binary byzantine consensus with $t < n/3$, $O(n^2)$ messages, and $O(1)$ expected time. *Journal of the ACM (JACM)*, 62(4):31, 2015.

11 Achour Mostefaoui, Michel Raynal, and Frédéric Tronel. From binary consensus to multivalued consensus in asynchronous message-passing systems. *Information Processing Letters*, 73(5-6):207–212, 2000.

12 Russell Turpin and Brian A Coan. Extending binary byzantine agreement to multivalued byzantine agreement. *Information Processing Letters*, 18(2):73–76, 1984.

Brief Announcement: Distributed Derandomization Revisited

Sameep Dahal
Aalto University, Finland

Francesco d'Amore
Aalto University, Finland

Henrik Lievonen
Aalto University, Finland

Timothé Picavet
Aalto University, Finland
ENS de Lyon, France

Jukka Suomela
Aalto University, Finland

Abstract

One of the cornerstones of the distributed complexity theory is the derandomization result by Chang, Kopelowitz, and Pettie [FOCS 2016]: any randomized LOCAL algorithm that solves a locally checkable labeling problem (LCL) can be derandomized with at most exponential overhead. The original proof assumes that the number of random bits is bounded by some function of the input size. We give a new, simple proof that does not make any such assumptions – it holds even if the randomized algorithm uses infinitely many bits. While at it, we also broaden the scope of the result so that it is directly applicable far beyond LCL problems.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Theory of computation → Pseudorandomness and derandomization

Keywords and phrases Distributed algorithm, Derandomization, LOCAL model

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.40

Funding This work was supported in part by the Research Council of Finland, Grant 333837.

Acknowledgements We thank the participants in our reading group at Aalto University for helpful discussions.

1 Introduction

Distributed derandomization. A long line of recent work has led to a near-complete understanding of the distributed computational complexity of *locally checkable labeling problems* (LCLs) [5]. These are graph problems that can be defined by giving a finite list of feasible local neighborhoods [3]; for example, c-coloring in graphs of maximum degree Δ (for some fixed c and Δ) is an LCL problem.

We are in particular interested in the round complexity of LCLs in two standard models of distributed computing: deterministic and randomized versions of the LOCAL model [2, 4]. One of the cornerstones of the distributed complexity theory is the derandomization result by Chang, Kopelowitz, and Pettie [1, Theorem 3.1]:

▶ **Theorem 1** (Chang, Kopelowitz, and Pettie). *Let $\mathcal{A}_{\text{rand}}$ be a randomized LOCAL algorithm that solves an LCL problem $\mathcal{P}$ in $T_{\text{rand}}(n)$ communication rounds in n-node graphs with probability at least $1 - 1/n$. Then there is a deterministic LOCAL algorithm $\mathcal{A}_{\text{det}}$ that solves $\mathcal{P}$ in $T_{\text{det}}(n)$ rounds, where $T_{\text{det}}(n) = T_{\text{rand}}(2^{n^2})$.*

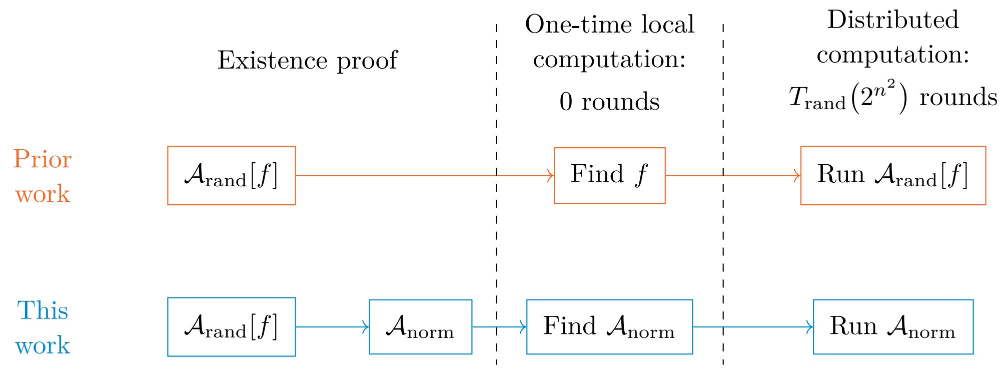

Existence proof
One-time local computation: 0 rounds
Distributed computation: $T_{\text{rand}}(2^{n^2})$ rounds
Prior work
$\mathcal{A}_{\text{rand}}[f]$
Find f
Run $\mathcal{A}_{\text{rand}}[f]$
This work
$\mathcal{A}_{\text{rand}}[f]$
$\mathcal{A}_{\text{norm}}$
Find $\mathcal{A}_{\text{norm}}$
Run $\mathcal{A}_{\text{norm}}$

Figure 1 Proof strategy in this work and prior work [1].

But what do we mean, precisely, when we say that $\mathcal{A}_{\text{rand}}$ is a randomized algorithm in the LOCAL model? Chang, Kopelowitz, and Pettie [1] assume that there is some upper bound $r(n)$ on the number of random bits used by a node. This is a non-standard definition; while many reasonable algorithms naturally satisfy this, formally speaking it is not compatible with e.g. a randomized algorithm in which each node picks a number from a geometric distribution by repeated Bernoulli trials. All other results that build on Theorem 1 are also influenced by this assumption; the foundations of the field are on a bit shaky ground.

New result: unbounded randomness. In this short note we prove a stronger version of Theorem 1. Our proof does not need to assume anything about the number of random bits consumed by a node. Hence, we can now safely conclude that all corollaries of Theorem 1 also hold in the standard randomized LOCAL model, in which local computation – including the number of random bits generated – is unbounded.

Similar to [1], we assume that n and T_{rand} (or sufficiently tight bounds on them) are known. Similar to [1], the proof is constructive and $\mathcal{A}_{\text{det}}$ is a uniform, computable, deterministic algorithm. The only difference is that we assume less about $\mathcal{A}_{\text{rand}}$.

Key new ideas. Exactly like Chang, Kopelowitz, and Pettie [1, Theorem 3.1], we start by defining $N = 2^{n^2}$. Then even though we are working in an n-node graph, we lie to $\mathcal{A}_{\text{rand}}$ that we have a graph with N nodes. The running time increases to $T_{\text{rand}}(N)$, but the success probability improves to $1 - 1/N$, which is large enough to show that there exists a mapping f from unique identifiers to random bits that works for every n-node graph.

At this point our paths deviate – see Figure 1 for an illustration. In [1, Theorem 3.1], $\mathcal{A}_{\text{det}}$ is constructed as follows: Each node checks each possible mapping f, and picks the first one that works for every n-node graph; then $\mathcal{A}_{\text{det}}$ simply simulates $\mathcal{A}_{\text{rand}}$ with random bits from f. This is where they make use of bounded randomness: for a fixed n there are only finitely many possible functions f to check.

We proceed as follows – instead of looking at the *internal behavior* of the algorithm we look at its *external behavior*:

1. Since a good mapping f exists, we could in principle hard-code this specific mapping to obtain a deterministic algorithm $\mathcal{A} = \mathcal{A}_{\text{rand}}[f]$. At this point we merely know that $\mathcal{A}$ exists – this step is non-constructive, and $\mathcal{A}$ might not even be computable.

2. However, any deterministic LOCAL algorithm can be represented in a *normal form* as a function $\mathcal{A}_{\text{norm}}$ that maps each possible $T_{\text{rand}}(N)$-radius neighborhood to a local output. Since $\mathcal{A}$ exists, we know that such a function $\mathcal{A}_{\text{norm}}$ also exists and solves $\mathcal{P}$ correctly in all n-node graphs.
3. Now $\mathcal{A}_{\text{det}}$ simply finds the first valid $\mathcal{A}_{\text{norm}}$, and then simulates $\mathcal{A}_{\text{norm}}$.

This way we can construct a computable, uniform, deterministic algorithm $\mathcal{A}_{\text{det}}$ even if we merely know that $\mathcal{A}_{\text{rand}}$ exists, and even if $\mathcal{A}_{\text{rand}}$ is non-computable or non-uniform.

Two extensions. While Theorem 1 was originally presented for LCL problems, our new proof works for a broader class of problems: we show how to handle labeling problems that are defined component-wise. The proof is given in Section 3; Theorem 1 then follows as a special case.

We also briefly discuss in Section 4 one extension: how to derandomize algorithms that are only guaranteed to work in connected graphs. A bit more care is needed when we lie about the number of nodes in that case.

2 Preliminaries

Let $G = (V, E)$ denote a simple undirected graph. For any two nodes $u, v \in V$, we denote their distance by $d(u, v)$, i.e., the number edges in a shortest path connecting u to v; if such path does not exist, then $d(u, v) = +\infty$. Furthermore, by $\deg(v)$ we denote the degree of v, i.e., the number of incident edges.

LOCAL model. Let $G = (V, E)$ be any graph with n nodes. In the *deterministic LOCAL model*, each node $v \in V$ is given a unique identifier $\text{id}(v) \in \{1, 2, \dots, n^c\}$ for some constant $c \geq 1$. The initial knowledge of a node consists of its own identifier, its degree, the number of nodes n and (possibly) an input label. Each node runs the same algorithm and computation proceeds in synchronous rounds. In each round, nodes send messages of arbitrary size to their neighbors, then receive some messages, and then perform local computations of arbitrary complexity. After some number of rounds, a node must terminate its computation and decide on its local output. The running time (or complexity) of a distributed algorithm is defined as the number of rounds needed by all nodes to decide the local output.

In the *randomized LOCAL model*, each node is also given access to an infinite random bit stream, and the bit streams of the nodes are mutually independent. We say that an algorithm is *uniform* if the size of the description of the algorithm does not depend on n.

For any fixed locality T, the LOCAL model can also be viewed as a mapping from each radius-T neighborhood $N_T[v]$ of each node v to a local output. Here by $N_T[v]$ we mean the graph (V', E'), where $V' \subseteq V$ is the set of all nodes $u \in V(G)$ with $d(v, u) \leq T$ and E' is the set of edges $\{s, t\} \in E$ with $d(v, s) \leq T - 1$ and $d(v, t) \leq T$. Each node of $N_T[v]$ is also labeled with its original degree $\deg(u)$, unique identifier $\text{id}(u)$, local input, and – for randomized algorithms – its stream of random bits. This is exactly the information node v can gather in T rounds.

Labeling problems. Let Σ_{in} be a finite set of input labels and Σ_{out} be a finite set of admissible output labels. An *input labeling* of a graph $G = (V, E)$ is a function $\lambda_{\text{in}} \colon V \to \Sigma_{\text{in}}$, and an *output labeling* is a function $\lambda_{\text{out}} \colon V \to \Sigma_{\text{out}}$. A *labeling problem* $\mathcal{P}$ specifies for each graph and each input labeling a set of feasible output labelings.

We say that $\mathcal{P}$ is a *component-wise verifiable problem* if for each graph G and each connected component C of G, the set of valid output labelings restricted to C only depends on C.

Let $r \in \mathbb{N}$ be a constant. We say that $\mathcal{P}$ is a *locally verifiable problem* with verification radius r if for each graph G and each node v of G, the set of valid output labelings restricted to $N_r[v]$ only depends on $N_r[v]$.

We note that LCL problems [3] are a special case of locally verifiable problems with a constant bound on the degree of the nodes. Locally verifiable problems are in turn a special case of component-wise verifiable problems.

3 Main result

We give the derandomization result directly for component-wise verifiable problems; Theorem 1 then follows as a corollary.

▶ **Theorem 2.** *Let $\mathcal{A}_{\text{rand}}$ be a randomized LOCAL algorithm that solves a component-wise verifiable problem $\mathcal{P}$ in $T_{\text{rand}}(n)$ communication rounds in n-node graphs with probability at least $1 - 1/n$. Then there is a deterministic LOCAL algorithm $\mathcal{A}_{\text{det}}$ that solves $\mathcal{P}$ in $T_{\text{det}}(n)$ rounds, where $T_{\text{det}}(n) = T_{\text{rand}}\left(2^{n^2}\right)$.*

Proof. Consider any sufficiently large n, and let $N = 2^{n^2}$. In what follows, we lie to algorithm $\mathcal{A}_{\text{rand}}$ that the input graph consists of N nodes. Hence, it runs in time $T := T_{\text{rand}}(N) = T_{\text{det}}(n)$ and succeeds with probability $1 - 1/N$.

Let $\mathcal{R}_n = \{f \colon \{0,1\}^{c \log n} \to \{0,1\}^{\mathbb{N}}\}$ be the family of all possible assignments of random bits streams to unique identifiers. For $f \in \mathcal{R}_n$, we write $\mathcal{A}_{\text{rand}}[f]$ to denote the *deterministic* LOCAL algorithm in which node v runs $\mathcal{A}_{\text{rand}}$ but uses $f(\text{id}(v))$ as its random bit stream. Note that $\mathcal{A}_{\text{rand}}$ is equivalent to the following process: choose $f \in \mathcal{R}_n$ uniformly at random and apply $\mathcal{A}_{\text{rand}}[f]$.

Let $\mathcal{G}_n$ be the set of all possible inputs $(G, \text{id}, \lambda_{\text{in}})$, where G is an n-node graph, id is a unique identifier assignment, and λ_{in} is an input labeling. We know that

$$|\mathcal{G}_n| \le 2^{\binom{n}{2}} \cdot 2^{cn \log n} \cdot |\Sigma_{\text{in}}|^n < N = 2^{n^2}$$

for a large enough n. We say that f is *good* if $\mathcal{A}_{\text{rand}}[f]$ outputs a valid solution for *every* input in family $\mathcal{G}_n$.

Now, we show there exists a good f. Let F be a uniform random variable over $\mathcal{R}_n$. Then

$$\Pr(F \text{ is bad}) \le \sum_{G \in \mathcal{G}_n} \Pr(\mathcal{A}_{\text{rand}}[F] \text{ fails on } G) = \sum_{G \in \mathcal{G}_n} \Pr(\mathcal{A}_{\text{rand}} \text{ fails on } G) \le \frac{|\mathcal{G}_n|}{N} < 1.$$

Therefore, $\Pr(F \text{ is good}) > 0$. Hence, there exists a good function; let f be any such function. Thus, there is a deterministic algorithm $\mathcal{A} = \mathcal{A}_{\text{rand}}[f]$ that solves $\mathcal{P}$ on all inputs in $\mathcal{G}_n$ in at most T rounds.

Any deterministic T-round algorithm in the LOCAL model defines a mapping $\mathcal{A}_{\text{norm}}$ from radius-T neighborhoods to local outputs. Conversely, such a mapping $\mathcal{A}_{\text{norm}}$ can be interpreted as a T-round algorithm. Furthermore, for a fixed n, there are only finitely many such mappings.

Now $\mathcal{A}_{\text{det}}$ works as follows: Given n, each node first enumerates all candidate mappings $\mathcal{A}_{\text{norm}}$ in lexicographic order, checks if $\mathcal{A}_{\text{norm}}$ solves $\mathcal{P}$ for every $\mathcal{G}_n$, and stops once the first such $\mathcal{A}_{\text{norm}}$ is found. Then $\mathcal{A}_{\text{det}}$ uses T rounds so that each node v learns its radius-T neighborhood $N_T[v]$, and finally each node applies mapping $\mathcal{A}_{\text{norm}}$ to $N_T[v]$ to determine its local output. ◀

4 Technicality: connected graphs

In the proof of Theorem 2, a key step was that we lied about n. The algorithm cannot catch us lying, as the n-node input graph G is indistinguishable from some hypothetical N-node input graph G' in which one connected component is isomorphic to G. As $\mathcal{P}$ was assumed to be component-wise verifiable, an algorithm that succeeds globally in G' also has to succeed locally when restricted to G.

The proof heavily exploited graphs that may consist of multiple connected components. In this section we briefly note that this is *not necessary*. We can prove the following version of Theorem 2 that holds even if $\mathcal{A}_{\text{rand}}$ only works correctly in connected graphs. However, component-wise problems are too broad class of problems in this case, and we consider locally verifiable problems instead:

▶ **Theorem 3.** *Let $\mathcal{A}_{\text{rand}}$ be a randomized LOCAL algorithm that solves a locally verifiable problem $\mathcal{P}$ in $T_{\text{rand}}(n)$ communication rounds in n-node connected graphs with probability at least $1 - 1/n$. Then there is a deterministic LOCAL algorithm $\mathcal{A}_{\text{det}}$ that solves $\mathcal{P}$ in $O(T_{\text{det}}(n))$ rounds, where $T_{\text{det}}(n) = T_{\text{rand}}(2^{n^2})$.*

Proof. Let $t = T_{\text{det}}(n) + r$, where r is the verification radius of problem $\mathcal{P}$. In algorithm $\mathcal{A}_{\text{det}}$, each node v first explores its radius-t neighborhood to determine if the entire input graph G is contained in $N_t[v]$. If yes, we spend another t rounds to inform all nodes about G. In this case all nodes have learned G, and we can solve $\mathcal{P}$ by brute force and stop.

Otherwise, we can proceed as we did in the proof of Theorem 2. We can now safely lie about N. To see this, assume that $\mathcal{A}_{\text{rand}}$ fails in some n-node graph G with probability more than n/N if we lie that G has N nodes. Then the algorithm also has to fail locally in the radius-r neighborhood of some node v with probability more than $1/N$. Now it is possible to construct an N-node graph G' with node v' such that radius-t neighborhood of v in G is isomorphic to the radius-t neighborhood of v' in G' (here we exploit the fact that radius-t neighborhood of v does not contain the entire graph G). As radius-t neighborhoods of v and v' agree, and the running time of $\mathcal{A}_{\text{rand}}$ is $t - r$ rounds, the output distributions of $N_r[v]$ and $N_r[v']$ also agree. Now it follows that $\mathcal{A}_{\text{rand}}$ fails locally in the radius-r neighborhood of v' in G' with probability more than $1/N$, and hence it also fails globally in G' with probability more than $1/N$, which is a contradiction with the assumption that $\mathcal{A}_{\text{rand}}$ solves $\mathcal{P}$ in connected N-node graphs with probability at least $1 - 1/N$.

Now as long as we choose a large enough n such that $|\mathcal{G}_n| < N/n$, the rest of the proof of Theorem 2 goes through. ◀

References

1 Yi-Jun Chang, Tsvi Kopelowitz, and Seth Pettie. An exponential separation between randomized and deterministic complexity in the local model. *SIAM Journal on Computing*, 48(1):122–143, 2019. doi:10.1137/17M1117537.

2 Nathan Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992. doi:10.1137/0221015.

3 Moni Naor and Larry J. Stockmeyer. What can be computed locally? *SIAM Journal on Computing*, 24(6):1259–1277, 1995. doi:10.1137/S0097539793254571.

4 David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. Society for Industrial and Applied Mathematics, 2000. doi:10.1137/1.9780898719772.

5 Jukka Suomela. Landscape of locality (invited talk). In *17th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT 2020)*, 2020. doi:10.4230/LIPIcs.SWAT.2020.2.

Brief Announcement: Byzantine Consensus Under Dynamic Participation with a Well-Behaved Majority

Eli Gafni
University of California, Los Angeles, CA, USA

Giuliano Losa
Stellar Development Foundation, San Francisco, CA, USA

Abstract

In a permissionless system like Ethereum, participation may fluctuate dynamically as some participants unpredictably go offline and some others come back online. In such an environment, traditional Byzantine fault-tolerant consensus algorithms may stall – even in the absence of failures – because they rely on the availability of fixed-sized quorums.

The sleepy model formally captures the main requirements for solving consensus under dynamic participation, and several algorithms solve consensus with probabilistic safety in this model assuming that, at any time, more than half of the online participants are well behaved. However, whether safety can be ensured deterministically under these assumptions, especially with constant latency, remained an open question.

Assuming a constant adversary, we answer in the positive by presenting a consensus algorithm that achieves deterministic safety and constant latency in expectation. In the full version of this paper, we also present a second algorithm which obtains both deterministic safety and liveness, but is likely only of theoretical interest because of its high round and message complexity. Both algorithms are striking in their simplicity.

2012 ACM Subject Classification Computer systems organization → Dependable and fault-tolerant systems and networks

Keywords and phrases Consensus, Sleepy Model, Dynamic Participation, Byzantine Failures

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.41

Related Version *Full Version*: `https://arxiv.org/abs/2301.04817`

Supplementary Material *Model*: `https://doi.org/10.5281/zenodo.8226236`

1 Introduction

In a permissionless system like Ethereum, the parties running the consensus algorithm are known and fixed (up to reconfigurations) but they may unpredictably go offline or come back online. We say that participation is dynamic.

Ideally, we would like to solve consensus even if the set of online participants fluctuates unpredictably, as long as a sufficient fraction of those who are online are well-behaved. BFT consensus algorithms like PBFT [5], Algorand Agreement [6], or Tendermint [4] do not work in this setting because they rely on the availability of fixed-size quorums to make progress, and thus they stall – even in the absence of failures – if too many participants go offline.

In this paper, we consider the consensus problem in a synchronous system with dynamic participation similar to the sleepy model [19]. We assume that a fixed set of processes execute a sequence of rounds where, each round, an adversary partitions the processes into three sets: offline processes, online and faulty (i.e. controlled by the adversary) processes, and online and well-behaved processes. Communication is synchronous, which means that a message sent in round r is guaranteed to be received by all processes that are online in round $r + 1$. Importantly, in each round, the processes do not know a priori who is online and who is not, nor who is well behaved and who is faulty.

Several algorithms solve consensus in variants of this setting [19, 2, 8, 13, 18, 16, 9, 20]. However, one question that remained open is whether we can solve consensus under dynamic participation with deterministic safety and constant latency in expectation, assuming that: a) each round, a strict majority of the online participants are well behaved, b) a PKI and verifiable random functions [17] (VRF) are available, and c) the faulty set is constant from one round to the next. In this paper, we present such a consensus algorithm.

Solving consensus under those constraints is not easy. On the one hand, we could imagine using a variant of the Dolev-Strong algorithm [10]. However, this algorithm takes a number of rounds linear in the number of failures, while we aim at constant latency. On the other hand, it seems that techniques based on intersecting quorums will not work: In a given round, it is possible that an online, well-behaved process p receives a message m from a strict majority of the processes it hears of, while another online, well-behaved process p' receives a message $m' \neq m$ from a strict majority of the processes it hears of.

We start by observing that we can prevent faulty processes from equivocating and witholding messages to some processes, using a simple two-round algorithm. This allows us to simulate what we call the no-equivocation model, in which it is no longer possible for two well-behaved processes to hear of conflicting strict majorities. Next, taking inspiration from Gafni and Zikas [12], we solve the commit-adopt problem [11] (a graded agreement [7] with two grades) deterministically and in exactly 2 rounds of the no-equivocation model (for a total of 4 rounds of the base model).

Finally, using commit-adopt, we propose a consensus algorithm reminiscent of the phase-king algorithm [3], where we use a probabilistically-elected leader (which can be done using VRFs) instead of selecting a king round-robin. The algorithm ensures safety deterministically, terminates in 18 rounds in expectation, and, in an execution with at most m online participants, each process broadcasts at most m messages each round. Whether the termination bound and message complexity can be improved is left for future work.

In the full version of this paper [15], we generalize the model to a setting where the set of processes has unknown cardinality and failures are governed by a more powerful mobile message adversary. We then show with detailed proofs that the consensus algorithm described in the present paper works in this model, and we present another consensus algorithm, inspired by the Dolev-Strong algorithm [10], that achieves both deterministic safety and liveness, although with a linear latency in the number of failures (even if the set of processes is unknown) as long as there is a strict subset of the processes that contains all the faulty sets.

2 The model

We consider a finite, nonempty set $\mathcal{P}$ of processes running an algorithm in a synchronous, message-passing system with point-to-point links between every pair of processes. An unknown subset $F \subset \mathcal{P}$ of the processes is faulty.

The set $\mathcal{P}$ is publicly known and every process p has a key pair K_p whose public component is also known to all (this is an abstraction of a PKI). We write $K_p(m)$ for the message m signed with the private component of K_p.

The system executes an infinite sequence of rounds numbered $1, 2, 3, \ldots$. Each round r, an adversary partitions the processes into a nonempty set O_r of online processes and a set $\mathcal{P} \setminus O_r$ of offline processes such that a) all the faulty processes are online (i.e. $F \subseteq O_r$) and b) the faulty processes consist of a strict minority of the online processes (i.e. $2|F| < |O_r|$). The sets O_r and F are a priori unknown to the processes.

Operationally, execution proceeds as follows. Initially, each process receives an external input. Then, each round r consists of a send phase followed by a receive phase. In the send phase, each online, well-behaved process sends a set of messages that, if $r = 1$, is a function of its input, and that otherwise is a function of the set of messages it received in the preceding round and of the output of a leader-election oracle described below. In both cases, the algorithm determines the function. Faulty processes are controlled by the adversary and may deviate by sending arbitrary messages, except that they cannot forge signatures. In the receive phase, each process, online or not, receives all the messages sent to it in the round[1].

The leader-election oracle gives an output to each online process at the beginning of every round $r > 1$, and it ensures with probability $1/2$ that every process that is online and well-behaved in round r receives the identity of a unique process that is online and well-behaved in round $r - 1$. In practice, the oracle can be implemented using VRFs by selecting the process that has the highest VRF output.

Note that, as mentioned in the introduction, a difficulty in this model is that two well-behaved processes may witness two conflicting strict majorities in the same round. For example, take 5 processes p_1 to p_5 and consider a round r where all processes participate and only p_4 and p_5 are faulty. Let p_1 and p_2 broadcast message m while p_3 broadcasts message $m' \neq m$. Moreover, let p_4 and p_5 send message m' to p_1 while they do not send any messages to p_2. Observe that p_1 hears of 5 processes and receives m' from 3 processes, and so it witnesses a strict majority for m'. However, p_2 hears of 3 processes and receives m from 2 processes, so it witnesses a strict majority for m.

3 Implementing commit-adopt

The most important building block of the consensus algorithm we present in the next section is a solution to the commit-adopt problem. In the commit-adopt problem, each online process initially receives an arbitrary input. After a fixed number of rounds R, each online process must produce an output either of the form $\langle \text{commit}, v \rangle$ for some v, in which case we say the process commits v, or of the form $\langle \text{adopt}, v \rangle$ for some v, in which case we say that the process adopts v. The outputs must satisfy the following properties:

Agreement If a well-behaved process commits a value v, then every process that is online and well-behaved in round R must either commit or adopt v.

Validity If all well-behaved processes that initially receive an input receive the same value v as input, then all well-behaved processes that are online in round R commit v.

Next, we present an algorithm that implements commit-adopt in $R = 4$ rounds. We first simulate a model that we call the no-equivocation model, and then implement commit-adopt in the no-equivocation model.

[1] Assuming that all processes receive messages is convenient, and it does not make things easier since only the processes that are online in round $r + 1$ can use the messages they received in round r.

3.1 Simulating the no-equivocation model

The no-equivocation model is similar to the base model of the preceding section except that, each round r, each online process broadcasts a single message, and faulty processes may deviate only by omitting to send their message to some processes (they cannot equivocate, i.e. send different messages to different processes). Moreover, when a faulty process q deviates in round r, then the online, well-behaved processes of the next round $r+1$ get a failure notification for q, noted λ, and such that one of the following three cases hold in round $r+1$:

1. All online, well-behaved processes receive the same message m from q, or
2. Some online, well-behaved processes receive the same message m from q while all the others receive the failure notification λ for q, or
3. Some online, well-behaved processes receive the failure notification λ for q while all the others do not hear of q.

Note that the no-equivocation model does not allow for two different online, well-behaved processes to witness two conflicting strict majorities of messages in the same round. Moreover, the failure notification λ limits the ability of faulty processes to make other processes witness different levels of participation.

We now simulate each no-equivocation round in two rounds of the base model:

1. In the first round, each process p must broadcast the tuple $\langle m, K_p(m)\rangle$, where m is the message to simulate the broadcasting of.
2. In the second round, each process p must broadcast $\langle \text{heard-of}, m, K_p(m)\rangle$ for each message $\langle m, K_p(m)\rangle$ that it received in the first round. Finally, at the end of the second round, for each process q such that p receives $\langle \text{heard-of}, m_q, K_q(*)\rangle$ for some message $*$:
   a. If there is a message m' such that p receives $\langle \text{heard-of}, m', K_q(m')\rangle$ from a strict majority of the processes it hears of and p does not receive $\langle \text{heard-of}, m'', K_q(m'')\rangle$ for any $m'' \neq m'$, then p must simulate receiving m' for q.
   b. Otherwise, p must simulate receiving the failure notification λ for q.

Essentially, processes broadcast their message and then tell each other what messages they received in order to detect equivocations or missing messages. Note that the simulated messages satisfy the minority-failure requirement because, since the set of faulty processes being constant, the adversary cannot inflate round-1 participation after the fact.

3.2 Implementing commit-adopt in the no-equivocation model

We now implement commit-adopt in 2 rounds of the no-equivocation model (which amounts to 4 rounds of the base model) as follows.

1. In the first no-equivocation round, each process must broadcast its input in_p.
2. In the second no-equivocation round, each process must broadcast $\langle \text{propose-commit}, v\rangle$ if it receives v from a strict majority of the processes it hears of, and otherwise it must broadcast $\langle \text{no-commit}\rangle$.
3. Finally, at the end of the second no-equivocation round, for each process p:
   a. If p receives $\langle \text{propose-commit}, v\rangle$ from a strict majority of the processes it hears of, then it must commit v.
   b. Else, if there is a value v such that p receives $\langle \text{propose-commit}, v\rangle$ more often than it receives $\langle \text{propose-commit}, v'\rangle$ for any other value v', then p must adopt v.
   c. Else, p must adopt its input in_p.

Let us sketch why the commit-adopt algorithm satisfies the agreement property. Assume that a well-behaved process p commits a value v. Note that it suffices to show that, for every $v' \neq v$, no well-behaved process p' receives $\langle \text{propose-commit}, v'\rangle$ more often than

$\langle$propose-commit$, v\rangle$. For every process q and value w, let count(q, w) be the number of times q receives the message $\langle$propose-commit$, w\rangle$. With this notation, what we would like to show is that count$(p', v) -$ count$(p', v') > 0$.

Note that, since processes may not witness conflicting strict majorities in the no-equivocation model, no well-behaved process broadcasts $\langle$propose-commit$, v'\rangle$ for $v' \neq v$. Additionally, remember that a faulty process q cannot send different messages to different processes and that, should it selectively send a message to only some processes but not others, the others receive the failure notification λ for q. From this, we get that count$(p', v) -$ count(p', v') is equal to count(p, v) minus the number of processes q such that either a) p receives $\langle$propose-commit$, v\rangle$ from q and p' receives the failure notification λ for q, or b) p' receives $\langle$propose-commit$, v'\rangle$ from q and p did not receive $\langle$propose-commit$, v\rangle$ from q. In both cases, q must be a faulty process. Thus, we have count$(p', v) -$ count$(p', v') \geq$ count$(p, v) - |F_2|$, where F_2 is the set of faulty processes in the second no-equivocation round.

Finally, since p commits v, we have count$(p, v) > |F_2|$.[2] We conclude that count$(p', v) -$ count$(p', v') > 0$, i.e. p' receives $\langle$propose-commit$, v\rangle$ more often than $\langle$propose-commit$, v'\rangle$.

4 Consensus with deterministic safety and constant expected latency

In the consensus problem, each process that is initially online receives an input, and each process may produce an output called a decision subject to the following requirements:

Agreement No two well-behaved processes produce different decisions.

Validity If all processes receive the same input v, the no well-behaved process decides $v' \neq v$.

Liveness With nonzero probability, in some round, all online, well-behaved processes decide.

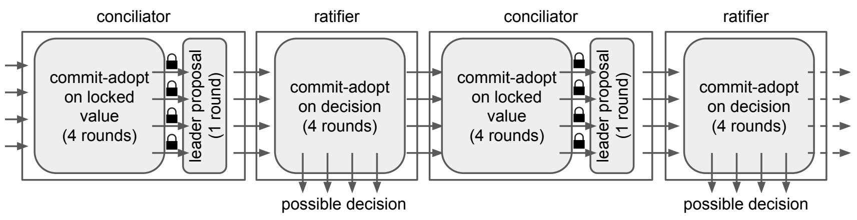

conciliator
ratifier
conciliator
ratifier
commit-adopt on locked value (4 rounds)
leader proposal (1 round)
commit-adopt on decision (4 rounds)
commit-adopt on locked value (4 rounds)
leader proposal (1 round)
commit-adopt on decision (4 rounds)
possible decision
possible decision

Figure 1 Infinite alternating sequence of conciliators and ratifiers. Horizontal arrows represent processes locally taking their output from one block and using it as input to the next block, while vertical arrows represent processes possibly outputing a consensus decision.

To solve consensus, we use the construction shown in Figure 1. It consists of an infinite alternating sequence of conciliators and ratifiers (following the terminology of Aspnes [1]), starting with a conciliator. Informally, a ratifier tries to produce a consensus decision using commit-adopt, but since processes are not guaranteed to commit, it may fail to do so. Thus, the job of a conciliator is to try to make processes agree on their input to the next ratifier. We do this using a leader. However, to ensure that no leader overrides a previous decision, processes first negotiate, again using commit-adopt, on whether to listen to the leader or not.

More precisely, a ratifier simply consists of an instance of the commit-adopt algorithm in which processes try to commit a consensus decision. Each process p inputs $\langle$decide$, v_p\rangle$ to the commit-adopt, where v_p is p's output in the preceding conciliator, and each process that commits $\langle$decide$, v\rangle$ for some v decides v as a consensus decision.

[2] This is only true if all faulty processes make themselves heard. A complete proof appears in the full version of this paper.

In a conciliator, each process p first inputs $\langle \text{lock}, v_p \rangle$ into a commit-adopt instance to try to lock the value v_p that it gets as output in the preceding ratifier. Then follows an additional round, called the leader-proposal round, in which each process p broadcasts its commit-adopt output. At the end of the leader-proposal round, for each process p that receives $\langle \text{commit}, \langle \text{lock}, v \rangle \rangle$ for some v from a strict majority of the processes it hears of, p considers v locked and outputs v. Otherwise, p outputs the value v received from the process identified as leader by the leader-election oracle, if any, and else outputs an arbitrary value.

The lock mechanism ensures that a value unanimously supported by online, well-behaved processes at the beginning of the conciliator cannot be overridden during the leader-proposal round. Thus, once a value is first decided, all online, well-behaved processes keep unanimously supporting that value in all subsequent rounds and the agreement property of consensus is guaranteed. The validity property holds for similar reasons.

Finally, for liveness, note that if the oracle outputs the same online, well-behaved leader to all, then all online, well-behaved processes output the same value from the conciliator (because if an online, well-behaved process considers a value locked, then the leader must have proposed that value). Since this happens with probability 1/2, the liveness property holds. Moreover, given that a ratifier takes 4 rounds and a conciliator takes 5 rounds, it takes at best 9 rounds to decide and 18 rounds in expectation.

5 Related Work

Starting with Bitcoin's longest-chain protocol, a series of works solve consensus under dynamic participation with probabilistic safety [19, 8, 2, 9, 13].

In this paper, we are interested in deterministic safety. Sandglass [20] achieves deterministic safety under a minority of benign failures even under a growing adversary. The algorithm we present also works in the Sandglass model: without Byzantine failures, we do not need to assume a constant adversary.

Momose and Ren [18] tolerate a minority of Byzantine failures under an eventual-stabilization assumption. Malkhi, Momose, and Ren [16] remove the stabilization assumption but tolerate only one-third failures. We improve on this result by tolerating a minority (i.e. less than 1/2) of failures, albeit only under a constant adversary. Whether there is an algorithm that tolerates a growing adversary remains open.

Finally, we have used a simple model in order to focus on essential algorithmic issues. See Lewis-Pye and Roughgarden [14] for more holistic models of permissionless systems.

References

1 James Aspnes. A modular approach to shared-memory consensus, with applications to the probabilistic-write model. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, 2010. doi:10.1145/1835698.1835802.

2 Christian Badertscher, Peter Gaži, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Ouroboros Genesis: Composable Proof-of-Stake Blockchains with Dynamic Availability. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018. doi:10.1145/3243734.3243848.

3 P. Berman, J. A. Garay, and K. J. Perry. Towards optimal distributed consensus. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, 1989. doi:10.1109/SFCS.1989.63511.

4 Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus, 2019. doi:10.48550/arXiv.1807.04938.

5 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20, 2002. doi:10.1145/571637.571640.

6 Jing Chen, Sergey Gorbunov, Silvio Micali, and Georgios Vlachos. ALGORAND AGREEMENT: Super Fast and Partition Resilient Byzantine Agreement, 2018. URL: https://eprint.iacr.org/2018/377.

7 Jing Chen and Silvio Micali. Algorand: A secure and efficient distributed ledger. *Theoretical Computer Science*, 777, 2019. doi:10.1016/j.tcs.2019.02.001.

8 Phil Daian, Rafael Pass, and Elaine Shi. Snow White: Robustly Reconfigurable Consensus and Applications to Provably Secure Proof of Stake. In *Financial Cryptography and Data Security*, 2019. doi:10.1007/978-3-030-32101-7_2.

9 Francesco D'Amato, Joachim Neu, Ertem Nusret Tas, and David Tse. No More Attacks on Proof-of-Stake Ethereum? *arXiv preprint arXiv:2209.03255*, 2022.

10 D. Dolev and H. R. Strong. Authenticated Algorithms for Byzantine Agreement. *SIAM Journal on Computing*, 12, 1983. doi:10.1137/0212045.

11 Eli Gafni. Round-by-round Fault Detectors (Extended Abstract): Unifying Synchrony and Asynchrony. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, 1998. doi:10.1145/277697.277724.

12 Eli Gafni and Vasileios Zikas. Synchrony/Asynchrony vs. Stationary/Mobile? The Latter is Superior...in Theory, 2023. doi:10.48550/arXiv.2302.05520.

13 Vipul Goyal, Hanjun Li, and Justin Raizes. Instant Block Confirmation in the Sleepy Model. In *Financial Cryptography and Data Security*, 2021. doi:10.1007/978-3-662-64331-0_4.

14 Andrew Lewis-Pye and Tim Roughgarden. Permissionless Consensus, 2023. doi:10.48550/arXiv.2304.14701.

15 Giuliano Losa and Eli Gafni. Consensus in the Unknown-Participation Message-Adversary Model, January 2023. doi:10.48550/arXiv.2301.04817.

16 Dahlia Malkhi, Atsuki Momose, and Ling Ren. Byzantine Consensus under Fully Fluctuating Participation, 2022. URL: https://eprint.iacr.org/2022/1448.

17 S. Micali, M. Rabin, and S. Vadhan. Verifiable random functions. In *40th Annual Symposium on Foundations of Computer Science*, 1999. doi:10.1109/SFFCS.1999.814584.

18 Atsuki Momose and Ling Ren. Constant Latency in Sleepy Consensus. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022. doi:10.1145/3548606.3559347.

19 Rafael Pass and Elaine Shi. The Sleepy Model of Consensus. In *Advances in Cryptology – ASIACRYPT 2017*, 2017. doi:10.1007/978-3-319-70697-9_14.

20 Youer Pu, Lorenzo Alvisi, and Ittay Eyal. Safe Permissionless Consensus. In *36th International Symposium on Distributed Computing (DISC 2022)*, volume 246, 2022. doi:10.4230/LIPIcs.DISC.2022.33.

Brief Announcement: Scalable Agreement Protocols with Optimal Optimistic Efficiency

Yuval Gelles
Hebrew University of Jerusalem, Israel

Ilan Komargodski
Hebrew University of Jerusalem, Israel
NTT Research, Sunnyvale, CA, USA

Abstract

Designing efficient distributed protocols for various agreement tasks such as Byzantine Agreement, Broadcast, and Committee Election is a fundamental problem. We are interested in *scalable* protocols for these tasks, where each (honest) party communicates a number of bits which is sublinear in n, the number of parties. The first major step towards this goal is due to King et al. (SODA 2006) who showed a protocol where each party sends only $\tilde{O}(1)$[1] bits throughout $\tilde{O}(1)$ rounds, but guarantees only that $1 - o(1)$ fraction of honest parties end up agreeing on a consistent output, assuming constant $< 1/3$ fraction of static corruptions. Few years later, King et al. (ICDCN 2011) managed to get a full agreement protocol in the same model but where each party sends $\tilde{O}(\sqrt{n})$ bits throughout $\tilde{O}(1)$ rounds. Getting a full agreement protocol with $o(\sqrt{n})$ communication per party has been a major challenge ever since.

In light of this barrier, we propose a new framework for designing efficient agreement protocols. Specifically, we design $\tilde{O}(1)$-round protocols for all of the above tasks (assuming constant $< 1/3$ fraction of static corruptions) with optimistic and pessimistic guarantees:

- **Optimistic complexity**: In an honest execution, all parties send only $\tilde{O}(1)$ bits.
- **Pessimistic complexity**: In any other case, (honest) parties send $\tilde{O}(\sqrt{n})$ bits.

Thus, all an adversary can gain from deviating from the honest execution is that honest parties will need to work harder (i.e., transmit more bits) to reach agreement and terminate. Besides the above agreement tasks, we also use our new framework to get a scalable secure multiparty computation (MPC) protocol with optimistic and pessimistic complexities.

Technically, we identify a relaxation of Byzantine Agreement (of independent interest) that allows us to fall-back to a pessimistic execution in a coordinated way by all parties. We implement this relaxation with $\tilde{O}(1)$ communication bits per party and within $\tilde{O}(1)$ rounds.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Byzantine Agreement, Consensus, Optimistic-Pessimistic, Secure Multi-Party Computation

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.42

Related Version *Full Version*: `https://eprint.iacr.org/2023/751` [12]

Funding This research is supported in part by an Alon Young Faculty Fellowship, by a grant from the Israel Science Foundation (ISF Grant No. 1774/20), and by a grant from the US-Israel Binational Science Foundation and the US National Science Foundation (BSF-NSF Grant No. 2020643).
Ilan Komargodski: Incumbent of the Harry & Abe Sherman Senior Lectureship at the School of Computer Science and Engineering at the Hebrew University.

[1] We use the notation $\tilde{O}(\cdot), \tilde{\Omega}(\cdot)$ to hide poly-logarithmic factors in n.

1 Introduction and Results

We propose a new framework for designing efficient fault-tolerant distributed algorithms for large scale networks. Fault-tolerance means that the functionality of the protocol should not be compromised even if some of the participants in the protocol collude and arbitrarily deviate from the protocol's specification. Basic and well-studied abstractions in this context are Byzantine Agreement (BA) [20, 19], Broadcast, and Committee Election. These serve as building blocks in many distributed protocols. Perhaps most notably, these protocols underlie essentially all secure (cryptographic) multiparty computation protocols [2, 13, 5].

We are interested in designing protocols for all of the above tasks that can be used in large scale settings, where the number of parties could potentially be really large. Let n be the number of parties involved. For protocols to be scalable, we want the amount of work or communication required by each party to be sub-linear in n. Even for the very basic tasks, such as BA, this question was out of reach for many years. The situation changed with the groundbreaking work of [18] that gave a technique for "almost everywhere" *scalable* agreement. Using their technique, it is possible to solve BA, Broadcast, and Committee Election in a scalable fashion with the caveat that only $1 - o(1)$ fraction of the honest parties agree on the output of the protocol. In terms of efficiency though, their protocol is essentially optimal: it requires only $\tilde{O}(1)$ communication rounds and $\tilde{O}(1)$ communication per party.

The model that is considered in [18] is point-to-point synchronous communication and with a computationally unbounded Byzantine (a.k.a malicious) adversary that controls a $(1/3 - \epsilon)$-fraction of the parties for any $\epsilon > 0$. Corruptions occur statically, namely, before the protocol begins (but after it is specified). Further, the adversary has full-information, namely, it sees all messages sent, even messages sent between two honest party, and it is rushing, namely, it gets to send its messages after seeing the honest party's messages for that round. This is the model that we consider in this work as well.

Since [18]'s work, there has been an effort to boost their almost everywhere agreement property into full agreement in various different settings [16, 17, 15, 3]. In the classical information theoretic setting, the work of [15] managed to get full agreement protocols for all of the above tasks at the cost of increased overhead: each party's communication is $\tilde{O}(\sqrt{n})$ bits (the round complexity remains $\tilde{O}(1)$). More than a decade since this work, it is still essentially the state of the art (excluding protocols relying on cryptographic assumptions). The work of [14] proved a lower bound saying that $\tilde{\Omega}(\sqrt[3]{n})$ communication is necessary for at least one party for a certain (non-trivial) class of protocols. See Section 2 for more details. Thus, it is still major open problem to get a full agreement protocol with $o(\sqrt{n})$ communication per party.

Optimistic/pessimistic efficiency

Given the lack of progress in getting better agreement protocols in the worst-case, we suggest a new "beyond worst-case" approach for designing protocols. Specifically, we consider protocols that could have different complexities, depending on the attacker's actions. It is important to emphasize that we want protocols that are correct/secure no matter what; i.e., all honest parties terminate and full agreement is reached. Thus, all an adversary can gain from deviating from the honest execution is that honest parties will need to work harder (i.e., transmit more bits) to reach agreement and terminate. So, it is conceivable to assume that in some applications an attacker will have no incentive to execute an attack. That is, if all the adversary cares about is breaking correctness/security (and gains nothing from a delay), then there is no point to deviate from the protocol and much better efficiency is obtained in this case. We state our main result next.

▶ **Theorem 1** (Scalable agreement compiler). *Let $X \in \{\mathit{ByzantineAgreement}, \mathit{Broadcast}, \mathit{CommitteeElection}\}$ be a task. Assume that there is a protocol Π for X tolerating $1/3 - \epsilon$ fraction of static corruptions for any $\epsilon > 0$. Then, there is another protocol Π' for X (tolerating $1/3 - \epsilon$ fraction of static corruptions for any $\epsilon > 0$) with the following features:*

Optimistic Complexity: *In an honest execution, each party sends $\tilde{O}(1)$ bits and the protocol terminates within $\tilde{O}(1)$ rounds.*

Pessimistic Complexity: *In the worst-case, the communication and round complexities of Π' is the same as that of Π plus $\tilde{O}(1)$.*

We remark that while we stated above that our protocols obtain optimistic efficiency ($\tilde{O}(1)$ communication overall per party) in honest executions, in fact, this is the case even if crash failures are allowed (i.e., an adversary can crash corrupt nodes).

As a corollary of Theorem 1, using the protocols of [18, 15] as Π, we get the following result.

▶ **Corollary 2** (Scalable agreement instantiation). *There are $\tilde{O}(1)$-round Byzantine Agreement, Committee Election, and Broadcast protocols tolerating $1/3 - \epsilon$ fraction of static corruptions for any $\epsilon > 0$ with the following features:*

Optimistic Complexity: *In an honest execution, each party sends $\tilde{O}(1)$ bits.*

Pessimistic Complexity: *In the worst-case, each honest party sends $\tilde{O}(\sqrt{n})$ bits.*

The $\tilde{O}(\sqrt{n})$ term in the pessimistic complexity bullet comes from the cost of [15]'s protocol. Any improvement on the latter will immediately translate to improved pessimistic complexity.

Technical highlight: agreement with error detection. The main building block of the above theorem is a new protocol for a relaxed variant of an agreement functionality. Recall that in an agreement functionality the goal is to guarantee that all parties terminate and agree on some specified value. We introduce a relaxation of the above requirement by requiring that (1) in an honest execution, indeed all parties terminate and reach agreement, but (2) in all other cases, all parties should agree on a special Fail symbol. With this abstraction, we can identify failure, fallback and invoke another (expensive) agreement protocol. To get Theorem 1, we implement such a relaxed agreement protocol using a protocol with $\tilde{O}(1)$ rounds and per-party communication.

Generic optimistic/pessimistic framework. More generally, we use our agreement with error detection protocol to obtain a generic framework for combining efficient optimistic protocols with less efficient pessimistic protocols. Specifically, we manage to combine an efficient protocol Π_{light} (solving a given task X) where it is guaranteed that a failure was noticed by at least one party with a less efficient protocol Π_{heavy} (solving the same task X) that is executed only if everyone knows that some failure occurred. The complexity of the combined protocol is inherited from Π_{light} in an honest execution and from Π_{heavy} in any other case.

Application to Scalable MPC

The above agreement tasks can be thought of as special cases of secure multi-party computation (MPC) [13, 2, 5]. MPC protocols enable a set of mutually distrusting parties to compute a function on their private inputs, while guaranteeing various properties such as correctness, privacy, independence of inputs, and more. We consider the problem of scalable MPC in the peer-to-peer synchronous communication model with private channels. (We emphasize that only this result relies on private channels.)

Feasibility results for (non-scalable) MPC have been long known, e.g., the BGW [2] protocol gives a method for computing an arbitrary function with communication cost that grows multiplicatively with the circuit size of the function and some polynomial in the number of parties. That is, the communication complexity of each of the n parties is $s \cdot \mathsf{poly}(n)$ when computing a function represented as a circuit of size s.[2] The question of scalable MPC, i.e., protocols where the dominant term in the complexity is just the circuit size, is still an active topic and modern results achieve MPC protocols with strong security guarantees and communication complexity $\tilde{O}(s + \mathsf{poly}(n))$ (see Section 2 for an overview and references).

We use our scalable agreement protocols to obtain a new generic scalable MPC. The properties of the resulting protocol are summarized in the next theorem.

▶ **Theorem 3** (Scalable MPC). *There is a statistically maliciously secure MPC protocol tolerating $1/3 - \epsilon$ fraction of static corruptions for any $\epsilon > 0$ with the following features. Given a circuit of size s and depth d over n inputs, the protocol has the following complexity:*
Optimistic Complexity: *In an honest execution, each party sends $\tilde{O}(s/n)$ bits.*
Pessimistic Complexity: *In the worst-case, each party sends $\tilde{O}(s/n + \sqrt{n})$ bits.*
Round Complexity: *All parties terminate after $\tilde{O}(d)$ rounds.*

The additive $\tilde{O}(\sqrt{n})$ term in the pessimistic complexity bullet comes generically from Theorem 1. At a high level, the above MPC is obtained by using our agreement protocol to generate a *quorum*: assign to each party its own representative (small and balanced) committee where there is a strong majority of honest parties. Then, we distribute the gates of the circuit to these committees. Each gate is evaluated by its assigned committees using some standard MPC (e.g., BGW). We emphasize that our protocol has the appealing feature that in an honest execution the total communication complexity is essentially equal to the circuit size, for any circuit, and it is split in a balanced manner across parties. This idea largely appeared in the scalable MPC protocol of [9]. The main difference is that we obtain optimistic/pessimistic complexity, while they only had pessimistic complexity, and this is due to the use of Theorem 1 instead of the protocol of [15].

2 Background and Related Work

Scalable agreement

The BA problem was introduced in the landmark work of Lamport, Shostak, and Pease [19]. In the following couple of decades, several protocols were presented (e.g., [10]) but they all had quadratic total overhead, that is, every party had to essentially communicate with every other party. The protocol of [18] was the first to break this barrier, but it had the caveat of almost-everywhere agreement (rather than full agreement). Extending their almost-everywhere agreement to full agreement in a scalable manner and with minimal cost is still an exciting challenge. We mention some of the key papers addressing this challenge.

First, [16] presented a protocol that satisfies full agreement but it is not balanced. That is, while most parties do communicate a sublinear amount of bits in n overall, there are few parties that communicate essentially with everyone. Several follow up works (e.g., [4, 1]) suffer from the same issue. Then, [15] presented a protocol that satisfies full agreement and it is balanced, but this comes with an extra $\tilde{O}(\sqrt{n})$ term in the communication cost.

[2] Interestingly, in BGW it was already observed that their protocol has an optimistic/pessimistic flavor where in the former the polynomial in n is slightly better than in the pessimistic case.

The extra cost in efficiency is partially explained by an $\Omega(\sqrt[3]{n})$ lower bound on the communication complexity of at least one party in any BA protocol with full agreement, due to [14]. This lower bound, however, applies only to protocols with *static filtering*. In static filtering, every party decides on the set of parties it will listen to before the beginning of each round (as a function of its internal view at the end of the previous round). It is an intriguing open problem to extend the lower bound beyond protocol with static filtering rules.

Lastly, we mention a work of [3]. They assume cryptographic and trusted setup assumptions and further that the adversary is computationally bounded. Also, they assume dynamic filter – namely, the decision of which message is received can be based on the content of received messages (in their case, every message is checked if it contains a valid digital signature). With these relaxations of the model, no lower bound is known. They showed a communication-optimal protocol: only $\tilde{O}(1)$ bits of communication per party are needed to reach full agreement.[3]

We remark that all of the above works, as well as ours, assume near-optimal resilience, i.e., up to $(1/3-\epsilon)$ fraction of corruptions (i.e., near-optimal resilience range). Less than $n/3-1$ corruptions is strictly necessary due to lower bounds of [19, 11] (unless further assumptions are made such as a trusted setup or a computationally bounded adversary).

Scalable MPC

There has been a rich line of work on scalable MPC protocols. The main goal is to design protocols where the total communication complexity scales like $O(s+\mathsf{poly}(n))$ for securely computing a size s circuit by n parties. This was studied in the context of perfect or statistical security and optimal resilience (up to $n/3-1$ or $n/2-1$ corrupted parties) e.g., [2, 13], or with perfect or statistical security and near-optimal resilience (up to $(1/3-\epsilon)$ or $(1/2-\epsilon)$ fraction of corrupted parties) e.g., [9, 6]. In all of these works a broadcast channel is assumed but its usage is limited to a number of times which is independent of the circuit size. All of these works obtain somewhat stronger notions of security than what we obtain in Theorem 3 (e.g., they often tolerate adaptive corruptions while we handle only static ones). Note that we can use our broadcast protocol from Theorem 1 to instantiate the broadcast channel in the above works, achieving statistical security for $(1/3-\epsilon)$ fraction of static corruptions.

Most related to us are the works [7, 9]. In these works the authors used the full agreement protocol of [15] to get a scalable MPC with complexity $\tilde{O}(s/n+\sqrt{n})$ to compute a size s circuit by n parties, per party. The corruption model is $(1/3-\epsilon)$ fraction static corruptions, same as ours. Our MPC prootcol is very similar to theirs (associating the wires of the circuit to quorum members); but, our description is somewhat simpler because we use generic maliciously secure MPC as black-box whereas they sometime use internal building blocks such as verifiable secret sharing. The optimistic/pessimistic aspect is new to our work. Lastly, we mention the work of [8] who studied scalable MPC protocols in an asynchronous setting.

[3] [3] have two variants presenting tradeoffs between the cryptographic assumptions and the trusted setup assumptions. Either a weaker trusted setup assumption (a public-key infrastructure and a common random string) and a stronger cryptographic assumption (SNARKs with linear-time extraction and a collision resistant hash).

References

1 Ittai Abraham, T.-H. Hubert Chan, Danny Dolev, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. Communication complexity of byzantine agreement, revisited. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC*, pages 317–326, 2019.

2 Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, STOC*, pages 1–10, 1988.

3 Elette Boyle, Ran Cohen, and Aarushi Goel. Breaking the $O(\sqrt{n})$-bit barrier: Byzantine agreement with polylog bits per party. In *ACM Symposium on Principles of Distributed Computing, PODC*, pages 319–330, 2021.

4 Nicolas Braud-Santoni, Rachid Guerraoui, and Florian Huc. Fast byzantine agreement. In *ACM Symposium on Principles of Distributed Computing, PODC*, pages 57–64, 2013.

5 David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (abstract). In *Advances in Cryptology - CRYPTO*, volume 293, page 462, 1987.

6 Ivan Damgård, Yuval Ishai, Mikkel Krøigaard, Jesper Buus Nielsen, and Adam D. Smith. Scalable multiparty computation with nearly optimal work and resilience. In *Advances in Cryptology - CRYPTO*, pages 241–261, 2008.

7 Varsha Dani, Valerie King, Mahnush Movahedi, and Jared Saia. Breaking the $O(mn)$ bit barrier: Secure multiparty computation with a static adversary. In *8th Student Conference*, page 64, 2012.

8 Varsha Dani, Valerie King, Mahnush Movahedi, and Jared Saia. Quorums quicken queries: Efficient asynchronous secure multiparty computation. In *Distributed Computing and Networking - ICDCN*, pages 242–256, 2014.

9 Varsha Dani, Valerie King, Mahnush Movahedi, Jared Saia, and Mahdi Zamani. Secure multi-party computation in large networks. *Distributed Computing*, 30:193–229, 2017.

10 Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM J. Comput.*, 12(4):656–666, 1983.

11 Michael J. Fischer, Nancy A. Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Comput.*, 1(1):26–39, 1986.

12 Yuval Gelles and Ilan Komargodski. Scalable agreement protocols with optimal optimistic efficiency. Cryptology ePrint Archive, Paper 2023/751, 2023.

13 Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, STOC*, pages 218–229, 1987.

14 Dan Holtby, Bruce M. Kapron, and Valerie King. Lower bound for scalable byzantine agreement. *Distributed Comput.*, 21(4):239–248, 2008.

15 Valerie King, Steven Lonargan, Jared Saia, and Amitabh Trehan. Load balanced scalable byzantine agreement through quorum building, with full information. In *Distributed Computing and Networking - ICDCN*, pages 203–214, 2011.

16 Valerie King and Jared Saia. From almost everywhere to everywhere: Byzantine agreement with õ($\mathrm{n}^{3/2}$) bits. In *Distributed Computing, 23rd International Symposium, DISC*, pages 464–478, 2009.

17 Valerie King and Jared Saia. Breaking the $O(n^2)$ bit barrier: scalable byzantine agreement with an adaptive adversary. In *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC*, pages 420–429, 2010.

18 Valerie King, Jared Saia, Vishal Sanwalani, and Erik Vee. Scalable leader election. In *17th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 990–999, 2006.

19 Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

20 Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

Brief Announcement: Let It TEE: Asynchronous Byzantine Atomic Broadcast with $n \geq 2f + 1$

Marc Leinweber
Institute of Information Security and Dependability (KASTEL), Karlsruhe Institute of Technology (KIT), Germany

Hannes Hartenstein
Institute of Information Security and Dependability (KASTEL), Karlsruhe Institute of Technology (KIT), Germany

Abstract

Asynchronous Byzantine Atomic Broadcast (ABAB) promises simplicity in implementation as well as increased performance and robustness in comparison to partially synchronous approaches. We adapt the recently proposed DAG-Rider approach to achieve ABAB with $n \geq 2f + 1$ processes, of which f are faulty, with only a constant increase in message size. We leverage a small Trusted Execution Environment (TEE) that provides a unique sequential identifier generator (USIG) to implement Reliable Broadcast with $n > f$ processes and show that the quorum-critical proofs still hold when adapting the quorum size to $\lfloor \frac{n}{2} \rfloor + 1$. This first USIG-based ABAB preserves the simplicity of DAG-Rider and serves as starting point for further research on TEE-based ABAB.

2012 ACM Subject Classification Security and privacy → Distributed systems security

Keywords and phrases Byzantine Fault Tolerance, Trusted Execution Environments, Asynchrony

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.43

Funding This work was supported by funding from the topic Engineering Secure Systems of the Helmholtz Association (HGF).

1 Introduction

Atomic Broadcast primitives play a crucial role for Byzantine-fault tolerant (BFT) State Machine Replication (SMR). A prominent example for BFT SMR in the partially synchronous model is PBFT [6]. By use of small Trusted Execution Environments (TEE) that generate and sign unique sequential identifiers on each process, called USIGs, Veronese et al. [12] showed that PBFT's communication complexity can be reduced and the fault tolerance can be increased to $n \geq 2f + 1$ while still tolerating Byzantine faults. The authenticity/integrity of TEEs can be verified remotely and, thus, TEEs are assumed to only fail by crashing. However, as shown by Miller et al. [11], Asynchronous Byzantine Atomic Broadcast (ABAB) outperforms approaches based on the partially synchronous model particularly under faults and tends to show a simpler design. While it is known that any asynchronous crash fault-tolerant algorithm can be compiled to withstand Byzantine faults using TEEs [3, 7], the proposed compilers show either a polynomial or an exponential overhead in runtime. We are interested in a simple and straightforward design of a USIG-enhanced ABAB that does not add further message rounds and only adds a constant number of bits to each message (essentially a counter value and a signature). To this end, we adapt DAG-Rider [10] to provide ABAB with $n \geq 2f + 1$ processes. We give a quick recap on DAG-Rider and explain the adaption *TEE-Rider*. Besides using TEE-based Reliable Broadcast and changing the required quorums from $2f + 1$ to $\lfloor \frac{n}{2} \rfloor + 1$, we leave DAG-Rider unchanged. We show that the quorum-based arguments of DAG-Rider still hold for TEE-Rider.

2 TEE-Rider: Transforming DAG-Rider to $n \geq 2f + 1$

We make use of the following definition of Atomic Broadcast for a set of processes $P, n := |P|$. The processes communicate over authenticated point-to-point links with eventual delivery.

▶ **Definition 1** (Atomic Broadcast). *Each process $p_i \in P$ receives client transactions t via events* clientRequest(t)*. Correct processes deliver tuples (t, r, p_i), where t is a client transaction, $r \in \mathbb{N}_0$ a round number, and $p_i \in P$ the process that initially received t, satisfying the following properties:*

Agreement: *If a correct process $p_i \in P$ delivers (t, r, p_j), then every other correct process $p_k \in P, k \neq i$ eventually delivers (t, r, p_j) with probability 1.*

Integrity: *For each round $r \in \mathbb{N}_0$ and process $p_j \in P$, a correct process $p_i \in P$ delivers (t, r, p_j) at most once.*

Validity: *If a correct process $p_i \in P$ receives an event* clientRequest(t)*, then every correct process $p_k \in P$ eventually delivers (t, r, p_i) with probability 1.*

Total Order: *Let m_1 and m_2 be any two valid tuples that are delivered by any two correct processes $p_i, p_j \in P$. If p_i delivers m_1 before m_2, then p_j delivers m_1 before m_2.*

2.1 Changes in Assumptions, Building Blocks, and Setup

In addition to the assumptions of DAG-Rider, we assume that each process is equipped with a USIG [12] that may only fail by crashing. It implements a signature service that binds a unique counter value to each signature it produces. The USIG is used for Reliable Broadcast with a fault tolerance of $n > f$ as, e.g., implemented in [8, Algorithm 1]: it is a "single echo" algorithm with USIG-signed messages and attached counter value. Correct processes relay a message once and reject messages with invalid USIG signatures or with counter values already received which prevents equivocating messages for the same counter. An instance of the Reliable Broadcast abstraction has two functions: broadcast(r, m) to reliably broadcast exactly one arbitrary message m for round r to all processes in P, and delivered() which returns all messages that were received by the instance since the last call to delivered(). We expect the Reliable Broadcast abstraction to fullfil the following properties:

▶ **Definition 2** (Reliable Broadcast). *A sender $p_s \in P, n := |P|$ can call* broadcast(c, m)*. Correct processes deliver (c, m) where $c \in \mathbb{N}_0$ and m an arbitrary message satisfying the following properties:*

RB-Agreement: *If a correct process $p_i \in P$ delivers (c, m), then every other correct process $p_k \in P, k \neq i$ eventually delivers the same (c, m).*

RB-Integrity: *For each $c \in \mathbb{N}_0$, a correct process $p_k \in P$ delivers (c, m) at most once.*

RB-Validity: *If a correct sender calls* broadcast(c, m)*, then every correct process $p_i \in P$ eventually delivers (c, m).*

Additionally, we assume an asynchronous common coin, e.g. as defined by Cachin et al. [5], that produces a uniformly distributed common random number p out of $\{p \mid p \in \mathbb{N}_0 \colon p < n\}$ for all correct processes and a name $i \in \mathbb{N}_0$ as soon as $f + 1$ processes invoked toss(i); repetitive calls with same the i yield the same p. We further assume a trusted setup of the common coin and the USIGs using a public key infrastructure (to set up the common coin's threshold signature scheme, dealerless variants [4] and those with an asynchronous setup [1] exist).

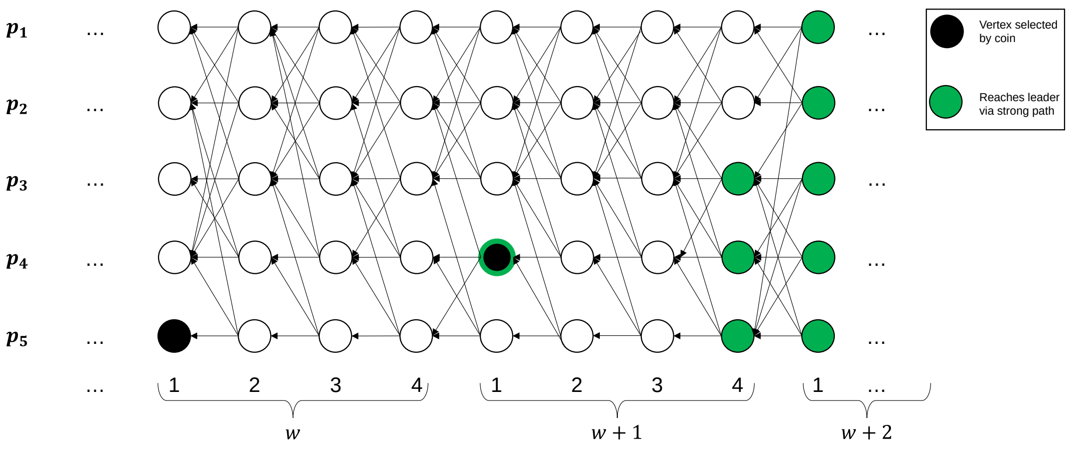


Figure 1 Example DAG for $n = 5$ processes of which at maximum 2 may be faulty. Shown is the "global" state of the graph, i.e., after every process eventually received every vertex. For simplicity, all vertices are valid and weak edges are left out. The direct commit rule is not fulfilled for any process for wave w; it is fulfilled for processes p_3, p_4, and p_5 for wave $w + 1$. The green coloring highlights the effect of the direct commit rule as proven in Lemma 3. The direct commit rule ensures that a correct process can commit a wave retrospectively if it was not able to commit when it finished the wave. Since the leader vertex of wave $w + 1$ has a strong path to the leader vertex of wave w, wave w will be committed retrospectively.

2.2 The Algorithm

DAG-Rider uses n Reliable Broadcast instances to disseminate process messages and to construct a directed acyclic graph (DAG) that captures the communication history of all processes. In a second step, each process derives consensus on the order of transactions using a Common Coin based on the graph structure. The adapted DAG-Rider algorithm executed by a correct process $p_i \in P$ is shown in Algorithm 1. Utility functions are listed in Algorithm 2. The core of the approach is the construction and interpretation of a (local) DAG that captures received transactions and the observed communication sequence between processes. The DAG is structured in rounds and a round contains at maximum one vertex per process, i.e., n vertices. Rounds are addressed in an array style and the local view of a process p_i on the DAG is indicated by an index i. The very first round $DAG_i[0]$ is initialized with n hard-coded "genesis" vertices. A vertex in round r has two types of edges: strong edges point to vertices of round $r-1$ and weak edges point to vertices of any round $r' \leq r-2$. As soon as p_i received $\lfloor \frac{n}{2} \rfloor + 1$ *valid* vertices for a round r, i.e., $\lfloor \frac{n}{2} \rfloor + 1$ vertices referencing $\lfloor \frac{n}{2} \rfloor + 1$ vertices of round $r - 1$ as strong edges ($v.strong$, l. 11), for which it also knows its predecessors (l. 15), p_i will *complete* round r and transition to round $r + 1$. Now, as soon as p_i receives a client transaction, it will become the payload of a vertex v which is created and broadcast by p_i for round $r + 1$ (ll. 44 and 21-27). The vertex v connects to all vertices p_i received for round r (l. 23). If p_i received vertices u for older rounds that are not reachable from the newly created vertex using the transitive closure of strong and weak edges (a "path"), u will become a weak edge of v ($v.weak$, l. 26). The new vertex is broadcast using Reliable Broadcast instance i to all processes (l. 27). Every fourth round a so-called wave, consisting of four rounds, is completed (l. 19) and the DAG structure is used to derive a total order on the transactions (ll. 28-42). Each wave w has exactly one wave leader v which is chosen calling $coin.\text{toss}(w)$ from the vertices of w's first round$(w, 1)$. The random number is used to select the process whose vertex is to be used as wave leader. If v was not (yet) received or there are no $\lfloor \frac{n}{2} \rfloor + 1$ vertices in w's fourth $\mathrm{round}(w, 4)$ that have v in their transitive closure of strong edges (a "strong path"), i.e. the *direct commit rule* is not fulfilled, the wave cannot be committed (l. 30). If wave w can be committed, process p_i checks first if there are wave leaders of waves w' between the last wave that was committed (variable *decidedWave*) and the current wave w that were received in the meantime and are connected to the leader of the wave $w' + 1$ (retrospective commit, ll. 33-36). The wave leaders are used as the root for a deterministic graph traversal to determine the total order of transactions (ll. 38-42). An example for a resulting graph with $n = 5$ processes, i.e. $f \leq 2$, is shown in Figure 1.

3 Correctness Argument

Lemmas 1 and 2 of the original DAG-Rider publication [10] are crucial for Total Order and Agreement and rely on quorum intersection arguments. The following Lemmas 3 and 4 show the corresponding results for a quorum size of $\lfloor \frac{n}{2} \rfloor + 1$. Results for Integrity and Validity simply follow from the original paper.

▶ **Lemma 3.** *If a correct process $p_i \in P$ commits the wave leader v of a wave w when it completes wave w in $\mathrm{round}(w, 4)$, then any valid vertex v' of any process $p_j \in P$ broadcast for a round $r \geq \mathrm{round}(w + 1, 1)$ will have a strong path to v.*

Proof. Since p_i commits v in $\mathrm{round}(w, 4)$, the direct commit rule is fulfilled (l. 30): $\exists U \subseteq DAG_i[\mathrm{round}(w, 4)]\colon |U| \geq \lfloor \frac{n}{2} \rfloor + 1 \wedge \forall u \in U\colon \mathrm{strongPath}(u, v)$. A valid vertex must reference at least $\lfloor \frac{n}{2} \rfloor + 1$ distinct vertices of the previous round with a strong edge (l. 11). Thus, a process $p_j \in P$ broadcasting a valid vertex v_j for $\mathrm{round}(w + 1, 1)$ selected at least $\lfloor \frac{n}{2} \rfloor + 1$ vertices of $\mathrm{round}(w, 4)$ as strong edges for v_j. Any two subsets of size $\lfloor \frac{n}{2} \rfloor + 1$ of a superset of size n intersect at least in one element. Thus, every valid vertex of a process broadcast for $\mathrm{round}(w + 1, 1)$ must have at least one edge to a vertex of U, and, via U to v. As every valid vertex of $\mathrm{round}(w + 1, 1)$ has a strong path to v, and every valid vertex of $\mathrm{round}(w + 1, 2)$ connects to at least $\lfloor \frac{n}{2} \rfloor + 1$ vertices of $\mathrm{round}(w + 1, 1)$, by induction, any valid vertex v' of any process $p_j \in P$ broadcast for a round $r \geq \mathrm{round}(w + 1, 1)$ has a strong path to v. ◀

▶ **Lemma 4.** *When a correct process $p_i \in P$ completes $\mathrm{round}(w, 4)$ of wave w, then $\exists V_1 \subseteq DAG_i[\mathrm{round}(w, 1)], V_4 \subseteq DAG_i[\mathrm{round}(w, 4)]\colon |V_1| \geq \lfloor \frac{n}{2} \rfloor + 1 \wedge |V_4| \geq \lfloor \frac{n}{2} \rfloor + 1 \wedge (\forall v_1 \in V_1, \forall v_4 \in V_4\colon \mathrm{strongPath}(v_4, v_1))$.*

Proof. By use of Reliable Broadcast and validity checks in ll. 11 and 15, faulty processes are limited to omission faults. Thus, the *get-core* argument of Attiya and Welch [2, Sec. 14.3.1] still holds [2, Sec. 14.3.3]: Let $A \in \{0, 1\}^{n \times n}$ be a matrix that contains a row for each possible vertex of $\mathrm{round}(w, 3)$ and a column for each possible vertex of $\mathrm{round}(w, 2)$. Let $A[j, k] = 1$ if the vertex of process p_j of $\mathrm{round}(w, 3)$ has a strong edge to the vertex of process p_k of $\mathrm{round}(w, 2)$ or p_j sends no vertex (or an invalid one) but p_k sends a valid vertex for $\mathrm{round}(w, 2)$. As there are at least $\lfloor \frac{n}{2} \rfloor + 1 \leq n - f$ correct processes, each row of A contains at least $\lfloor \frac{n}{2} \rfloor + 1$ ones and A contains at least $n(\lfloor \frac{n}{2} \rfloor + 1)$ ones. Since there are n columns, there must be a column l with at least $\lfloor \frac{n}{2} \rfloor + 1$ ones. This implies there is a vertex v_l by process p_l in $\mathrm{round}(w, 2)$ s.t. $\exists V_3 \subseteq DAG_i[\mathrm{round}(w, 3)]\colon |V_3| \geq \lfloor \frac{n}{2} \rfloor + 1 \wedge \forall v_3 \in V_3\colon \mathrm{strongPath}(v_3, v_l)$. As at most f vertices in V_3 belong to faulty processes that may commit send omission faults for $\mathrm{round}(w, 3)$ and $\lfloor \frac{n}{2} \rfloor + 1 \geq f + 1$, by quorum section at least one vertex of V_3 is received by any correct process $p_j \in P$ before it sends its vertex for $\mathrm{round}(w, 4)$. Thus, every valid vertex in $DAG_i[\mathrm{round}(w, 4)]$ has at least one strong edge to a vertex of V_3. Since v_l must be valid

Algorithm 1 TEE-Rider pseudocode for process $p_i \in P, n := |P|, \boldsymbol{n \geq 2f+1}$.

```
1: state $DAG$: array of sets of vertices, $DAG[0]$ initialized with “genesis” vertices
2: state $r$: $\mathbb{N}$, initialized with 1
3: state $decidedWave$: $\mathbb{N}_0$, initialized with 0
4: state $transactionsToPropose$: queue of client transactions $t$, initialized empty
5: state $buffer$: set of vertices, initialized empty
6: state $rb$: array of $n$ Reliable Broadcast instances with delivered() and broadcast$(r, m)$
7: state $coin$: common coin instance with toss$(w)$
8: while True do
9:     for $k \leftarrow 0$ up to $n-1$ do
10:        for $m = (r', v) \in rb[k]$.delivered() do
11:            if $|v.strong| < \lfloor \frac{n}{2} \rfloor + 1$ then continue
12:            $v.source \leftarrow p_k; v.round \leftarrow r'; v.delivered \leftarrow$ False
13:            $buffer$.add$(v)$
14:    for $v \in buffer$ do
15:        if $v.round > r \vee \exists u \in v.strong \cup v.weak\colon u \notin \cup_{r' \geq 0} DAG[r']$ then continue
16:        $DAG[v.round]$.add$(v)$
17:        $buffer$.remove$(v)$
18:    if $|DAG[r]| < \lfloor \frac{n}{2} \rfloor + 1$ then continue
19:    if $r \mod 4 = 0$ then waveReady$(\frac{r}{4})$
20:    $r \leftarrow r + 1$
21:    wait until $\neg transactionsToPropose$.isEmpty()
22:    $v \leftarrow$ new vertex
23:    $v.block \leftarrow transactionsToPropose$.dequeue(); $v.strong \leftarrow DAG[r-1]$
24:    for $r' \leftarrow r-2$ down to 1 do
25:        for $u \in DAG[r']$ do
26:            if $\neg$path$(v, u)$ then $v.weak$.add$(u)$
27:    $rb[i]$.broadcast$(r, v)$
28: function $waveReady(w)$
29:    $v \leftarrow$ getWaveLeader$(w)$    ▷ $\bot$ if round$(w, 1)$ vertex of chosen process is not in $DAG$
30:    if $v = \bot \vee |\{u \mid u \in DAG[$round$(w,4)]\colon$ strongPath$(u, v)\}| < \lfloor \frac{n}{2} \rfloor + 1$ then return
31:    $leadersStack \leftarrow$ new stack; $leadersStack$.push$(v)$
32:    for $w' \leftarrow w-1$ down to $decidedWave + 1$ do
33:        $u \leftarrow$ getWaveLeader$(w')$
34:        if $u \neq \bot \wedge$ strongPath$(v, u)$ then
35:            $leadersStack$.push$(u)$; $v \leftarrow u$
36:    $decidedWave \leftarrow w$
37:    while $\neg leadersStack$.isEmpty() do
38:        $v \leftarrow leadersStack$.pop()
39:        $verticesToDeliver \leftarrow \{u \mid u \in \cup_{r'>0} DAG[r']\colon$ path$(v, u) \wedge \neg u.delivered\}$
40:        for $u \in verticesToDeliver$ in deterministic order do
41:            $u.delivered \leftarrow$ True; deliver $(u.block, u.round, u.source)$
42: upon clientRequest $(t)$
43:    $transactionsToPropose$.enqueue$(t)$
```

Algorithm 2 Utility functions pseudocode.

```
function $path(v, u)$ : boolean
    return exists a sequence of vertices $(v_1, v_2, ..., v_k) \in \cup_{r' \geq 0} DAG[r']$ such that
    $v_1 = v \wedge v_k = u \wedge \forall i \in [2, k]: v_i \in v_{i-1}.strong \cup v_{i-1}.weak$
function $strongPath(v, u)$ : boolean
    return exists a sequence of vertices $(v_1, v_2, ..., v_k) \in \cup_{r' \geq 0} DAG[r']$ such that
    $v_1 = v \wedge v_k = u \wedge \forall i \in [2, k]: v_i \in v_{i-1}.strong$
function $getWaveLeader(w)$ : vertex or $\perp$
    $p_j \leftarrow coin.\text{toss}(w)$
    if $\exists v \in DAG[\text{round}(w, 1)]: v.source = p_j$ then return $v$
    return $\perp$
function $round(w, i)$ : $\mathbb{N}$
    return $4(w - 1) + i$
```

and thus has a strong edge to each vertex of a set $V_1 \subseteq DAG_i[\text{round}(w, 1)], |V_1| \geq \lfloor \frac{n}{2} \rfloor + 1$, any valid vertex of rounds $r \geq \text{round}(w, 4)$ has a strong path to every vertex, including V_1, reached by v_l via strong paths. Please note that the construction of the set V_1 is valid for all correct processes that complete the wave and, thus, represents the "common core". ◀

4 Discussion and Conclusion

DAG-Rider shows the power of causal order broadcast to implement consensus. The adaption for TEEs preserves the simplicity of DAG-Rider while increasing its fault tolerance and reducing the communication effort (i.e., from "double echo" to "single echo" Reliable Broadcast). The ease of adaption of DAG-Rider for TEEs make it a perfect textbook example for TEE-based ABAB. Follow-up work to DAG-Rider, Tusk [9], addresses a major deployability issue, namely garbage collection, shortens the wave length, and replaces the underlying Reliable Broadcast with a communication scheme that leverages the graph structure to achieve linear communication complexity in the happy case. Additionally, to the best of our knowledge, there exists no TEE-based, dealerless, and asynchronous common coin primitive. In summary, investigating a TEE-based dealerless setup as well as transforming the follow-ups of DAG-Rider for empirical studies to investigate the assumed superiority of asynchronous TEE-based approaches, e.g., in comparison to MinBFT [12], is a promising line of research.

References

1 Ittai Abraham, Marcos Kawazoe Aguilera, and Dahlia Malkhi. Fast asynchronous consensus with optimal resilience. In Nancy A. Lynch and Alexander A. Shvartsman, editors, *Distributed Computing*, volume 6343 of *Lecture Notes in Computer Science*, pages 4–19. Springer, 2010. `doi:10.1007/978-3-642-15763-9_3`.

2 Hagit Attiya and Jennifer L. Welch. *Distributed computing - fundamentals, simulations, and advanced topics (2. ed.)*. Wiley, 2004. `doi:10.1002/0471478210`.

3 Naama Ben-David, Benjamin Y. Chan, and Elaine Shi. Revisiting the power of non-equivocation in distributed protocols. In Alessia Milani and Philipp Woelfel, editors, *PODC '22: ACM Symposium on Principles of Distributed Computing, Salerno, Italy, July 25 - 29, 2022*, pages 450–459. ACM, 2022. `doi:10.1145/3519270.3538427`.

4 Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In Yvo Desmedt, editor, *Public Key Cryptography — PKC 2003*, volume 2567 of *Lecture Notes in Computer Science*, pages 31–46. Springer, 2003. doi:10.1007/3-540-36288-6_3.

5 Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *J. Cryptol.*, 18(3):219–246, 2005. doi:10.1007/s00145-005-0318-0.

6 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002. doi:10.1145/571637.571640.

7 Allen Clement, Flavio Junqueira, Aniket Kate, and Rodrigo Rodrigues. On the (limited) power of non-equivocation. In Darek Kowalski and Alessandro Panconesi, editors, *ACM Symposium on Principles of Distributed Computing, PODC '12, Funchal, Madeira, Portugal, July 16-18, 2012*, pages 301–308. ACM, 2012. doi:10.1145/2332432.2332490.

8 Miguel Correia, Giuliana Santos Veronese, and Lau Cheuk Lung. Asynchronous byzantine consensus with 2f+1 processes. In Sung Y. Shin, Sascha Ossowski, Michael Schumacher, Mathew J. Palakal, and Chih-Cheng Hung, editors, *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, March 22-26, 2010*, pages 475–480. ACM, 2010. doi:10.1145/1774088.1774187.

9 George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and Tusk: a DAG-based mempool and efficient BFT consensus. In Yérom-David Bromberg, Anne-Marie Kermarrec, and Christos Kozyrakis, editors, *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*, pages 34–50. ACM, 2022. doi:10.1145/3492321.3519594.

10 Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is DAG. In Avery Miller, Keren Censor-Hillel, and Janne H. Korhonen, editors, *PODC '21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021*, pages 165–175. ACM, 2021. doi:10.1145/3465084.3467905.

11 Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of BFT protocols. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proc. 2016 ACM SIGSAC Conf. on Computer and Communications Security, Vienna, Austria, 2016*, pages 31–42. ACM, 2016. doi:10.1145/2976749.2978399.

12 Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Veríssimo. Efficient byzantine fault-tolerance. *IEEE Trans. Computers*, 62(1):16–30, 2013. doi:10.1109/TC.2011.221.

Brief Announcement: Recoverable and Detectable Self-Implementations of Swap

Tomer Lev Lehman
Department of Computer Science, Ben Gurion University, Beer Sheva, Israel

Hagit Attiya
Department of Computer Science, Technion, Haifa, Israel

Danny Hendler
Department of Computer Science, Ben Gurion University, Beer Sheva, Israel

Abstract

Recoverable algorithms tolerate failures and recoveries of processes by using non-volatile memory. Of particular interest are *self-implementations* of key operations, in which a recoverable operation is implemented from its non-recoverable counterpart (in addition to reads and writes).

This paper presents two self-implementations of the SWAP operation. One works in the *system-wide failures* model, where all processes fail and recover together, and the other in the *independent failures* model, where each process crashes and recovers independently of the other processes.

Both algorithms are wait-free in crash-free executions, but their recovery code is blocking. We prove that this is inherent for the independent failures model. The impossibility result is proved for implementations of *distinguishable* operations using *interfering* functions, and in particular, it applies to a recoverable self-implementation of swap.

2012 ACM Subject Classification Theory of computation → Shared memory algorithms

Keywords and phrases Persistent memory, non-volatile memory, recoverable objects, detectablitly

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.44

Related Version *Full Version*: `https://arxiv.org/abs/2308.03485`

Funding Partially supported by the Israel Science Foundation (grants 380/18 and 22/1425).
Tomer Lev Lehman: Partially supported by the Lynne and William Frankel Center for Computer Science.

1 Introduction

Recent years have seen a rising interest in the failure-recovery model for concurrent computing. This model captures an unstable system, where processes may crash and recover, and it has two variants: In the *system-wide failure* model (also called the *global-crash* model), all processes fail simultaneously and a single process is responsible for the recovery of the whole system. In the *independent failures* model (also called the *individual-crash* model), each process can incur a crash independently of other processes and recovers independently. *Recoverable* algorithms, tolerating failures and recoveries, have been presented for various concurrent data structures, for both the system-wide model [5, 7, 9, 14, 18, 20, 21] and the independent-failure model [1, 3, 5, 18, 20].

The correctness of a recoverable algorithm can be specified in several ways. *Durable Linearizability* [16] intuitively requires linearizability of all operations that survive the crashes. *Detectability* [9] ensures that upon recovery, it is possible to infer whether the failed operation took effect or not and, in the former case, obtain its response. *Nesting-safe Recoverable Linearizability* (NRL) [1], defined for the independent failures model, ensures detectability and linearizability. It also allows the nesting of recoverable objects. By providing implementations of NRL primitive objects, a programmer can combine several of these primitives to create recoverable implementations of complex higher-level objects and algorithms. This level of abstraction can be helpful in the adoption of recoverable algorithms.

To facilitate high-level implementations of complex NRL objects it is helpful to introduce implementations of low-level base NRL objects. An attractive approach to designing low-level based NRL objects is through *self-implementations* [20], in which a recoverable operation is implemented with instances of the same primitive operation, possibly with additional reads and writes on shared variables. This approach ensures that when using the recoverable version of an operation, the system must only support its hardware-implemented counterpart.

NRL self-implementations already exist for various primitives, including *read*, *write*, *test&set*, and *compare&swap* [1, 3], as well as *fetch&add* [20]. A universal construction [3] using NRL read, write and *compare&swap* objects builds upon previously-introduced self-implementations of NRL objects to take any concurrent program with read, write and CAS, and make it recoverable while adding only constant computational overhead.

This paper presents the first NRL self-implementations of *swap*, for both the system-wide and the independent failures models. Swap is a widely-used primitive that is employed by concurrent algorithms. Our implementations borrow ideas from the *recoverable mutual exclusion (RME)* [12] algorithms of [11, 17], which use a similar approach to overcome swap failures. Unlike these algorithms, however, our implementations are also challenged with the task of satisfying wait-freedom and linearizability.

Both our algorithms are wait-free in crash-free executions, while the recovery code in both is blocking.

We present an impossibility proof for implementing a class of *distinguishable operations* using a set of *interfering functions* [13] in a recoverable *lock-free* fashion in the independent failures model. In particular, this result applies to self-implementations of swap, but it also holds for, e.g., implementing swap using fetch-and-add and swap combined. Other distinguishable operations to which this proof applies are the *deque* of a queue object and the *pop* of a stack object. Our impossibility result unifies and extends specific results for self-implementations of *test&set* [1] and *fetch&add* [20]. Another related impossibility result addresses recoverable consensus in the independent failures model [10].

Several previous papers introduce general transformations to port existing algorithms and make them persistent, e.g., [3, 4, 6, 8, 15]. Most of these transformations use strong primitives such as *compare&swap* while their non-recoverable counterparts may use weaker primitives, in terms of their consensus number [13]. We believe future research may use our self-implementation of swap to extend general constructions such as [3] mentioned above to programs that also use swap as a primitive.

Similarly to NRL, *detectable sequence specifications* (DSS), introduced by Li and Golab [18], formalizes the notion of detectability. The DSS-based approach is more portable and less reliant on system assumptions in comparison to NRL, but delegates the responsibility for nesting to application code.

2 Model, In Brief

We use a simplified version of the NRL system model [1]. There are n asynchronous *processes* $p_1, \ldots, p_n$, which communicate by applying atomic *primitive* read, write and read-modify-write operations to *base objects*. The state of each process consists of non-volatile *shared variables*, which serve as base objects, as well as volatile *local variables*. A *crash-failure* (or simply a *crash*) can occur at any point during the execution. A crash resets all local variables to arbitrary values but preserves the values of all non-volatile variables.

A process p *invokes an operation Op* on an object by performing an *invocation step*. *Op completes* by executing a *response step*, in which *the response of OP is stored to a local volatile variable of p*. It follows that the response value is lost if p incurs a crash before *persisting* it, that is, before writing it to a non-volatile variable.

In the *independent failures* model a *recoverable operation* Op is associated with a *recovery procedure* Op.RECOVER that is responsible for completing Op upon recovery from a crash. Following a crash of process p that occurs when p has a pending recoverable operation op_p, the system eventually resurrects p by invoking the recovery procedure of the failed op_p.

As proven by [2], detectable algorithms for the NRL model must keep an auxiliary state that is provided from outside the operation, either via operation arguments or via a non-volatile variable accessible by it. We assume that Op.RECOVER has access to a designated per-process non-volatile variable storing the sequence number of Op which is incremented before each operation invocation.

For the *system-wide failures* model in which all processes crash simultaneously, the system recovers by executing a parameterless *global recovery procedure* called Op.GRECOVER. Once Op.GRECOVER completes, the system resurrects each of the failing processes for performing an *individual recovery procedure for Op*, called Op.RECOVER.

3 Detectable Swap Algorithms

A swap object supports the *SWAP(val)* operation, which atomically swaps the object's current value *cur* to *val* and returns *cur*. A key challenge to overcome when implementing a detectable swap object from read, write, and primitive swap operations is that the return values of one or more primitive swap operations may be lost upon a system-wide failure that occurs before the operations are persisted. These non-persisted operations may have already affected the state of the swap object and, moreover, operations by other processes may have already returned the values written by these primitive operations. To ensure linearizability, the implementation must identify such operations and handle them correctly.

The return value of each SWAP operation must be the input of another SWAP operation (or the initial value of the swap object). Furthermore, the operand swapped in by one SWAP operation can be returned by at most a single other SWAP operation.

To ensure linearizability, the real-time order between non-overlapping operations must be preserved, as illustrated in Figure 1. This scenario involves six processes, $p_1, \ldots p_6$, performing eight SWAP operations, $op_0, \ldots op_7$. A system-wide crash occurs when operations op_0, op_2, op_4, op_6 have already been completed (hence their return values are specified), while operations op_1, op_4, op_5, op_7 are pending. Note that operations op_1, op_4, op_5, although not completed, have surely affected the global state of the swap object as their inputs are the return values of other operations, while op_7 (pending as well) might or might not have affected the object's state.

There are several ways the system may recover in order to produce a correct linearizable result. In all of them, op_1 must return 0. The remaining operations might return different values in the following ways: (1) op_4 returns 2, op_5 returns 3, and op_7 returns 6. (2) op_7 returns 2, op_4 returns 7, and op_5 returns 3. (3) op_4 returns 2, op_7 returns 3, and op_5 returns 7. There are several possible linearizations in this example, because op_7 may be linearized in several ways since its effect on the global state is unknown.

We represent the order of SWAP operations as a linked list of Node structures, the end of which is pointed to by a *tail* variable manipulated with primitive swaps. The list starts with a sentinel node called *headNode*, which holds the object's initial value (denoted $\perp$).

Each Node structure represents a single SWAP operation and stores a pointer *prev* to the node of its predecessor operation and the SWAP's operand *val*. The order of SWAP operations is reflected by the order of the Node structures in the list. By doing so, each Node points to the previous Node structure that represents the previous SWAP operation, hence, the operation's return value will be *Node.prev.val*.

A problem occurs if a process successfully swaps its Node into the list but fails before pointing from its structure to the previous Node. This type of failure may create *fragments* in the list representing the SWAP operations. Thus, instead of a single complete list, crashes may result in several incomplete disconnected lists. In order to reconnect these fragments back to a complete list, our algorithm goes over all previously-announced operations upon recovery and recreate a correctly-ordered complete list of operations.

A similar challenge occurs in the RME algorithms of Golab and Hendler [11] and Jayanti et al. [17]. These algorithms mend fragments of the linked-list based queue, used in the MCS lock [19], which are created when failures occur just before or after primitive swap operations.

Our algorithm needs to address two additional challenges, however. First, the SWAP operations of our algorithm should be wait-free, whereas RME implementations are allowed to block. Second, unlike the RME implementations, our algorithm should provide linearizability. Specifically, the order of list fragments, constructed during recovery, must respect the real-time order between SWAP operations.

We address these challenges by employing a *fragment ordering* scheme, which we view as the key algorithmic novelty of our algorithms. The scheme encapsulates the critical steps of each SWAP operation by two vector timestamp computations. Based on the resulting timestamps, the recovery code ensures the following invariant: if a fragment A contains a Node n_A that was created after an operation associated with a Node n_B on fragment B was completed, then fragment B will be ordered after fragment A in the connected list. (Note that *prev* pointers define the *reverse order* between operations.)

Figure 2 presents a set of fragments that may be generated immediately after the system-wide crash ending the execution depicted in Figure 1. As specified, when ordering fragments, the algorithm uses vector timestamps for maintaining linearizability. As an example, consider a linked list, reconnecting the fragments of Figure 2, in which $op_4.prev \leftarrow op_0$, $op_1.prev \leftarrow op_3$, $op_7.prev \leftarrow op_2$ and $op_5.prev \leftarrow op_7$. Although this list contains the Nodes of all operations from tail to head, it violates linearizability because op_3 is ordered after op_2 although it follows it in real-time order. By using the two vector timestamps, our algorithm is able to order the fragments so that linearizability is maintained.

The full version presents the details of the algorithm and its correctness proof, proving the next theorem:

▶ **Theorem 1.** *There is an algorithm that implements a recoverable NRL SWAP in the system-wide failures model using only read, write and primitive Swap operations. The SWAP operations are wait-free.*

For the independent failures model, a recoverable algorithm must allow one or more processes to execute its recovery code concurrently, while other processes may concurrently execute their SWAP operations. In order to handle this concurrency correctly, we introduce two key changes to the system-wide failures algorithm. First, the recovery procedure now

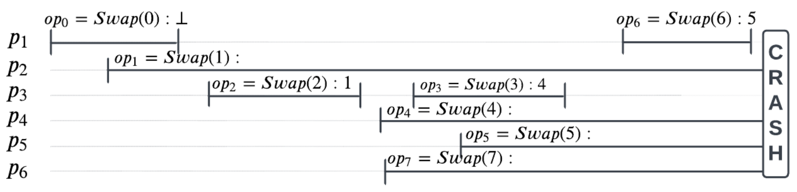


Figure 1 An example of the effect of a system-wide failure.

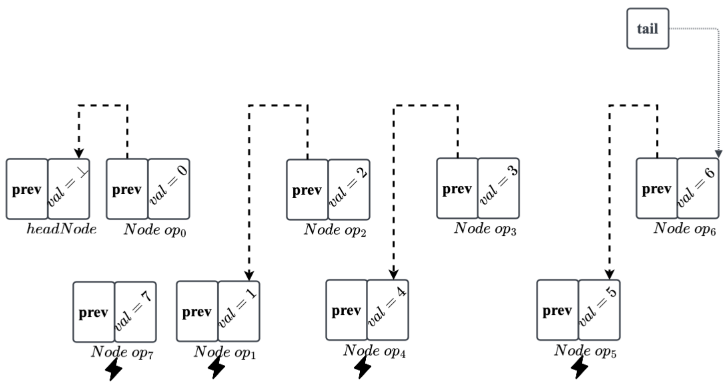


Figure 2 List fragments existing after the execution described in Figure 1, immediately after a system-wide crash. Operations 1,4,5 crashed after swapping their Nodes to *tail* but before persisting their pointer to their predecessor Node.

synchronizes concurrent invocations by using a starvation-free RME lock, effectively serializing the execution of the recovery code. We employ the RME lock of Golab and Ramaraju [12], which uses only reads and writes. The second change allows the recovery code to wait for a concurrent SWAP operation *Op* to either complete or crash. Only once this happens, can the recovery code continue. The full version presents the details of the algorithm and its correctness proof, proving the following theorem:

▶ **Theorem 2.** *There is an algorithm that implements a recoverable NRL SWAP in the independent failures model using only read, write and primitive Swap operations. The SWAP operations are wait-free.*

4 Impossibility of lock-freedom for the independent failures model

We prove a theorem establishing the impossibility of implementing lock-free algorithms for a wide variety of recoverable objects under the independent failures model. This result generalizes previous results [1, 20], to a wider family of operations and implementations.

The result applies to *distinguishable operations*: Informally, an operation *Op* is distinguishable if it can be invoked with two operands, $x \neq y$, such that the return values of these invocations allows the system to determine which operation was applied first. The proof applies when implementations use only read, write, and a set of *interfering functions* [13], which are functions that either commute or overwrite. (See the full version.)

▶ **Theorem 3.** *There is no recoverable implementation of a distinguishable operation M from read, write, and interfering primitive operations in the independent failures model, such that both M and M.RECOVER are lock-free.*

Consider a swap object with initial value 0; when SWAP(1) and SWAP(2) are applied sequentially, only the first operation applied returns 0. This shows that SWAP is a distinguishable operation. Note also that a primitive swap is overwriting, since applying it twice overwrites the first application. Thus, the theorem implies that there is no recoverable self-implementation of SWAP, where both SWAP and SWAP.RECOVER are lock-free. This shows that the mutual exclusion lock in our algorithm for the individual failures model cannot be avoided.

It is also possible to show that the pop operation of a stack object, and the deque operation of a queue object, as well as *fetch&add* and *test&set*, are distinguishable.

5 Discussion

We present two NRL self-implementations of the swap object, one for the system-wide failures model and the other for the independent failures model. In both, SWAP operations are wait-free and the recovery code is blocking. In the system-wide failures model, this is a result of delegating the recovery to a single process, while in the independent failures model, it is due to coordination between the recovering process and the other processes. We also prove the impossibility of a lock-free implementation of distinguishable operations using read-write and a set of interfering functions, in the independent failures model. In particular, this shows that with independent failures, a self-implementation of swap cannot be lock-free.

Our algorithms use $O(m * n)$ space, where m is the number of SWAP invocations in the execution. Bounding memory consumption to $O(n)$ is relatively easy if a recoverable swap operation by one process can wait for operations by other processes to either make progress or fail. An interesting open question is to figure out whether the space complexity of detectable swap self-implementations with wait-free operations can be reduced to $o(m)$ or if $\Omega(m)$ is inherently required. We leave this question for future work.

References

1 Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. Nesting-safe recoverable linearizability: Modular constructions for non-volatile memory. In *ACM Symposium on Principles of Distributed Computing*, pages 7–16, 2018.

2 Ohad Ben-Baruch, Danny Hendler, and Matan Rusanovsky. Upper and lower bounds on the space complexity of detectable objects. In *39th ACM Symposium on Principles of Distributed Computing*, pages 11–20, 2020.

3 Naama Ben-David, Guy E Blelloch, Michal Friedman, and Yuanhao Wei. Delay-free concurrency on faulty persistent memory. In *31st ACM Symposium on Parallelism in Algorithms and Architectures*, pages 253–264, 2019.

4 Ryan Berryhill, Wojciech Golab, and Mahesh Tripunitara. Robust shared objects for non-volatile main memory. In *19th International Conference on Principles of Distributed Systems (OPODIS)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

5 Kyeongmin Cho, Seungmin Jeon, and Jeehoon Kang. Practical detectability for persistent lock-free data structures. *arXiv preprint arXiv:2203.07621*, 2022.

6 Andreia Correia, Pascal Felber, and Pedro Ramalhete. Persistent memory and the rise of universal constructions. In *15th European Conference on Computer Systems*, pages 1–15, 2020.

7 Panagiota Fatourou, Nikolaos D Kallimanis, and Eleftherios Kosmas. The performance power of software combining in persistence. In *27th ACM Symposium on Principles and Practice of Parallel Programming*, pages 337–352, 2022.

8 Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E Blelloch, and Erez Petrank. NVTraverse: In NVRAM data structures, the destination is more important than the journey. In *41st ACM Conference on Programming Language Design and Implementation*, pages 377–392, 2020.

9 Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. *ACM SIGPLAN Notices*, 53(1):28–40, 2018.

10 Wojciech Golab. The recoverable consensus hierarchy. In *32nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 281–291, 2020.

11 Wojciech Golab and Danny Hendler. Recoverable mutual exclusion in sub-logarithmic time. In *ACM Symposium on Principles of Distributed Computing*, pages 211–220, 2017.

12 Wojciech Golab and Aditya Ramaraju. Recoverable mutual exclusion. In *2016 ACM Symposium on Principles of Distributed Computing*, pages 65–74, 2016.

13 Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

14 Morteza Hoseinzadeh and Steven Swanson. Corundum: Statically-enforced persistent memory safety. In *26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 429–442, 2021.

15 Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via justdo logging. *ACM SIGARCH Computer Architecture News*, 44(2):427–442, 2016.

16 Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *30th International Symposium on Distributed Computing*, pages 313–327. Springer, 2016.

17 Prasad Jayanti, Siddhartha Jayanti, and Anup Joshi. A recoverable mutex algorithm with sub-logarithmic RMR on both cc and dsm. In *ACM Symposium on Principles of Distributed Computing*, pages 177–186, 2019.

18 Nan Li and Wojciech Golab. Detectable sequential specifications for recoverable shared objects. In *35th International Symposium on Distributed Computing (DISC)*, 2021.

19 John M Mellor-Crummey and Michael L Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

20 Liad Nahum, Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. Recoverable and detectable fetch&add. In *25th International Conference on Principles of Distributed Systems (OPODIS 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

21 Matan Rusanovsky, Hagit Attiya, Ohad Ben-Baruch, Tom Gerby, Danny Hendler, and Pedro Ramalhete. Flat-combining-based persistent data structures for non-volatile memory. In *23rd International Symposium on Stabilization, Safety, and Security of Distributed Systems, SSS*, pages 505–509. Springer, 2021.

Brief Announcement: Line Formation in Silent Programmable Matter

Alfredo Navarra
Department of Mathematics and Computer Science, University of Perugia, Perugia, Italy

Francesco Piselli[1]
Department of Mathematics and Computer Science, University of Perugia, Perugia, Italy

Abstract

Programmable Matter (PM) has been widely investigated in recent years. One reference model is certainly Amoebot, with its recent canonical version (DISC 2021). Along this line, with the aim of simplification and to address concurrency, the SILBOT model has been introduced (AAMAS 2020). Within SILBOT, we consider the *Line formation* primitive in which particles are required to end up in a configuration where they are all aligned and connected. We propose a simple and elegant distributed algorithm, optimal in terms of number of movements.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Theory of computation → Concurrency; Theory of computation → Self-organization

Keywords and phrases Programmable Matter, Line formation, Asynchrony, Stigmergy

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.45

Related Version An extended version appears in the Proceedings of the 25th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS), 2023.
Full Version: `https://arxiv.org/abs/2307.16731`

1 Introduction

In the recent years, main attention has been devoted to the so-called *Programmable Matter* (PM). This usually refers to a set of weak and self-organizing computational entities, called *particles*, with the ability to change its physical properties (e.g., shape or color) in a programmable way. Various models have been proposed so far. One that deserves main attention is certainly Amoebot, introduced in [7]. By then, various papers have considered that model, possibly varying some parameters. Moreover, a recent proposal to try to homogenize the referred literature has appeared in [6]. The main intent was to enhance the model with concurrency.

One of the weakest models for PM, that includes concurrency and eliminates direct communication among particles as well as local and shared memory, is SILBOT [4]. The purpose was to investigate the minimum settings for PM under which basic global tasks can be performed in a distributed manner. Toward this direction, we aim at studying in SILBOT the *Line formation* problem, where particles are required to reach a configuration where they are all aligned (i.e., lie on a same axis) and connected.

The relevance of the Line formation problem is provided by the interest shown in the last decades within various contexts of distributed computing. In graph theory, the problem has been considered in [10] where the requirement was to design a distributed algorithm that, given an arbitrary connected graph G of nodes with unique labels, converts G into a sorted list of nodes. In swarm robotics, the problem has been faced from a practical point of view, see, e.g. [11]. The relevance of line or V-shape formations has been addressed in various

[1] Corresponding author.

practical scenarios, as in [1, 19]. Most of the work on robots considers direct communications, memory, and some computational power. For application underwater or in the outerspace, instead, direct communications are rather unfeasible and this motivates the investigation on removing such a capability, see, e.g. [12, 17]. Concerning more theoretical models, the aim has been usually to study the minimal settings under which it is possible to realize basic primitives like Line formation. In [2, 16], for instance, Line formation has been investigated for (semi-)synchronized robots (punctiform or not, i.e., entities occupying some space) moving within the Euclidean plane, admitting limited visibility, and sharing the knowledge of one axis of direction. For synchronous robots moving in 3D space, in [18], the plane formation has been considered, which might be considered as the problem corresponding to Line formation for robots moving in 2D. In [13], robots operate within a triangular grid and Line formation is required as a preliminary step for accomplishing the Coating of an object. Within Amoebot, Line formation has been approached in [8], subject to the resolution of the leader election.

2 Definitions and notation

In this section, we review the SILBOT model for PM introduced in [4], and then we formalize the Line formation problem along with other useful definitions.

In SILBOT, particles operate on an infinite triangular grid embedded in the plane. Each node can contain at most one particle. Each particle is an automaton with two states, CONTRACTED or EXPANDED (they do not have any other form of persistent memory). In the former state, a particle occupies a single node of the grid while in the latter, the particle occupies one single node and one of the adjacent edges, see, e.g. Figure 1. Hence, a particle always occupies one node, at any time. Each particle can sense its surrounding up to a distance of 2 hops, i.e., if a particle occupies a node v, then it can see the neighbors of v, denoted by $N(v)$, and the neighbors of the neighbors of v. Hence, within its visibility range, a particle can detect empty nodes, CONTRACTED, and EXPANDED particles.

Any positioning of CONTRACTED or EXPANDED particles that includes all n particles composing the system is referred to as a *configuration*. Particles alternate between active and inactive periods decided by an adversarial schedule, independently for each particle.

In order to move, a particle alternates between EXPANDED and CONTRACTED states. In particular, a CONTRACTED particle occupying node v can move to a neighboring node u by expanding along edge (v, u), and then re-contracting on u. Note that, if node u is already occupied by another particle then the EXPANDED one will reach u only if u becomes empty, eventually, in a successive activation. There might be arbitrary delays between the actions of these two particles. When the particle at node u has moved to another node, the edge between v and u is still occupied by the originally EXPANDED particle. In this case, we say that node u is *semi-occupied.*

A particle commits itself into moving to node u by expanding in that direction, and at the next activation of the same particle, it is constrained to move to node u, if u is empty. A particle cannot revoke its expansion once committed.

The SILBOT model introduces a fine grained notion of asynchrony with possible delays between observations and movements performed by the particles. This reminds the so-called ASYNC schedule designed for theoretical models dealing with mobile and oblivious robots (see, e.g. [3, 5, 9]). All operations performed by the particles are non-atomic: there can be delays between the actions of sensing the surroundings, computing the next decision (e.g., expansion or contraction), executing the decision.

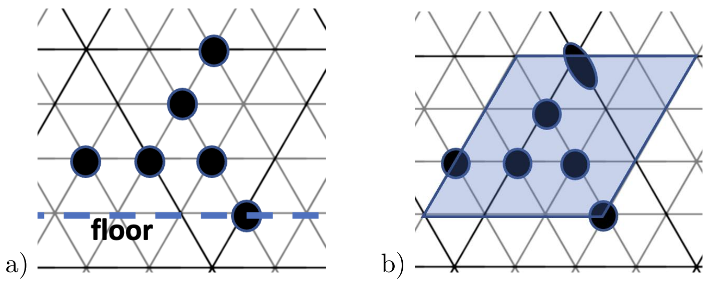


Figure 1 (*a*) A possible initial configuration with emphasized the *floor* (dashed line); (*b*) a possible evolution of the configuration shown in (a) with an expanded particle. The shaded parallelogram is the minimum bounding box containing all the particles.

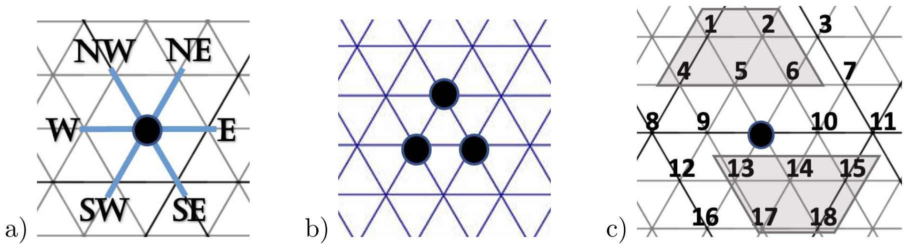


Figure 2 (*a*) A representation of the orientation of a particle; (*b*) An initial configuration where Line formation is unsolvable within SILBOT; (*c*) Enumerated visible neighborhood of a particle; the two trapezoids emphasize two relevant areas for the definition of our algorithm for Line formation.

The well-established fairness assumption is included, where each particle must be activated within finite time, infinitely often, in any execution of the particle system, see, e.g., [9].

Particles are required to take deterministic decisions. Each particle may be activated at any time independently from the others. Once activated, a particle looks at its surrounding (i.e., at 2 hops distance) and, on the basis of such an observation, decides (deterministically) its next *action*.

If two CONTRACTED particles decide to expand on the same edge simultaneously, exactly one of them (arbitrarily chosen by the adversary) succeeds.

If two particles are EXPANDED along two distinct edges incident to a same node w, toward w, and both particles are activated simultaneously, exactly one of the particles (again, chosen arbitrarily by the adversary) contracts to node w, whereas the other particle does not change its EXPANDED state according to the commitment constraint described above.

A relevant property that is usually required in such systems concerns connectivity. A configuration is said to be *connected* if the set of nodes occupied by particles induces a connected subgraph of the grid.

▶ **Definition 1.** *A configuration is said to be* initial*, if all the particles are* CONTRACTED *and connected.*

▶ **Definition 2** (Line formation)**.** *Given an initial configuration, the* Line formation *problem asks for an algorithm that leads to a configuration where all the particles are* CONTRACTED*, connected and aligned.*

▶ **Definition 3.** *Given a configuration C, the corresponding* bounding box *of C is the smallest parallelogram with sides parallel to the West–East and SouthWest–NorthEast directions, enclosing all the particles.*

Table 1 Literature on SILBOT.

Problem	*Schedule*	*View*	*Orientation*	*Reference*
Leader Election	ASYNC	2 hops	no	[4]
Scattering	ED-ASYNC	1 hop	no	[14]
Coating	ASYNC	2 hops	chirality	[15]
Line formation	ASYNC	2 hops	yes	**this paper**

See Figure 1.b for a visualization of the bounding box of a configuration. Note that, in general, since we are dealing with triangular grids, there might be three different bounding boxes according to the choice of two directions out of the three available. As it will be clarified later, for our purposes we just need to define one by choosing the West–East and SouthWest–NorthEast directions. In fact, as we are going to see in the next section, in order to solve Line formation in SILBOT, we need to add some capabilities to the particles. In particular, we add a common orientation to the particles. As shown in Figure 2.a, all particles commonly distinguish among the six directions of the neighborhood that by convention are referred to as the cardinal points $\mathbb{NW}$, $\mathbb{NE}$, $\mathbb{W}$, $\mathbb{E}$, $\mathbb{SW}$, and $\mathbb{SE}$.

Furthermore, in order to describe our algorithm, we need two further definitions that identify where the particles will be aligned.

▶ **Definition 4.** *Given a configuration C, the line of the triangular grid containing the southern side of the bounding box of C is called the* floor.

▶ **Definition 5.** *A configuration is said to be* final *if all the particles are* CONTRACTED*, connected and lie on floor.*

By the above definition, a final configuration is also initial. Moreover, if a configuration is final, then Line formation has been solved. Actually, it might be the case that a configuration satisfies the conditions of Def. 2 but still it is not final with respect to Def. 5. This is just due to the design of our algorithm that always leads to solve Line formation on floor.

3 Impossibility results

As shown in the previous section, the SILBOT model is very constrained in terms of particles capabilities. Since its first appearance [4], where the Leader Election problem has been solved, the authors pointed out the need of new assumptions in order to allow the resolution of other basic primitives. In fact, due to the very constrained capabilities of the particles, it was not possible to exploit the election of a leader to solve subsequent tasks. The parameters that can be manipulated have concerned the type of schedule, the hop distance from which particles acquire information, and the orientation of the particles. Table 1 summarizes the primitives so far approached within SILBOT and the corresponding assumptions. Leader Election was the first problem solved when introducing SILBOT [4]. Successively, the Scattering problem has been investigated [14]. It asks for moving the particles in order to reach a configuration where no two particles are neighboring to each other. Scattering has been solved by reducing the visibility range to just 1 hop distance but relaxing on the schedule which is not ASYNC. In fact, the ED-ASYNC schedule has been considered. It stands for Event-Driven Asynchrony, i.e., a particle activates as soon as it admits a neighboring particle, even though all subsequent actions may take different but finite time as in ASYNC. For Coating [15], where particles are required to surround an object that occupies some connected nodes of the grid, the original setting has been considered apart for admitting chirality, i.e., a common handedness among particles.

In this paper, we consider the Line formation problem, where particles are required to reach a configuration where they are all aligned and connected. About the assumptions, we add a common orientation to the particles to the basic SILBOT model. The motivation for endowing the particles with such a capability comes by the following result:

▶ **Theorem 6.** *Line formation is unsolvable within* SILBOT*, even though particles share a common chirality.*

By the assumed orientation, a particle can enumerate its neighborhood, up to distance of 2 hops, as shown in Figure 2.c. This will be useful for the definition of our algorithm.

4 Algorithm *WRain*

The rationale behind the name *WRain* of the proposed algorithm comes by the type of movements allowed. In fact, the evolution of the system on the basis of the algorithm mimics the behavior of particles that fall down like drops of rain subject to a westerly wind. The Line formation is then reached on the lower part of the initial configuration where there is at least a particle – what we have called *floor*.

In order to define Algorithm *WRain*, we need to define some functions, expressing properties related to a node of the grid. We make use of the enumeration shown in Fig. 2.c, and in particular to the neighbors enclosed by the two trapezoids.

▶ **Definition 7.** *Given a node v, the next Boolean functions are defined:*

- $\mathrm{Upper}(v)$ *is* true *if at least one of the visible neighboring nodes from v at positions* $\{1, 2, 4, 5, 6\}$ *is occupied by a particle;*
- $\mathrm{Lower}(v)$ *is* true *if at least one of the visible neighboring nodes from v at positions* $\{13, 14, 15, 17, 18\}$ *is occupied by a particle;*
- $\mathrm{Pointed}(v)$ *is* true *if there exists a particle p occupying a node $u \in N(v)$ such that p is* EXPANDED *along edge (u, v);*
- $\mathrm{Near}(v)$ *is* true *if there exists an empty node $u \in N(v)$ such that* $\mathrm{Pointed}(u)$ *is true.*

For the sake of conciseness, sometimes we make use of the above functions by providing a particle p as input in place of the corresponding node v occupied by p.

We are now ready to formalize our Algorithm *WRain*.

Algorithm 1 *WRain.*

```
Require: Node $v$ occupied by a CONTRACTED particle $p$.
Ensure: Line formation.
1: if $\neg Near(v)$ then
2:     if $Pointed(v)$ then
3:         $p$ expands toward $\mathbb{E}$
4:     else
5:         if $\neg Upper(v) \wedge Lower(v)$ then
6:             $p$ expands toward $\mathbb{SE}$
```

It is worth noting that Algorithm *WRain* allows only two types of expansion, toward $\mathbb{E}$ or $\mathbb{SE}$. Moreover, the movement toward $\mathbb{E}$ can happen only when the node v occupied by a particle is intended to be reached by another particle, i.e., *Pointed*(v) holds. Another remarkable property is that the algorithm only deals with expansion actions. This is due to the constraint of the SILBOT model that does not permit to intervene on EXPANDED particles, committed to terminate their movement.

5 Correctness and Optimality

In this section, we sketch the proof of correctness of Algorithm *WRain* as well as its optimality in terms of number of moves performed by the particles.

The correctness of Algorithm *WRain* is based on four claims:

Claim 1 - Configuration Uniqueness. Each configuration generated during the execution of the algorithm is unique, i.e., non-repeatable, after movements, on the same nodes nor on different nodes;

Claim 2 - Limited Dimension. The extension of any (generated) configuration is confined within a finite bounding box of sides $O(n)$;

Claim 3 - Evolution guarantee. If the (generated) configuration is connected and not final there always exists at least a particle that can expand or contract;

Claim 4 - Connectivity. If two particles initially neighboring to each other get disconnected, they recover their connection sooner or later (not necessarily becoming neighbors).

The four claims guarantee that a final configuration is achieved, eventually, in finite time, i.e., Line formation is solved. In fact, if from any non-final configuration reached during an execution of *WRain* there is always at least one particle that moves (Claim 3), the subsequent configuration must be different from any already reached configuration (Claim 1). However, since the area where the particles move is limited (Claim 2), then a final configuration must be reached as the number of achievable configurations is finite. Actually, if we imagine a configuration made of disconnected and CONTRACTED particles, all lying on *floor*, then the configuration is not final according to Def. 5 but none of the particles would move. We can prove that such type of configurations cannot occur, and in particular if two particles initially neighboring to each other get disconnected, then they recover their connection, eventually (Claim 4). Since the initial configuration is connected, then we are ensured that also the final configuration is connected as well.

We are now ready to state the correctness and the optimality of *WRain*.

▶ **Theorem 8.** *Given n CONTRACTED particles forming a connected configuration, Algorithm WRain solves Line formation within $\Theta(n^2)$ movements.*

6 Conclusion

We investigated on the Line formation problem within PM on the basis of the SILBOT model. With the aim of considering the smallest set of assumptions, we proved how chirality was not enough for particles to accomplish Line formation. We then endowed particles with a common sense of direction and we proposed *WRain*, an optimal algorithm – in terms of number of movements, for solving Line formation. Actually, it remains open whether by assuming just one common direction is enough for solving the problem. Furthermore, although in the original paper about SILBOT [4], it has been pointed out that 1 hop visibility is not enough for solving the Leader Election, it is worth investigating what happens for Line formation.

Other interesting research directions concern the resolution of other basic primitives, the formation of different shapes or the more general pattern formation problem.

References

1 He Cai, Shuping Guo, and Huanli Gao. A dynamic leader–follower approach for line marching of swarm robots. *Unmanned Systems*, 11(01):67–82, 2023.

2 Jannik Castenow, Thorsten Götte, Till Knollmann, and Friedhelm Meyer auf der Heide. The max-line-formation problem - and new insights for gathering and chain-formation. In *Proc. 23rd Int.'l Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 13046 of *LNCS*, pages 289–304. Springer, 2021.

3 Serafino Cicerone, Gabriele Di Stefano, and Alfredo Navarra. A structured methodology for designing distributed algorithms for mobile entities. *Information Sciences*, 574:111–132, 2021. `doi:10.1016/j.ins.2021.05.043`.

4 Gianlorenzo D'Angelo, Mattia D'Emidio, Shantanu Das, Alfredo Navarra, and Giuseppe Prencipe. Asynchronous silent programmable matter achieves leader election and compaction. *IEEE Access*, 8:207619–207634, 2020.

5 Gianlorenzo D'Angelo, Gabriele Di Stefano, Alfredo Navarra, Nicolas Nisse, and Karol Suchan. Computing on rings by oblivious robots: A unified approach for different tasks. *Algorithmica*, 72(4):1055–1096, 2015.

6 Joshua J. Daymude, Andréa W. Richa, and Christian Scheideler. The canonical amoebot model: Algorithms and concurrency control. In *Proc. 35th Int.'l Symp. on Distributed Computing (DISC)*, volume 209 of *LIPIcs*, pages 20:1–20:19, 2021.

7 Zahra Derakhshandeh, Shlomi Dolev, Robert Gmyr, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. Brief announcement: amoebot - a new model for programmable matter. In *Proc. 26th ACM Symp. on Parallelism in Algorithms and Architectures, (SPAA)*, pages 220–222. ACM, 2014.

8 Zahra Derakhshandeh, Robert Gmyr, Thim Strothmann, Rida A. Bazzi, Andréa W. Richa, and Christian Scheideler. Leader election and shape formation with self-organizing programmable matter. In Andrew Phillips and Peng Yin, editors, *Proc. 21st Int.'l Conf. on Computing and Molecular Programming (DNA)*, volume 9211 of *LNCS*, pages 117–132. Springer, 2015.

9 Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro, editors. *Distributed Computing by Mobile Entities, Current Research in Moving and Computing*, volume 11340 of *Lecture Notes in Computer Science*. Springer, 2019. `doi:10.1007/978-3-030-11072-7`.

10 Dominik Gall, Riko Jacob, Andréa W. Richa, Christian Scheideler, Stefan Schmid, and Hanjo Täubig. A note on the parallel runtime of self-stabilizing graph linearization. *Theory Comput. Syst.*, 55(1):110–135, 2014.

11 Donghwa Jeong and Kiju Lee. Dispersion and line formation in artificial swarm intelligence. In *Proc. 20th Int.'l Conf. on Collective Intelligence)*, 2014.

12 Zhiying Jiang, Xin Wang, and Jian Yang. Distributed line formation control in swarm robots. In *IEEE Int.'l Conf. on Information and Automation (ICIA)*, pages 636–641. IEEE, 2018.

13 Yonghwan Kim, Yoshiaki Katayama, and Koichi Wada. Pairbot: A novel model for autonomous mobile robot systems consisting of paired robots, 2020. `arXiv:2009.14426`.

14 Alfredo Navarra, Giuseppe Prencipe, Samuele Bonini, and Mirco Tracolli. Scattering with programmable matter. In *Proc. 37th Int.'l Conf. on Advanced Information Networking and Applications (AINA)*, Lecture Notes in Networks and Systems. Springer, 2023.

15 Francesco Piselli. Silent programmable matter: Coating. Master's thesis, University of Perugia, Italy, 2022.

16 Arijit Sil and Sruti Gan Chaudhuri. Formation of straight line by swarm robots. In *Proc. 7th Int.'l Conf. on Advanced Computing, Networking, and Informatics (ICACNI)*, pages 99–111. Springer, 2020.

17 Thomas Sousselier, Johann Dréo, and Marc Sevaux. Line formation algorithm in a swarm of reactive robots constrained by underwater environment. *Expert Syst. Appl.*, 42(12):5117–5127, 2015.

18 Yukiko Yamauchi, Taichi Uehara, Shuji Kijima, and Masafumi Yamashita. Plane formation by synchronous mobile robots in the three-dimensional euclidean space. *J. ACM*, 64(3):16:1–16:43, 2017.

19 Jian Yang, Xin Wang, and Peter Bauer. Line and v-shape formation based distributed processing for robotic swarms. *Sensors*, 18(8):2543, 2018.

Brief Announcement: The Space Complexity of Set Agreement Using Swap

Sean Ovens
University of Toronto, Canada

Abstract

We prove that any randomized wait-free n-process k-set agreement algorithm using only swap objects requires at least $\lceil \frac{n}{k} \rceil - 1$ objects. We also sketch a proof that any randomized wait-free consensus algorithm using only readable swap objects with domain size b requires at least $\frac{n-2}{3b+1}$ objects.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases space complexity, consensus, set agreement, lower bound, shared memory

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.46

Related Version *Full Version*: `https://arxiv.org/abs/2305.06507` [9]

Funding Support is gratefully acknowledged from the Natural Sciences and Engineering Research Council of Canada under grant RGPIN-2020-04178 and the Ontario Graduate Scholarships program.

1 Introduction

Consensus is one of the most well-studied problems in distributed computing. In the consensus problem, n processes begin with inputs and collectively agree on a single output. A consensus algorithm satisfies the following properties: agreement (no two outputs may differ) and validity (every output must be an input). There are known randomized wait-free [1, 5] and obstruction-free [10] consensus algorithms using n registers.

In 1993, Ellen, Herlihy, and Shavit [7] proved that $\Omega(\sqrt{n})$ registers are required to solve nondeterministic solo-terminating consensus. Lower bounds for nondeterministic solo-terminating algorithms also apply to randomized wait-free and obstruction-free algorithms. In 2016, Zhu [10] proved that $n-1$ registers are required to solve obstruction-free consensus. Finally, in 2018, Ellen, Gelashvili, and Zhu [6] used a completely new technique to prove that n registers are required to solve obstruction-free consensus. They also showed that space complexity lower bounds for obstruction-free algorithms using readable objects apply to nondeterministic solo-terminating algorithms.

The $\Omega(\sqrt{n})$ space complexity lower bound actually applies to consensus algorithms that use only historyless objects. A *historyless* object can only support two kinds of operations: *trivial* operations, which cannot change the value of the object, and *historyless* operations, which set the object to a fixed value. Registers are historyless objects because *Read* is trivial and *Write* is historyless. *Swap objects* are historyless objects that support *Swap*(v), which returns the current value of the object and then sets its value to v. Any historyless object can be simulated by one *readable swap object*, which supports *Read* and *Swap*(v).

The k-set agreement problem, first defined by Chaudhuri [4], is a generalization of consensus in which n processes begin with inputs and collectively agree on at most k distinct outputs. Obstruction-free k-set agreement is solvable using $n-k+1$ registers [2].

Ellen, Gelashvili, and Zhu [6] proved that any obstruction-free n-process k-set agreement algorithm using only registers requires at least $\lceil \frac{n}{k} \rceil$ registers. Before our results, there were no known non-constant lower bounds on the space complexity of solving k-set agreement using swap objects when $k > 1$.

Last year, we proved that any obstruction-free consensus algorithm using only readable swap objects with domain size 2 requires at least $n-2$ objects [8]. We also proved that any obstruction-free consensus algorithm using only swap objects requires at least $n-1$ objects. We have since refined the techniques from that paper to prove that at least $\frac{n-2}{3b+1}$ readable swap objects with domain size b are required to solve obstruction-free consensus, and that at least $\lceil \frac{n}{k} \rceil - 1$ swap objects are required to solve obstruction-free k-set agreement. In the full version of this paper [9], we also give an obstruction-free k-set agreement algorithm using $n-k$ swap objects, exactly matching our lower bound for $k=1$.

In Section 2, we present some definitions needed to prove our lower bounds. In Section 3, we prove our lower bound for set agreement using swap objects. In Section 4, we sketch the proof of our lower bound for consensus using readable swap objects with bounded domains.

2 Definitions

Two configurations C and C' are *indistinguishable* to a process p_i, denoted $C \overset{p_i}{\sim} C'$, if and only if p_i has the same state in C and C'. Two executions α and α' from C and C', respectively, are *indistinguishable* to a process p_i if $C \overset{p_i}{\sim} C'$ and p_i takes the same sequence of steps (and obtains the same responses) in α and α'. We use $\mathit{value}(B, C)$ to denote the value of the object B in configuration C.

In the *m-valued k-set agreement* problem, all process inputs are in $\{0, \dots, m-1\}$. The 2-valued 1-set agreement problem is also called *binary consensus*. A nonempty set of processes $\mathcal{P}$ is *v-univalent* in a configuration of a binary consensus algorithm C if, for every $\mathcal{P}$-only execution from C in which some process $p \in \mathcal{P}$ decides, v is the value decided by p. If $\mathcal{P}$ is neither 0-univalent nor 1-univalent in C, then $\mathcal{P}$ is *bivalent* in C.

An algorithm is *nondeterministic solo-terminating* if, for every configuration C of the algorithm and every process p, there is a solo-terminating execution by p from C. An algorithm is *obstruction-free* if it is deterministic and nondeterministic solo-terminating. An algorithm is *randomized wait-free* if, for every scheduler, the expected number of steps in any execution produced by that scheduler is finite.

3 Lower Bound for Set Agreement Using Swap Objects

Our lower bound relies on the following technical lemma.

▶ **Lemma 1.** *Consider an initial configuration C in which a set of processes $\mathcal{Q}$ have the same input v. Let α be an execution from C that does not involve $\mathcal{Q}$ such that k distinct values different from v are decided in $C\alpha$. Then the algorithm uses at least $|\mathcal{Q}|$ swap objects.*

Proof. Let $\mathcal{Q} = \{q_1, \dots, q_{|\mathcal{Q}|}\}$. Define $\mathcal{Q}_i = \{q_1, \dots, q_i\}$, for $1 \leq i \leq |\mathcal{Q}|$, and define $\mathcal{Q}_0 = \emptyset$. Let D be an initial configuration in which all processes have input v. For $0 \leq i \leq |\mathcal{Q}|$, we show that there is a set of i swap objects $\mathcal{A}_i$ and a pair of $\mathcal{Q}_i$-only executions γ_i and δ_i from $C\alpha$ and D, respectively, such that $\mathit{value}(B, C\alpha\gamma_i) = \mathit{value}(B, D\delta_i)$, for all $B \in \mathcal{A}_i$. For $i = |\mathcal{Q}|$, this claim proves the lemma. We use induction on i. When $i = 0$, γ_i and δ_i are empty executions, $\mathcal{A}_i = \emptyset$, and the claim is trivially satisfied.

Now let $0 \leq i < |\mathcal{Q}|$ and suppose there exists γ_i, δ_i, and $\mathcal{A}_i$ such that $\mathit{value}(B, C\alpha\gamma_i) = \mathit{value}(B, D\delta_i)$, for all $B \in \mathcal{A}_i$. Notice that $C\alpha\gamma_i \overset{q_{i+1}}{\sim} D\delta_i$, since q_{i+1} has input v in both configurations and takes no steps in α, γ_i, or δ_i. Consider a q_{i+1}-only solo-terminating execution σ from $D\delta_i$. By validity, q_{i+1} decides v in σ. Let τ be the longest prefix of σ such that q_{i+1} only accesses objects in $\mathcal{A}_i$ during τ. Since $\mathit{value}(B, C\alpha\gamma_i) = \mathit{value}(B, D\delta_i)$, for all

$B \in \mathcal{A}_i$, there is a q_{i+1}-only execution τ' from $C\alpha\gamma_i$ such that $\tau' \overset{q_{i+1}}{\sim} \tau$. If $\tau = \sigma$, then q_{i+1} decides v in τ and τ'. This is impossible, since $k+1$ different values are decided in $C\alpha\gamma_i\tau'$. Thus, τ is a proper prefix of σ. Then in $C\alpha\gamma_i\tau'$ and $D\delta_i\tau$, q_{i+1} is poised to apply a *Swap* operation s to an object $B^\star \notin \mathcal{A}_i$.

Since q_{i+1} applies the same sequence of operations in τ' and τ and $\mathit{value}(B, C\alpha\gamma_i) = \mathit{value}(B, D\delta_i)$ for all $B \in \mathcal{A}_i$, it follows that $\mathit{value}(B, C\alpha\gamma_i\tau') = \mathit{value}(B, D\delta_i\tau)$ for all $B \in \mathcal{A}_i$. By definition of *Swap*, $\mathit{value}(B^\star, C\alpha\gamma_i\tau's) = \mathit{value}(B^\star, D\delta_i\tau s)$. Taking $\gamma_{i+1} = \gamma_i\tau's$, $\delta_{i+1} = \delta_i\tau s$, and $\mathcal{A}_{i+1} = \mathcal{A}_i \cup \{B^\star\}$ completes the inductive step. ◀

We can now apply Lemma 1 to obtain the desired lower bound.

▶ **Theorem 2.** *For all $n > k \geq 1$, every nondeterministic solo-terminating, n-process $(k+1)$-valued k-set agreement algorithm from swap objects uses at least $\lceil \frac{n}{k} \rceil - 1$ objects.*

Proof. Consider such an algorithm for the set of processes $\mathcal{P} = \{p_0, \ldots, p_{n-1}\}$. We will use induction on k. When $k = 1$, the algorithm solves binary consensus. Consider an initial configuration C of the algorithm in which process p_0 has input 0 and all other processes have input 1. Note that p_0 decides 0 in its solo-terminating execution α from C. Therefore, by Lemma 1, the algorithm uses at least $n-1$ swap objects.

Now let $1 < k < n$ and suppose the theorem holds for $k-1$. Let $\mathcal{R}$ be some set of $\lceil \frac{n(k-1)}{k} \rceil$ processes in $\mathcal{P}$. Let $\mathcal{I}$ be the set of all initial configurations in which $\mathcal{R}$'s inputs are in $\{0, \ldots, k-1\}$. If, for every initial configuration $C \in \mathcal{I}$ and every $\mathcal{R}$-only execution α from C, at most $k-1$ different values are decided in α, then the algorithm solves nondeterministic solo-terminating k-valued $(k-1)$-set agreement among the processes in $\mathcal{R}$. Hence, by the inductive hypothesis, the algorithm uses at least $\lceil \frac{|\mathcal{R}|}{k-1} \rceil - 1 = \lceil \frac{n}{k} \rceil - 1$ swap objects.

Otherwise, there is a $C \in \mathcal{I}$ and an $\mathcal{R}$-only execution α from C in which $0, \ldots, k-1$ are decided. Notice $|\mathcal{P} - \mathcal{R}| = n - \left\lceil \frac{n(k-1)}{k} \right\rceil = \left\lfloor \frac{n}{k} \right\rfloor \geq \left\lceil \frac{n}{k} \right\rceil - 1$. By Lemma 1 (with $\mathcal{Q} = \mathcal{P} - \mathcal{R}$ and $v = k$), the algorithm uses at least $|\mathcal{P} - \mathcal{R}| \geq \lceil \frac{n}{k} \rceil - 1$ swap objects. ◀

4 Lower Bound for Consensus Using Readable Swap Objects

Let $\mathcal{Q} = \{q_0, q_1\}$ be a pair of processes and let $\mathcal{P} = \{p_0, \ldots, p_{n-3}\}$ be the rest of the processes. For all $0 \leq i \leq n-3$, define $\mathcal{P}_i = \{p_i, \ldots, p_{n-3}\}$. In particular, $\mathcal{P}_0 = \mathcal{P}$. Define $\mathcal{P}_{n-2} = \emptyset$. Let $\mathcal{A}$ be the set of all readable swap objects with domain size b used by the algorithm.

A set S of processes *covers* a set $\mathcal{B}$ of objects in a configuration C if, for every object $B \in \mathcal{B}$, there is a unique process in S that is poised to apply a *Swap* to B in C. A *block swap* by S is an execution that consists of a single step by each process in S.

▶ **Lemma 3** ([8]). *Let C be a configuration in which $\mathcal{Q}$ is bivalent and a set $S \subseteq \mathcal{P}$ of processes cover a set $\mathcal{B}$ of readable swap objects. Then there is a $\mathcal{Q}$-only execution γ from C such that $\mathcal{Q}$ is bivalent in $C\gamma\beta$, where β is a block swap by S.*

The following result uses Lemma 3 and appears in the full version of the paper [9].

▶ **Lemma 4.** *Let $p_i \in \mathcal{P}$, let C be a configuration in which $\mathcal{Q}$ is bivalent, let C' be a configuration such that $C \overset{p_i}{\sim} C'$, and let δ be p_i's solo-terminating execution from C'. Suppose δ consists of r steps and, for all $s \in \{0, \ldots, r\}$, let δ_s be the prefix of δ that consists of the first s steps by p_i. Then there is a $j \in \{0, \ldots, r-1\}$ such that,*

(a) *for all $j' \in \{0, \ldots, j\}$, there is a $(\mathcal{Q} \cup \mathcal{P}_i)$-only execution $\alpha_{j'}$ from C such that $\mathcal{Q}$ is bivalent in $C\alpha_{j'}$ and $\alpha_{j'} \overset{p_i}{\sim} \delta_{j'}$.*

Consider any $(\mathcal{Q} \cup \mathcal{P}_i)$-only execution α_j from C such that $\mathcal{Q}$ is bivalent in $C\alpha_j$ and $\alpha_j \overset{p_i}{\sim} \delta_j$. Let d be the operation that p_i is poised to apply to the object B in $C'\delta_j$. Then for every $(\mathcal{Q} \cup \mathcal{P}_{i+1})$-only execution λ' from $C\alpha_j$,

(b) *if value$(B, C\alpha_j\lambda')$ = value$(B, C'\delta_j)$, then $\mathcal{Q}$ is univalent in $C\alpha_j\lambda'd$, and*
(c) *if value$(B, C'\delta_j)$ = value$(B, C'\delta_j d)$ and in some configuration of λ' the value of B is value$(B, C'\delta_j)$, then $\mathcal{Q}$ is univalent in $C\alpha_j\lambda'$.*

We now sketch a proof of our main technical lemma. For all $0 \le i \le n-2$, it constructs a configuration C_i and two functions f_i and g_i that map objects to increasingly large sets of values that are forbidden in certain executions from C_i.

▶ **Lemma 5.** *For all $0 \le i \le n-2$, there is a configuration C_i reachable from C_0, a set of processes $S_i \subseteq \mathcal{P} - \mathcal{P}_i$, and a pair of functions f_i, g_i that map objects to subsets of $\{0, \ldots, b-1\}$ such that the following holds for every $(\mathcal{Q} \cup \mathcal{P}_i)$-only execution λ from C_i:*

(a) *$\mathcal{Q}$ is bivalent in C_i,*
(b) *S_i covers a set of $|S_i|$ objects in C_i,*
(c) *for every process $p \in S_i$, if p is poised to apply a* Swap(B, x) *operation in C_i, then $x \notin f_i(B) \cup g_i(B)$,*
(d) *$\sum_{B \in \mathcal{A}} (2 \cdot |f_i(B)| + |g_i(B)|) + |S_i| \ge i$,*
(e) *if the value of some object B is equal to some value in $f_i(B)$ in some configuration of λ, then $\mathcal{Q}$ is univalent in $C_i\lambda$, and*
(f) *if some process $p \in \mathcal{P}_i$ is poised to apply a* Swap(B, x) *operation in $C_i\lambda$ for some object B and some $x \in g_i(B)$, then $\mathcal{Q}$ is univalent in $C_i\lambda$.*

Proof sketch. We use induction on i. Let $S_0 = \emptyset$ and let $f_0(B) = g_0(B) = \emptyset$ for all $B \in \mathcal{A}$. All properties of the lemma are simple to verify for $i = 0$.

Now suppose that the lemma holds for some $0 \le i \le n-3$. Let δ be p_i's solo-terminating execution from $C_i\beta_i$, where β_i is a block swap by S_i. Suppose that δ consists of r steps by p_i, and let δ_s be the prefix of δ that consists of its first s steps. Let $0 \le j \le r-1$ be the value that satisfies the conditions of Lemma 4 (with $C = C_i$ and $C' = C_i\beta_i$).

In the full version of this paper [9], we prove that, for all $B \in \mathcal{A}$ and all forbidden values $x \in f_i(B) \cup g_i(B)$, process p_i does not apply any *Swap*(B, x) operations during δ_{j+1}.

Let d be the final step of δ_{j+1} by p_i. Let $B^\star$ be the object accessed by p_i in step d. Let $v^\star = \mathit{value}(B^\star, C_i\beta_i\delta_j)$. By Lemma 4(a), there is a $(\mathcal{Q} \cup \mathcal{P}_i)$-only execution α_j from C_i such that $\mathcal{Q}$ is bivalent in $C_i\alpha_j$ and $\alpha_j \overset{p_i}{\sim} \delta_j$. Define $C_{i+1} = C_i\alpha_j$, so property (a) holds for $i+1$.

Case 1. Step d does not change the value of $B^\star$ when it is applied in $C_i\beta_i\delta_j$. Define $f_{i+1}(B) = f_i(B)$ for all $B \in \mathcal{A} - \{B^\star\}$, $g_{i+1}(B) = g_i(B)$ for all $B \in \mathcal{A}$, and $f_{i+1}(B^\star) = f_i(B^\star) \cup \{v^\star\}$.

If there is a process $p \in S_i$ that is poised to apply a *Swap*$(B^\star, v^\star)$ operation in C_i, then define $S_{i+1} = S_i - \{p\}$. Otherwise, define $S_{i+1} = S_i$. Since no process in S_i takes steps during α_j and $S_{i+1} \subseteq S_i$, property (b) holds for $i+1$. In addition, property (c) holds for i, so it holds for $i+1$ as well.

Suppose $v^\star \in f_i(B^\star)$. Then process p_i does not apply *Swap*$(B^\star, v^\star)$ during δ_{j+1}. Hence, $\mathit{value}(B^\star, C_i\beta_i) = v^\star$. Property (c) for i implies that no process applies *Swap*$(B^\star, v^\star)$ during β_i. Thus, $\mathit{value}(B^\star, C_i) = v^\star$. By property (e) for i (where λ is the empty execution), $\mathcal{Q}$ is univalent in C_i. This contradicts property (a) for i. Hence, $v^\star \notin f_i(B^\star)$. This implies that $|f_{i+1}(B^\star)| = |f_i(B^\star)| + 1$. Since $|S_{i+1}| \ge |S_i| - 1$, property (d) holds for $i+1$.

Let λ be a $(\mathcal{Q} \cup \mathcal{P}_{i+1})$-only execution from C_{i+1}. Then $\alpha_j\lambda$ is a $(\mathcal{Q} \cup \mathcal{P}_i)$-only execution from C_i. By property (e) for i, if the value of some object B is equal to a value in $f_i(B)$ in any configuration of $\alpha_j\lambda$, then $\mathcal{Q}$ is univalent in $C_i\alpha_j\lambda$. Lemma 4(c) (with $\lambda' = \lambda$) implies that, if the value of $B^\star$ is $v^\star$ in some configuration of λ, then $\mathcal{Q}$ is univalent in $C_i\alpha_j\lambda$. This gives us property (e) for $i+1$. Since $g_{i+1}(B) = g_i(B)$ for all $B \in \mathcal{A}$, property (f) for $i+1$ follows from property (f) for i.

Case 2. Step d changes the value of $B^\star$ when it is applied in $C_i\beta_i\delta_j$. Then d is a *Swap*$(B^\star, v')$ operation, for some $v' \in \{0, \ldots, b-1\} - \{v^\star\}$. Define $f_{i+1}(B) = f_i(B)$ for all $B \in \mathcal{A}$, $g_{i+1}(B) = g_i(B)$ for all $B \in \mathcal{A} - \{B^\star\}$, and $g_{i+1}(B^\star) = g_i(B^\star) \cup \{v^\star\}$.

If some process $p \in S_i$ is poised to access $B^\star$ in C_i, then define $S_{i+1} = (S_i - \{p\}) \cup \{p_i\}$. Otherwise, define $S_{i+1} = S_i \cup \{p_i\}$. In either case, we obtain property (b) for $i+1$.

Since p_i does not apply *Swap*(B, x) during δ_{j+1}, for any $B \in \mathcal{A}$ and any $x \in f_i(B) \cup g_i(B)$, it follows that $v' \notin f_i(B^\star) \cup g_i(B^\star)$. Furthermore, we know that $v' \neq v^\star$. Hence, $v' \notin f_{i+1}(B^\star) \cup g_{i+1}(B^\star)$. This along with property (c) for i gives us property (c) for $i+1$.

If $B^\star$ is not covered by S_i in C_i, then $S_{i+1} = S_i \cup \{p_i\}$, so $|S_{i+1}| = |S_i| + 1$. Furthermore, $|g_{i+1}(B^\star)| \geq |g_i(B^\star)|$. Property (d) for $i+1$ follows from this and property (d) for i.

Otherwise, $B^\star$ is covered by S_i in C_i. By property (c) for i, *value*$(B, C_i\beta_i) \notin f_i(B) \cup g_i(B)$ for all objects B covered by S_i in C_i. Furthermore, p_i does not apply *Swap*(B, x) during δ_{j+1}, for any $B \in \mathcal{A}$ and any $x \in f_i(B) \cup g_i(B)$. Hence, *value*$(B, C_i\beta_i\delta_j) \notin f_i(B) \cup g_i(B)$ for all objects B covered by S_i in C_i. Since $B^\star$ is covered by S_i in C_i, it follows that $v^\star \notin f_i(B^\star) \cup g_i(B^\star)$. Hence, $|g_{i+1}(B^\star)| = |g_i(B^\star)| + 1$. Furthermore, $|S_{i+1}| = |S_i|$. Combined with property (d) for i, this gives us property (d) for $i+1$.

Let λ be a $(\mathcal{Q} \cup \mathcal{P}_{i+1})$-only execution from C_{i+1}. Then $\alpha_j\lambda$ is a $(\mathcal{Q} \cup \mathcal{P}_i)$-only execution from C_i. Since $f_{i+1}(B) = f_i(B)$ for all $B \in \mathcal{A}$, property (e) for $i+1$ follows from property (e) for i. Suppose there is a $p \in \mathcal{P}_{i+1}$ poised to apply a *Swap*$(B^\star, v^\star)$ operation t in $C_{i+1}\lambda$. Recall that $\alpha_j \overset{p_i}{\sim} \delta_j$, so p_i is poised to apply d in $C_i\alpha_j = C_{i+1}$. Since p_i takes no steps in λ, it is poised to apply d in $C_{i+1}\lambda$. Since d is a *Swap*$(B^\star, v')$ operation, p_i covers $B^\star$ in $C_{i+1}\lambda$. If $\mathcal{Q}$ is bivalent in $C_{i+1}\lambda$, then, by Lemma 3 (with $C = C_{i+1}\lambda$ and $S = \{p_i\}$), there is a $\mathcal{Q}$-only execution γ from $C_{i+1}\lambda$ such that $\mathcal{Q}$ is bivalent in $C_{i+1}\lambda\gamma d$ and, hence, in $C_{i+1}\lambda\gamma t d$. However, Lemma 4(b) (with $\lambda' = \lambda\gamma t$) implies that $\mathcal{Q}$ is univalent in $C_{i+1}\lambda\gamma t d$. Hence, $\mathcal{Q}$ is univalent in $C_{i+1}\lambda$. Property (f) for $i+1$ follows from this and property (f) for i. ◀

Lemma 5(d) (with $i = n-2$) says that $\sum_{B \in \mathcal{A}} \big(2 \cdot |f_{n-2}(B)| + |g_{n-2}(B)|\big) + |S_{n-2}| \geq n-2$. By part (b), S_{n-2} covers a set of $|S_{n-2}|$ objects in C_{n-2}. Hence, $|S_{n-2}| \leq |\mathcal{A}|$. Moreover, $\sum_{B \in \mathcal{A}} \big(2 \cdot |f_{n-2}(B)| + |g_{n-2}(B)|\big) \leq 3 \cdot b \cdot |\mathcal{A}|$ since $f_{n-2}(B)$ and $g_{n-2}(B)$ are subsets of $\{0, \ldots b-1\}$. Thus, $3 \cdot b \cdot |\mathcal{A}| + |\mathcal{A}| \geq n-2$, which implies the lower bound.

▶ **Theorem 6.** *For all $n, b \geq 2$, any n-process, obstruction-free binary consensus algorithm from readable swap objects with domain size b uses at least $\frac{n-2}{3b+1}$ objects.*

5 Conclusion

Determining the exact space complexity of solving obstruction-free k-set agreement using swap objects when $k > 1$ remains an open problem. We conjecture that $n-k$ swap objects are required. Theorem 6 implies that any obstruction-free consensus algorithm from readable swap objects with constant-sized domain requires $\Omega(n)$ objects. This asymptotically matches Bowman's [3] algorithm, which uses $2n-1$ registers with domain size 2.

References

1 James Aspnes and Maurice Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–461, 1990. doi:10.1016/0196-6774(90)90021-6.

2 Zohir Bouzid, Michel Raynal, and Pierre Sutra. Anonymous obstruction-free (n, k)-set agreement with n-k+1 atomic read/write registers. *Distributed Comput.*, 31(2):99–117, 2018. doi:10.1007/s00446-017-0301-7.

3 Jack R. Bowman. Obstruction-free snapshot, obstruction-free consensus, and fetch-and-add modulo k. Master's thesis, Dartmouth College, Computer Science, 2011. URL: https://digitalcommons.dartmouth.edu/senior_theses/67.

4 S. Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132–158, 1993. doi:10.1006/inco.1993.1043.

5 B. Chor, A. Israeli, and M. Li. Wait-free consensus using asynchronous hardware. *SIAM J. Comput.*, 23:701–712, 1994.

6 Faith Ellen, Rati Gelashvili, and Leqi Zhu. Revisionist simulations: A new approach to proving space lower bounds. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, PODC '18, pages 61–70, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3212734.3212749.

7 Faith Fich, Maurice Herlihy, and Nir Shavit. On the space complexity of randomized synchronization. *J. ACM*, 45(5):843–862, September 1998. A preliminary version appeared in PODC '93. doi:10.1145/290179.290183.

8 Sean Ovens. The space complexity of consensus from swap. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, PODC'22, pages 176–186, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3519270.3538420.

9 Sean Ovens. The space complexity of consensus from swap. *CoRR*, abs/2305.06507, 2023. arXiv:2305.06507.

10 Leqi Zhu. A tight space bound for consensus. *SIAM J. Comput.*, 50(3), 2019. A preliminary version appeared in STOC '16. doi:10.1137/16M1096785.

Brief Announcement: Grassroots Distributed Systems: Concept, Examples, Implementation and Applications

Ehud Shapiro
Weizmann Institute of Science, Rehovot, Israel

Abstract

Informally, a distributed system is *grassroots* if it is permissionless and can have autonomous, independently-deployed instances – geographically and over time – that may interoperate voluntarily once interconnected. More formally, in a grassroots system the set of all correct behaviors of a set of agents P is strictly included in the set of the correct behaviors of P when they are embedded within a larger set of agents $P' \supset P$.

Grassroots systems are potentially important as they may allow communities to conduct their social, economic, civic, and political lives in the digital realm solely using their members' networked computing devices (e.g., smartphones), free of third-party control, surveillance, manipulation, coercion, or rent seeking (e.g., by global digital platforms such as Facebook or Bitcoin).

Client-server/cloud computing systems are not grassroots, and neither are systems designed to have a single global instance (Bitcoin/Ethereum with hardwired seed miners/bootnodes), and systems that rely on a single global data structure (IPFS, DHTs). An example grassroots system would be a serverless smartphone-based social network supporting multiple independently-budding communities that can merge when a member of one community becomes also a member of another.

Here, we formalize the notion of grassroots distributed systems; describe a grassroots dissemination protocol for the model of asynchrony and argue its safety, liveness, and being grassroots; extend the implementation to mobile (address-changing) devices that communicate via an unreliable network (e.g. smartphones using UDP); and discuss how grassroots dissemination can realize grassroots social networking and grassroots cryptocurrencies. The mathematical construction employs distributed multiagent transition systems to define the notions of grassroots protocols, to specify the grassroots dissemination protocols, and to prove their correctness. The protocols use the blocklace – a distributed, partially-ordered counterpart of the replicated, totally-ordered blockchain.

2012 ACM Subject Classification Computer systems organization → Peer-to-peer architectures; Networks → Network protocol design; Networks → Formal specifications; Software and its engineering → Distributed systems organizing principles

Keywords and phrases Grassroots Distributed Systems, Dissemination Protocol, Multiagent Transition Systems, Blocklace, Cordial Dissemination

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.47

Related Version *Full Version*: https://arxiv.org/abs/2301.04391

Acknowledgements I thank Andy Lewis-Pye, Nimrod Talmon and Oded Naor for their comments on an earlier version of the manuscript and Idit Keidar for pointing me to related work. Ehud Shapiro is the Incumbent of The Harry Weinrebe Professorial Chair of Computer Science and Biology at the Weizmann Institute.

1 Introduction

The digital realm is dominated by global digital platforms of two architectures: The first is autocratic (one person – all votes) cloud-based global digital platforms that have adopted surveillance-capitalism [20] as their business model, driving the platforms to monitor, induce, and manipulate their inhabitants for profit. Some regimes also require a "back-door" to the global platform to monitor, censor, control, and even punish the digital behavior of its citizens. The second is blockchain/cryptocurrencies-based systems, which are peer-to-peer and can be "permissionless", open to participation by everyone. These platforms are governed via proof-of-work or proof-of-stake protocols, which are intrinsically plutocratic (one coin – one vote). Despite the promise of openness and distribution of power, the leading cryptocurrency platforms are controlled by a handful of ultra-high-performance server-clusters [8]. See the full paper [13] for a discussion of other server-based systems [7, 3, 1, 6, 17, 4, 18].

Here, we are concerned with providing an alternative architecture for the digital realm, referred to as a *grassroots architecture*, to serve as a foundations for peer-to-peer, smartphone-based, serverless applications [18, 14]. Ultimately, a grassroots architecture may provide a protocol stack to support grassroots digital democracy [15].

In general, a system designed to have a single global instance is not grassroots. Client-server/cloud systems in which two instances cannot co-exist due to competition/conflict on shared resources (e.g., same web address), or cannot interoperate when interconnected, are not grassroots. Neither are peer-to-peer systems that require all-to-all dissemination, including mainstream cryptocurrencies and standard consensus protocols [5, 19, 9], since a community placed in a larger context cannot ignore members of the larger context. Neither are systems that use a global shared data-structure such as pub/sub systems [6], IPFS [3], and distributed hash tables [16], since a community placed in a larger context cannot ignore updates to the shared resource by others.

2 Grassroots Protocols

We assume a potentially-infinite set of agents Π (think of all the agents yet to be produced), but when referring to a subset of the agents $P \subseteq \Pi$ we assume P to be finite. Each agent is associated with a single and unique key-pair of its own choosing, and is identified by its public key $p \in \Pi$. We refer the reader to the full paper [13] for the necessary definitions and their explanations, and introduce the notion of a grassroots protocol without further ado:

▶ **Definition 1** (Grassroots). *A protocol $\mathcal{F}$ is* ***grassroots*** *if $\emptyset \subset P \subset P' \subseteq \Pi$ implies that $TS(P) \subset TS(P')/P$.*

Informally, a group of agents P with the a set of possible behaviors $TS(P)$ has possible behaviors $TS(P')/P$ when embedded within a larger group P'. A protocol is grassroots if: (i) The behaviors of the agents P on their own, $TS(P)$, are also possible behaviors of these agents when embedded within a larger group P', $TS(P')/P$. In other words, the agents in P may choose to ignore the agents in $P' \setminus P$. Hence the subset relation. (ii) This latter set of behaviors $TS(P')/P$ includes additional possible behaviors of P not in $TS(P)$. Thus, there are possible behaviors of P, when embedded within P', which are not possible when P are on their own. This is presumably due to interactions between members of P and members of $P' \setminus P$. Hence the subset relation is strict.

▶ **Observation 2.** *An all-to-all dissemination protocol cannot be grassroots.*

Intuitively, a group of agents P engaged in a hypothetical grassroots all-to-all dissemination protocol may ignore the additional agents in $P' \setminus P$, in contradiction to the dissemination protocol being all-to-all. The full paper provides a sufficient condition for a protocol to be grassroots, which are useful in proving such a claim:

▶ **Theorem 3** (Grassroots Protocol). *An asynchronous, interactive, and non-interfering protocol is grassroots.*

Informally, a protocol is *asynchronous* if a transition by an agent, once enabled, cannot be disabled by transitions taken by other agents; it is *interactive* if the addition of agents to a group results in additional possible behaviors of the group; and it is *non-interfering* if the possible behaviors of a group of agents are not hampered by the presence of additional stationary agents, namely agents that remain in their initial state.

3 The Social Graph

Paul Baran's original vision of the Internet [2] was of a network of unmanned nodes that would act as switches, routing information from one node to another to their final destinations. Grassroots dissemination is different from packet switching: A block in a blocklace has no "destination", only an author, and dissemination occurs via communication along the edges of the social graph, based on the following social principles.

Social Principles of Cordial Dissemination:
1. **Disclosure**: Tell your friends which blocks you know and which you need
2. **Cordiality**: Send to your friends blocks you know and think they need

In Cordial Dissemination, agents may *follow* other agents, and two agents are *friends* if they follow each other. The basic rule of Cordial Dissemination is that an agent p can receive from agent q a q'-block b if p and q are friends and either $q = q'$ or both p and q follow q'. Note that the friendship relation induces an undirected graph on the agents, referred to as the *social graph*. A *friendship path* is a path in the social graph. The key liveness claim of Cordial Dissemination is that p eventually knows any q-block if there is a friendship path of correct agents from p to q, all of which follow q. The social principles of Cordial Dissemination are realized via the blocklace, introduced next.

4 The Blocklace

The blocklace is a distributed, partially-ordered counterpart of the replicated, totally-ordered blockchain data-structure. It is formally introduced in the full paper [13], and has already been applied to the implementation of grassroots social networking [14], the Flash payment system [11], and the Cordial Miners family of consensus protocols [10]. Here is a concise introduction of its basic concepts.

We assume a given cryptographic hash function *hash*. A *block* created by agent $p \in \Pi$, also referred to as a *p-block*, is a triple $b = (h, x, H)$, with h being a hash pointer $\mathit{hash}((x, H))$ signed by p, also referred to as a *p-pointer*; x being the *payload* of b; and H a finite set of signed hash pointers. If $H = \emptyset$ then b is a *genesis block*; an hash pointer $h' \in H$ points to the block $(h', x', H)'$, if such a block exists; and if h' is a p-pointer it is also referred to as a *self-pointer*.

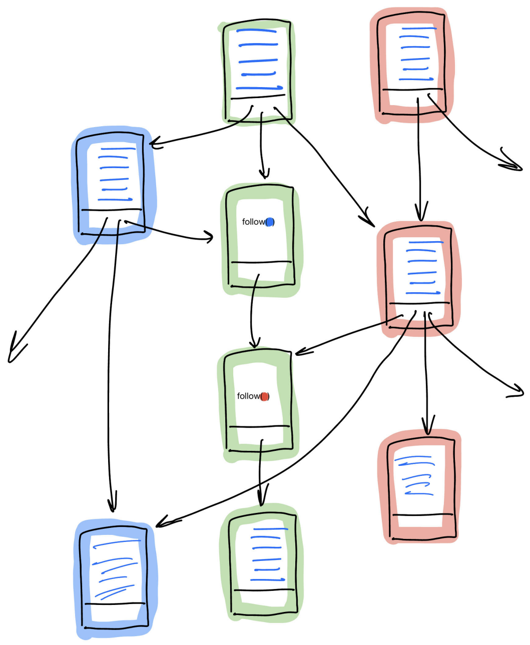


Figure 1 A self-closed blocklace: Blocks are color-coded by agent, thus pointers among blocks of the same color are self-pointers, others are non-self pointers, which may be dangling.

A *blocklace* is a set of blocks, which is *closed* if every pointer in every block in B points to a block in B. A pointer that does not point to a block in B is *dangling* in B, hence a closed blocklace has no dangling pointers. A blocklace is *self-closed* if it has no dangling self-pointers. Note that the notion of a cryptographic hash function implies that it is computationally-infeasible to create hash pointers that form a cycle. We are concerned only with computationally-feasible blocklaces, and any such blocklace B induces a DAG, with the blocks of B as vertices and with directed edges $b \to b'$ for every $b, b' \in B$ for which b includes a pointer to b'. A block b *observes* a block b' in B if there is a path $b = b_1 \to b_2 \ldots \to b_n = b'$ in B, $n \geq 1$. We note that the "observes" relation is a partial order on B. Two p-blocks that do not observe each other are referred to as an *equivocation* by p, and if their payloads include conflicting financial transactions by p, they are also called a *double-spend* by p. If B includes an equivocation by p we say that p is an *equivocator* in B.

5 Cordial Dissemination

Here we present the blocklace-based Cordial Dissemination Protocol $\mathcal{CD}$; claim it to be live (Prop. 4) and grassroots (Prop. 5); present pseudocode realizing $\mathcal{CD}$ for the model of Asynchrony (Alg. 1); and discuss an implementation for mobile agents communicating over an unreliable network, namely smartphones communicating via UDP.

We employ the blocklace as follows. The *local state* of each agent p is a blocklace, consisting of blocks produced by p and blocks by agents that p follows and that were received by p. A correct agent maintains a self-closed blocklace, buffering received out-of-order blocks. A *configuration* c consists of a set of local states, one for each agent. The local state of agent p in c is denoted by c_p. Given agents $p, q \in \Pi$, then p *follows* q in c if c_p includes a p-block with payload $(\textsc{follow}, q)$; p *needs* the q-block b in c if p follows q in c_p and $b \in c_q \setminus c_p$; and p and q are *friends* in c if p and q follow each other in c. The *social graph* $(P, E(c))$ induced by a configuration c has an edge $(p, q) \in E(c)$ if p and q are friends in c.

The social principles of Cordial Dissemination are realized by the blocklace as follows: Disclosure is realized by the Create transition, with any new p-block serving as a *multichannel ack/nack message*, informing whether p follows another agent q, and if so also of the latest q-block known to p, for every $q \in \Pi$. Cordiality is realized by the Cordially-Send-b-to-q transition. The liveness condition requires an agent to disclose every so often the blocks they know and to receive any block sent to it. The Follow transition allows p to offer friendship to any agent q; but there is no liveness requirement on p to do so. Agents p and q are friends if they follow each other, namely both have produced a FOLLOW block for the other. Agent p of course knows if it follows q; but p can know that q follows it only if it receives a (FOLLOW, p) block b. However, b may include q-pointers to blocks p does not know yet, hence p may have to "peek" into its received but not-yet-incorporated blocks and look for a (FOLLOW, p) q-block, in order to know whether it is friends with q.

The full paper describes the cordial dissemination protocol as a family of distributed multiagent transition systems. Here we capture the essence of the protocol informally.

The $\mathcal{CD}$ Cordial Dissemination Protocol
The local state of each agent p consists of a blocklace B, which initially includes a genesis p-block that has no payload and no pointers, as well as received blocks not yet incorporated in B.
The protocol proceeds by any agent p taking any of the following transitions:

1. **Create/Follow**: Create a new p-block b that points to the tips of B, as well as to the previous p-block, and add b to B. If the payload of b is (FOLLOW, q) then send b to q.
2. **Cordially-Send-b-to-q**: If B has a block b such that (i) p knows that q is a friend, (ii) p knows that q follows the creator of b, and (iii) p does not know that q knows b, then send b to q.
3. **Receive-b**: If a received q-block b is not in B then add b to B, provided p follows q and any q-block b points to is already in B.

The liveness condition of the protocol requires that every message sent is eventually received, and every agent every so often creates a new block and cordially sends blocks to their friends.

The safety assurance of the protocol is that the local blocklace of any correct agent p is self-closed and has no equivocations by p (but may include equivocation by faulty agents). The following liveness proposition holds for the model of asynchrony:

▶ **Proposition 4** ($\mathcal{CD}$ Liveness)**.** *Let r be a run of $\mathcal{CD}$, $p, q \in P$. If in some configuration $c \in r$, p and q are connected via a friendship path, all of its members follow q in c and are correct in r, then for every q-block b in r there is a configuration $c' \in r$ for which $b \in c'_p$.*

▶ **Proposition 5.** *The Cordial Dissemination protocol $\mathcal{CD}$ is grassroots.*

6 Pseudocode Implementation

Algorithm 1 presents pseudocode implementation of the Cordial Dissemination protocol $\mathcal{CD}$, for an single agent p for the model of Asynchrony. We assume that the agent p can specify the payload for a new block, including it being a friendship offer. As according to the model of asynchrony each message sent is eventually received, we assume the **reliably_send** construct to keep a record of sent messages so as not to send the same message to the same agent twice.

Algorithm 1 **Grassroots Cordial Dissemination for Asynchrony**
Code for agent p.

```
    Local variables:
1:  $B \leftarrow \{create\_block(\perp, \emptyset)\}$                    ▷ The local blocklace of agent $p$

2:  upon decision to create block with payload $x$ do
3:      $b \leftarrow create\_block(x, tips(B))$                           ▷ 1. Create
4:      $B \leftarrow B \cup \{b\}$
5:      if $x = (\text{FOLLOW}, q)$ then reliably_send $b$ to $q$         ▷ 1. Follow

6:  upon a new block in $B$ do                                          ▷ 2. Cordially-Send-$b$-to-$q$
7:      for all $b \in B, q \in P : friend(q) \wedge follows(q, b.creator) \wedge \neg agentObserves(q, b)$ do
8:          reliably_send $b$ to $q$

9:  upon receive $b$ s.t. $B \cup \{b\}$ is self-closed and $p$ follows $b.creator$ do   ▷ 3. Receive-$b$
10:     $B \leftarrow B \cup \{b\}$
```

Mobile Agents Communicating via UDP. A refinement of the Cordial Dissemination protocol for mobile (address-hopping) agents communicating over an unreliable network, namely smartphones communicating via UDP, is presented in the full paper, and a more concrete variant of it is preserted in the context of grassroots social networking [14]. Cordial dissemination over UDP exploits the ack/nak information of blocklace blocks to its fullest, by p retransmitting to every friend q every block b that p knows (not only p-blocks) and believes that q needs, until q acknowledges knowing b. In this protocol, every block includes also the IP address of its creator at the time of creation, and a new p-block is created whenever p changes its IP address. Retransmission is initiated by timeout, and assuming that timeouts are separated by seconds and mobile address changes are independent and are separated by hours, the probability of two friends hopping together without one successfully informing the other of its new IP address is around 10^{-7}. If the two hopping friends have a stationary joint friend, then it is enough that one of the hoppers successfully informs the stationary friend of the address change, for the other hopper to soon know this new address from their stationary common friend. Under the same assumptions, the probability of a clique of n friends loosing a member due to all hopping simultaneously is around $10^{-3.6*n}$. Note that such a loss is not terminal – assuming that friends have redundant ways to communicate (physical meetings, email, SMS, global social media), new addresses can be communicated and the digital friendship restored.

7 Applications of Cordial Dissemination

Applications of blocklace-based cordial dissemination include grassroots social networking [14], grassroots cryptocurrencies[12], as well as non-grassroots applications: The Flash payment system [11] and the Cordial Miners family of consensus protocol [10].

References

1 Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. Good-case latency of byzantine broadcast: A complete categorization. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, pages 331–341, New York, NY, USA, 2021. Association for Computing Machinery. `doi:10.1145/3465084.3467899`.

2 Paul Baran. On distributed communications networks. *IEEE transactions on Communications Systems*, 12(1):1–9, 1964.

3 Juan Benet. Ipfs-content addressed, versioned, p2p file system. *arXiv:1407.3561*, 2014.

4 Sonja Buchegger, Doris Schiöberg, Le-Hung Vu, and Anwitaman Datta. Peerson: P2p social networking: early experiences and insights. In *Proceedings of the Second ACM EuroSys Workshop on Social Network Systems*, pages 46–52, 2009.

5 Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OsDI*, volume 99, pages 173–186, 1999.

6 Gregory Chockler, Roie Melamed, Yoav Tock, and Roman Vitenberg. Constructing scalable overlays for pub-sub with many topics. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 109–118, 2007.

7 Bram Cohen. Incentives build robustness in bittorrent. In *Workshop on Economics of Peer-to-Peer systems*, volume 6, pages 68–72. Berkeley, CA, USA, 2003.

8 Adem Efe Gencer, Soumya Basu, Ittay Eyal, Robbert van Renesse, and Emin Gün Sirer. Decentralization in bitcoin and ethereum networks. In *International Conference on Financial Cryptography and Data Security*, pages 439–457. Springer, 2018.

9 Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is dag, 2021. `arXiv:2102.08325`.

10 Idit Keidar, Oded Naor, and Ehud Shapiro. Cordial miners: A family of simple and efficient consensus protocols for every eventuality. *arXiv preprint arXiv:2205.09174*, 2022.

11 Andrew Lewis-Pye, Oded Naor, and Ehud Shapiro. Flash: An asynchronous payment system with good-case linear communication complexity. *arXiv preprint arXiv:2305.03567*, 2023.

12 Ehud Shapiro. Grassroots cryptocurrencies: A foundation for a grassroots digital economy. *arXiv preprint arXiv:2202.05619*, 2022.

13 Ehud Shapiro. Grassroots distributed systems: Concept, examples, implementation and applications. *arXiv preprint arXiv:2301.04391*, 2023.

14 Ehud Shapiro. Grassroots social networking: Serverless, permissionless protocols for twitter-/linkedin/whatsapp. *arXiv preprint arXiv:2306.13941*, 2023.

15 Ehud Shapiro and Nimrod Talmon. Foundations for grassroots democratic metaverse. In *AAMAS '22: Proceedings of the 21st International Conference on Autonomous Agents and Multiagent Systems*, pages 1814–1818, 2022.

16 Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM computer communication review*, 31(4):149–160, 2001.

17 Robert Strom, Guruduth Banavar, Tushar Chandra, Marc Kaplan, Kevan Miller, Bodhi Mukherjee, Daniel Sturman, and Michael Ward. Gryphon: An information flow based approach to message brokering. *arXiv preprint cs/9810019*, 1998.

18 Dominic Tarr, Erick Lavoie, Aljoscha Meyer, and Christian Tschudin. Secure scuttlebutt: An identity-centric protocol for subjective and decentralized applications. In *Proceedings of the 6th ACM conference on information-centric networking*, pages 1–11, 2019.

19 Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.

20 Shoshana Zuboff. *The age of surveillance capitalism: The fight for a human future at the new frontier of power: Barack Obama's books of 2019*. Profile books, 2019.