

26th International Conference on Principles of Distributed Systems

OPODIS 2022, December 13–15, 2022, Brussels, Belgium

Edited by

Eshcar Hillel

Roberto Palmieri

Etienne Rivière



Editors

Eshcar Hillel

Pliops, Ramat Gan, Israel
eshcar@pliops.com

Roberto Palmieri

Lehigh University, Bethlehem, PA, USA
palmieri@lehigh.edu

Etienne Rivière

UCLouvain, Ottignies-Louvain-la-Neuve, Belgium
etienne.riviere@uclouvain.be

ACM Classification 2012

Theory of computation → Distributed computing models; Theory of computation → Distributed algorithms; Theory of computation → Concurrent algorithms; Theory of computation → Data structures design and analysis; Networks → Mobile networks; Networks → Wireless access networks; Networks → Ad hoc networks; Computing methodologies → Distributed algorithms; Security and privacy → Distributed systems security; Information systems → Distributed storage; Computer systems organization → Dependable and fault-tolerant systems and networks; Software and its engineering → Distributed systems organizing principles

ISBN 978-3-95977-265-5

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/978-3-95977-265-5>.

Publication date

February, 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>.

License

This work is licensed under a Creative Commons Attribution 4.0 International license (CC-BY 4.0): <https://creativecommons.org/licenses/by/4.0/legalcode>.



In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:

- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/LIPIcs.OPODIS.2022.0

ISBN 978-3-95977-265-5

ISSN 1868-8969

<https://www.dagstuhl.de/lipics>

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.

Editorial Board

- Luca Aceto (*Chair*, Reykjavik University, IS and Gran Sasso Science Institute, IT)
- Christel Baier (TU Dresden, DE)
- Mikolaj Bojanczyk (University of Warsaw, PL)
- Roberto Di Cosmo (Inria and Université de Paris, FR)
- Faith Ellen (University of Toronto, CA)
- Javier Esparza (TU München, DE)
- Daniel Král' (Masaryk University - Brno, CZ)
- Meena Mahajan (Institute of Mathematical Sciences, Chennai, IN)
- Anca Muscholl (University of Bordeaux, FR)
- Chih-Hao Luke Ong (University of Oxford, GB)
- Phillip Rogaway (University of California, Davis, US)
- Eva Rotenberg (Technical University of Denmark, Lyngby, DK)
- Raimund Seidel (Universität des Saarlandes, Saarbrücken, DE and Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Wadern, DE)

ISSN 1868-8969

<https://www.dagstuhl.de/lipics>

■ Contents

Preface	
<i>Eshcar Hillel, Roberto Palmieri, and Etienne Rivière</i>	0:vii
Program Committee	
.....	0:ix
Steering Committee	
.....	0:xi
External Reviewers	
.....	0:xiii

Invited Talks

Theory Meets Practice in the Algorand Blockchain	
<i>Victor Luchangco</i>	1:1–1:1
Recoverable Computing	
<i>Panagiota Fatourou</i>	2:1–2:2
Realistic Self-Stabilization	
<i>Sébastien Tixeuil</i>	3:1–3:1

Regular Papers

Efficient Wait-Free Queue Algorithms with Multiple Enqueuers and Multiple Dequeuers	
<i>Colette Johnen, Adnane Khattabi, and Alessia Milani</i>	4:1–4:19
EEMARQ: Efficient Lock-Free Range Queries with Memory Reclamation	
<i>Gali Sheffi, Pedro Ramalhete, and Erez Petrank</i>	5:1–5:22
The Step Complexity of Multidimensional Approximate Agreement	
<i>Hagit Attiya and Faith Ellen</i>	6:1–6:12
Performance Anomalies in Concurrent Data Structure Microbenchmarks	
<i>Rosina F. Kharal and Trevor Brown</i>	7:1–7:24
Robust and Fast Blockchain State Synchronization	
<i>Enrique Fynn, Ethan Buchman, Zarko Milosevic, Robert Soulé, and Fernando Pedone</i>	8:1–8:22
A Privacy-Preserving and Transparent Certification System for Digital Credentials	
<i>Rodrigo Q. Saramago, Hein Meling, and Leander N. Jehl</i>	9:1–9:24
When Is Spring Coming? A Security Analysis of Avalanche Consensus	
<i>Ignacio Amores-Sesar, Christian Cachin, and Enrico Tedeschi</i>	10:1–10:22
Computational Power of a Single Oblivious Mobile Agent in Two-Edge-Connected Graphs	
<i>Taichi Inoue, Naoki Kitamura, Taisuke Izumi, and Toshimitsu Masuzawa</i>	11:1–11:18

26th International Conference on Principles of Distributed Systems (OPODIS 2022).

Editors: Eshcar Hillel, Roberto Palmieri, and Etienne Rivière



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Line Search for an Oblivious Moving Target <i>Jared Coleman, Evangelos Kranakis, Danny Krizanc, and Oscar Morales-Ponce</i> ..	12:1–12:19
Randomized Byzantine Gathering in Rings <i>John Augustine, Arnhav Datar, and Nischith Shadagopan</i>	13:1–13:16
Gathering of Mobile Robots with Defected Views <i>Yonghwan Kim, Masahiro Shibata, Yuichi Sudo, Junya Nakamura, Yoshiaki Katayama, and Toshimitsu Masuzawa</i>	14:1–14:18
A Unifying Approach to Efficient (Near)-Gathering of Disoriented Robots with Limited Visibility <i>Jannik Castenow, Jonas Harbig, Daniel Jung, Peter Kling, Till Knollmann, and Friedhelm Meyer auf der Heide</i>	15:1–15:25
New Dolev-Reischuk Lower Bounds Meet Blockchain Eclipse Attacks <i>Ittai Abraham and Gilad Stern</i>	16:1–16:18
Quorum Systems in Permissionless Networks <i>Christian Cachin, Giuliano Losa, and Luca Zanolini</i>	17:1–17:22
Make Every Word Count: Adaptive Byzantine Agreement with Fewer Words <i>Shir Cohen, Idit Keidar, and Alexander Spiegelman</i>	18:1–18:21
Modeling Resources in Permissionless Longest-Chain Total-Order Broadcast <i>Sarah Azouvi, Christian Cachin, Duc V. Le, Marko Vukolić, and Luca Zanolini</i> ..	19:1–19:23
Computing Power of Hybrid Models in Synchronous Networks <i>Pierre Fraigniaud, Pedro Montealegre, Pablo Paredes, Ivan Rapaport, Martín Ríos-Wilson, and Ioan Todinca</i>	20:1–20:18
Mending Partial Solutions with Few Changes <i>Darya Melnyk, Jukka Suomela, and Neven Villani</i>	21:1–21:17
The Impossibility of Approximate Agreement on a Larger Class of Graphs <i>Shihao Liu</i>	22:1–22:20
On the Hierarchy of Distributed Majority Protocols <i>Petra Berenbrink, Amin Coja-Oghlan, Oliver Gebhard, Max Hahn-Klimroth, Dominik Kaaser, and Malin Rau</i>	23:1–23:19
Communication-Efficient BFT Using Small Trusted Hardware to Tolerate Minority Corruption <i>Sravya Yandamuri, Ittai Abraham, Kartik Nayak, and Michael K. Reiter</i>	24:1–24:23
Chopin: Combining Distributed and Centralized Schedulers for Self-Adjusting Datacenter Networks <i>Neta Rozen-Schiff, Klaus-Tycho Foerster, Stefan Schmid, and David Hay</i>	25:1–25:23
A Modular Approach to Construct Signature-Free BRB Algorithms Under a Message Adversary <i>Timothé Albouy, Davide Frey, Michel Raynal, and François Taïani</i>	26:1–26:23
Design of Self-Stabilizing Approximation Algorithms via a Primal-Dual Approach <i>Yuval Emek, Yuval Gil, and Noga Harlev</i>	27:1–27:19
Self-Stabilizing Clock Synchronization in Dynamic Networks <i>Bernadette Charron-Bost and Louis Penet de Monterno</i>	28:1–28:17

■ Preface

The papers in this volume were presented at the 26th International Conference on Principles of Distributed Systems (OPODIS 2022), held on December 13–15, 2022 in Brussels, Belgium. OPODIS is an open forum for the exchange of state-of-the-art knowledge about distributed computing. With strong roots in the theory of distributed systems, OPODIS has expanded its scope to cover the entire range between the theoretical aspects and practical implementations of distributed systems, as well as experimental and quantitative assessments.

All aspects of distributed systems are within the scope of OPODIS: theory, specification, design, performance, and system building. Specifically, this year, the topics of interest at OPODIS included:

- Distributed systems, theory and practice
- Blockchain, theory and practice
- Cloud and data centers
- Communication and mobile networks
- Parallelism, concurrency, and multicore systems
- Shared and transactional memory, memory management
- Dependable systems, system security
- Distributed graph algorithms
- Middleware and Operating systems
- File and storage systems
- Distributed ML
- Distributed data analytics
- Mobile agents and robots
- Self-stabilizing, self-organizing and autonomous systems
- Game-theory in distributed computing

We received 76 submissions, each of which underwent a double-blind peer review process. Overall, the quality of the submissions was very high. From the 76 submissions, 25 papers were selected to be included in these proceedings. To emphasize the system side of distributed computing, this year in addition to an academic forum the program committee included representatives from 9 industrial companies. Authors of more than a quarter of the accepted papers have an industrial affiliation.

The program committee decided to honor Hagit Attiya and Faith Ellen with the OPODIS 2022 Best Paper Award for their work on “The Step Complexity of Multidimensional Approximate Agreement”. A Best Student Paper Award was presented to Ittai Abraham and Gilad Stern for their paper “New Dolev-Reischuk Lower Bounds Meet Blockchain Eclipse Attacks”. In addition, the paper “Computational Power of a Single Oblivious Mobile Agent in Two-Edge-Connected Graphs” by Taichi Inoue, Naoki Kitamura, Taisuke Izumi, and Toshimitsu Masuzawa was recognized as a runner-up for the Best Student Paper Award.

The OPODIS proceedings appear in the Leibniz International Proceedings in Informatics (LIPIcs) series. LIPIcs proceedings are available online and free of charge to readers. The production costs are paid in part from the conference budget.

This year OPODIS had three distinguished invited keynote speakers: Panagiota Fatourou (University of Crete), Victor Luchangco (Algorand), and Sébastien Tixeuil (Sorbonne University, CNRS, LIP6, Institut Universitaire de France, France). We warmly thank all the authors

26th International Conference on Principles of Distributed Systems (OPODIS 2022).

Editors: Eshcar Hillel, Roberto Palmieri, and Etienne Rivière



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

that submitted their work to OPODIS. We are also grateful to the Program Committee members for their hard work reviewing papers and their active participation in the online discussions and the Program Committee meeting. We also thank the external reviewers for their help with the reviewing process.

The conference organization committee expresses its warmest regards to the sponsors of the conference: Input-Output (USA), Digital Wallonia and the Walchain network (Belgium), and the FNRS public research agency (Belgium). Organizing this event would not have been possible without the dedication of researchers from the Cloud and Large-Scale Computing group at UCLouvain and from support staff members of UCLouvain's computer science department. We also express our thanks to Steering Committee members for their valuable advice.

November 2022

Eshcar Hillel (Pliops, Israel)
Roberto Palmieri (Lehigh University, USA)
Etienne Rivière (UCLouvain, Belgium)

■ Program Committee

General Chair

Etienne Rivière (UCLouvain, Belgium)

Program Chairs

Eshcar Hillel (PLIOPS, Israel)

Roberto Palmieri (Lehigh University, USA)

Program Committee

Vitaly Aksenov (ITMO University, Russia)

Emmanuelle Anceaume (CNRS, France)

Masoud Ardekani (Google, USA)

Hagit Attiya (Technion, Israel)

Amir Bar-Or (AWS, USA)

Alysson Bessani (LASIGE and FCUL, Universidade de Lisboa, Portugal)

Silvia Bonomi (Sapienza, University of Rome, Italy)

Anastasia Braginsky (Technion, Israel)

Quentin Bramas (University of Strasbourg, France)

Armando Castaneda (UNAM, Mexico)

Bapi Chatterjee (IIIT-Delhi, India)

Shir Cohen (Technion, Israel)

Antonella Del Pozzo (CEA List, France)

Stéphane Devismes (Université de Picardie Jules Verne, France)

Giuseppe Antonio Di Luna (Sapienza, University of Rome, Italy)

Liran Funaro (IBM, Israel)

Alexey Gotsman (IMDEA Software Institute, Spain)

Guy Gueta (VMware, Israel)

Ahmed Hassan (Lehigh University, USA)

Alex Kogan (Oracle, USA)

Miguel Matos (Universidade de Lisboa & INESC-ID, Portugal)

Dennis Olivetti (Gran Sasso Science Institute, Italy)

Fernando Pedone (Università della Svizzera italiana, Switzerland)

Sebastiano Peluso (Meta, USA)

Maria Potop-Butucaru (LIP6, Sorbonne University, France)

Paolo Romano (INESC/IST, Portugal)

Valerio Schiavoni (University of Neuchâtel, Switzerland)

Rana Shahout (Technion, Israel)

Alexander Spiegelman (Aptos, USA)

Ram Sriharsha (Pinecone, USA)

Pierre Sutra (Télécom SudParis, France)

Sébastien Tixeuil (Sorbonne University & Institut Universitaire de France, France)

Lewis Tseng (Boston College, USA)

Jennifer L. Welch (Texas A&M University, USA)

Haibin Zhang (Beijing Institute of Technology, China)



■ Steering Committee

Panagiota Fatourou (University of Crete, Greece)
Pascal Felber (Université de Neuchâtel, Switzerland) – chair
Paola Flocchini (University of Ottawa, Canada)
Vincent Gramoli (University of Sydney, Australia)
Yannic Maus (TU Graz, Austria)
Alessia Milani (LIS, Aix-Marseille Université, France)
Paolo Romano (INESC-ID, University of Lisbon, Portugal)
Rotem Oshman (Tel-Aviv University, Israel)



■ External Reviewers

Anais Durand, LIMOS, Université Clermont Auvergne
Weiming Feng, University of Edinburgh
Rati Gelashvili, Aptos
Colette Johnen, Université de Bordeaux, LaBRI, CNRS
Yacov Manevich, IBM
Thomas Nowak, ENS Paris-Saclay
Sergio Rajsbaum, National Autonomous University of Mexico
Noa Schiller, Technion
Corentin Travers, LaBRI
Nitin Vaidya, Georgetown University
Zhuolun Xiang, Aptos



Theory Meets Practice in the Algorand Blockchain

Victor Luchangco ✉

Algorand, Inc., Boston, MA, USA

Abstract

Robust and effective distributed systems require good theory and good engineering, not separately but in concert: user requirements and system constraints are not merely implementation details but often must inform the design of algorithms for such systems. Blockchains are an excellent example. The heart of a blockchain is its (Byzantine) consensus protocol, and consensus protocols have been extensively studied in the theory community for decades. But traditional consensus protocols are not directly applicable to blockchains, which have, or hope to have, millions of participants. Furthermore, public blockchains, which allow anyone to participate, must have some mechanism to guarantee the security of the protocol, and traditional fault models do not adequately capture the assumptions of such mechanisms. In this talk, I will discuss these and other ways in which theory and practice meet in the context of the Algorand blockchain, and how Algorand is able to achieve high transaction throughput with low latency.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Computer systems organization → Dependable and fault-tolerant systems and networks

Keywords and phrases Theory and practice, Design of distributed systems, Blockchain, Consensus, Algorand

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2022.1

Category Invited Talk



© Victor Luchangco;

licensed under Creative Commons License CC-BY 4.0

26th International Conference on Principles of Distributed Systems (OPODIS 2022).

Editors: Eshcar Hillel, Roberto Palmieri, and Etienne Rivière; Article No. 1; pp. 1:1–1:1

Leibniz International Proceedings in Informatics



LIPIC Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Recoverable Computing

Panagiota Fatourou ✉

Institute of Computer Science, Foundation for Research and Technology-Hellas, Heraklion, Greece
Department of Computer Science, University of Crete, Heraklion, Greece

Abstract

Non-Volatile Memory (NVM) is an emerging memory technology which aims to address the high computational demands of modern applications and support recovery from crashes. Recovery ensures that after a crash every executed operation is able to recover and return a correct response. This talk will shed light on different aspects of the question “*How does concurrent computing change in systems with NVM and what will the impact of persistent memory be on the way we compute?*”. Specifically, this talk addresses the following four main challenges in NVM computing.

- **Challenge 1:** *How to appropriately model and abstract fundamental aspects of NVM computing?* The talk will provide an overview of the theoretical framework for NVM computing, including a discussion of correctness conditions, progress guarantees, failure types, etc.
- **Challenge 2:** *How to compute in a recoverable way at no significant cost?* The talk will summarize state-of-the-art generic approaches for deriving recoverable synchronization algorithms, as well as recoverable implementations of many widely-used concurrent data structures on top of them. The collection of data structures includes fundamental structures, such as stacks and queues, but also more complex structures that implement sets, such as linked-lists and trees.
- **Challenge 3:** *How to analyze the cost of recoverable algorithms?* The talk will present a way of analyzing the cost of persistence instructions, not by simply counting them but by separating them into categories based on the impact they have on the performance. This analysis reveals that understanding the actual persistence cost of an algorithm in machines with NVM, is more complicated than previously thought, and requires a thorough evaluation, since the performance impact of different persistence instructions may greatly vary.
- **Challenge 4:** *When is Recoverable Consensus Harder Than Consensus?* The talk will briefly discuss the ability of different shared object types to solve recoverable consensus using NVM when processes crash and recover, and it will compare the difficulty of solving recoverable consensus to the difficulty of solving the standard consensus problem in a system with halting failures.

For each of the above challenges, the talk will present main results, provide some of the details of the best-performing techniques, and discuss open problems and directions for further research. Some of the results that will be discussed in detail have appeared in [1, 2, 3].

2012 ACM Subject Classification Theory of computation → Distributed computing models; Theory of computation → Concurrent algorithms; Theory of computation → Data structures design and analysis

Keywords and phrases non-volatile memory, persistence, detectability, durability, recoverable algorithms, recoverable data structures, persistent objects, stacks, queues, heaps, synchronization, universal constructions, software combining, lock-freedom, wait-freedom, persistence cost analysis

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2022.2

Category Invited Talk

Supplementary Material *Text (Slides of the Talk):* <https://sites.uclouvain.be/OPODIS2022/>



© Panagiota Fatourou;

licensed under Creative Commons License CC-BY 4.0

26th International Conference on Principles of Distributed Systems (OPODIS 2022).

Editors: Eshcar Hillel, Roberto Palmieri, and Etienne Rivière; Article No. 2; pp. 2:1–2:2

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

References

- 1 H. Attiya, O. Ben-Baruch, P. Fatourou, D. Hendler, and E. Kosmas. Detectable recovery of lock-free data structures. In *Proc. of the 2022 PPOPP*, pages 262–277, 2022.
- 2 C. Delporte-Gallet, P. Fatourou, H. Fauconnier, and E. Ruppert. When is recoverable consensus harder than consensus? In *Proc. of the 2022 PODC*, pages 198–208, 2022.
- 3 P. Fatourou, N. D. Kallimanis, and E. Kosmas. The performance power of software combining in persistence. In *Proc. of the 2022 PPOPP*, pages 337–352, 2022 (Best Paper Award).

Realistic Self-Stabilization

Sébastien Tixeuil ✉

Sorbonne University, CNRS, LIP6, Institut Universitaire de France, Paris, France

Abstract

It is almost fifty years since Dijkstra coined the term “*self-stabilization*” to denote a distributed system able to recover correct behavior starting from any arbitrary (even unreachable) configuration. His seminal paper triggered many works since then, exploring over the years new variants of the original concept, new application domains, and new complexity results. While the huge majority of those contributions relates to theory, considering computability and worst case complexity issues, this talk revisits old and recent contributions from the prism of “realistic” distributed systems, aiming to address the following question: *is self-stabilization relevant in practice for distributed systems?*

2012 ACM Subject Classification Theory of computation → Distributed computing models; Theory of computation → Distributed algorithms; Networks → Mobile networks; Computing methodologies → Distributed algorithms; Security and privacy → Distributed systems security; Computer systems organization → Dependable and fault-tolerant systems and networks

Keywords and phrases Self-stabilization, Distributed systems, Probable stabilization, Performance evaluation, Asynchronous message passing, Multi-tolerance

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2022.3

Category Invited Talk



© Sébastien Tixeuil;

licensed under Creative Commons License CC-BY 4.0

26th International Conference on Principles of Distributed Systems (OPODIS 2022).

Editors: Eshcar Hillel, Roberto Palmieri, and Etienne Rivière; Article No. 3; pp. 3:1–3:1

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Efficient Wait-Free Queue Algorithms with Multiple Enqueuers and Multiple Dequeuers

Colette Johnen ✉

Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800, Talence, France

Adnane Khattabi ✉

Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800, Talence, France

Alessia Milani ✉

Aix Marseille Univ, CNRS, LIS, UMR 7020, Marseille, France

Abstract

Despite the widespread usage of FIFO queues in distributed applications, designing efficient **wait-free** implementations of queues remains a challenge. The majority of wait-free queue implementations restrict either the number of dequeuers or the number of enqueuers that can operate on the queue, even when they use strong synchronization primitives, like the *Compare&Swap*. If we do not limit the number of processes that can perform enqueue and dequeue operations, the best-known upper bound on the worst case step complexity for a wait-free queue is given by Khanchandani and Wattenhofer [10]. In particular, they present an implementation of a multiple dequeuer multiple enqueuer wait-free queue whose worst case step complexity is in $O(\sqrt{n})$, where n is the number of processes. In this work, we investigate whether it is possible to improve this bound. In particular, we present a wait-free FIFO queue implementation that supports n enqueuers and k dequeuers where the worst case step complexity of an *Enqueue* operation is in $O(\log n)$ and of a *Dequeue* operation is in $O(k \log n)$.

Then, we show that if the semantics of the queue can be relaxed, by allowing concurrent *Dequeue* operations to retrieve the same element, then we can achieve $O(\log n)$ worst-case step complexity for both the *Enqueue* and *Dequeue* operations.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Theory of computation → Distributed algorithms; Theory of computation → Proof complexity

Keywords and phrases Distributed computing, distributed algorithms, FIFO queue, shared memory, fault tolerance, concurrent data structures, relaxed specifications, complexity

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2022.4

Funding *Adnane Khattabi*: Adnane Khattabi is supported by UMI Relax.

1 Introduction

1.1 Context

Shared FIFO queues are an important building block for the design of many concurrent applications. Many implementations of concurrent FIFO queues have been proposed using shared objects provided by multiprocessor architectures, e.g. *Compare&Swap*, registers, *Fetch&Add*, and so on. In this paper, we are interested in **wait-free** implementations of shared queues where any operation by a correct process is guaranteed to terminate after a finite number of steps.

The design of efficient wait-free and linearizable concurrent queues is a difficult task even if the implementation is allowed to rely on strong synchronization primitives like *Compare&Swap*. Most implementations limit either the number of enqueuers or the number of dequeuers. In particular, David [3] presents a wait-free linearizable queue with a single enqueuer and multiple dequeuers with constant step complexity. Jayanti and Petrovic [9]



© Colette Johnen, Adnane Khattabi, and Alessia Milani;
licensed under Creative Commons License CC-BY 4.0

26th International Conference on Principles of Distributed Systems (OPODIS 2022).

Editors: Eshcar Hillel, Roberto Palmieri, and Etienne Rivière; Article No. 4; pp. 4:1–4:19



Leibniz International Proceedings in Informatics

LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

provide an implementation of a multiple enqueueer, single dequeueer queue with $O(\log n)$ worst-case step complexity, where n is the number of processes. More recently, Khanchandani and Wattenhofer proposed a multiple enqueueer and multiple dequeueer wait-free queue implementation where both the enqueue and the dequeue operations have a worst-case step complexity of $O(\sqrt{n})$. In this paper, we investigate if this complexity represents the cost necessary in order to not limit the number of processes that can apply enqueue and dequeue operations on the concurrent queue.

By extension of algorithmic ideas from [9], we first show that a better complexity can be achieved even with multiple enqueueers and multiple dequeueers. In particular, we present a wait-free linearizable concurrent queue for n processes from which all n are enqueueers and $k \leq n$ are dequeueers. In our implementation, the step complexity of an *Enqueue* operation is in $O(\log n)$, while the complexity of a *Dequeue* operation is in $O(k \log n)$. Our implementation has logarithmic complexity as long as k is a constant. Also, it improves on the implementation by Khanchandani and Wattenhofer solution as long as $k \in O(\frac{\sqrt{n}}{\log n})$.

Then, we show that both *Enqueue* and *Dequeue* operations can have worst-case step complexity in $O(\log n)$, if we allow concurrent *Dequeue* operations to return the same element. This relaxed semantic denoted *multiplicity* has been formalized and introduced for the FIFO queue in [1]. Table 1 summarizes the state of the art and compares it to the contributions in this work.

■ **Table 1** Comparing the contributions to state-of-the-art queue implementations (n is the number of processes and m is the number of enqueued elements).

	Step complexity	Space complexity	Concurrency limit	CAS - LL/SC	Fetch&Inc - Swap
Khanchandani and Wattenhofer [10]	$O(\sqrt{n})$	$O(nm)$ of $O(\max(\log n, \log m))$ registers	None	Y	Y
David [3]	$O(1)$	Unbounded	Single enqueueer	N	Y
Jayanti and Petrovic [9]	$O(\log n)$	$O(n + m)$	Single dequeueer	Y	N
Li [13]	$O(m)$	Unbounded	2 dequeueers	N	Y
Eisenstat [4]	$O(m)$	Unbounded	2 enqueueers	N	Y
Exact queue (this work)	$O(\log n)$ for Enq $O(k \log n)$ for Deq	Unbounded	k dequeueers	Y	Y
Relaxed queue (this work)	$O(\log n)$	Unbounded	None	Y	Y

1.2 Other Related Work

Several papers propose wait-free linearizable shared queue implementations that only use registers and `Common2` objects (a particular set of base objects with consensus number 2). All of them limit the concurrency. In particular, there are queues shared by one or two dequeueers and any number of enqueueers [8, 13] and a queue with a single enqueueer and any number of dequeueers [3]. In fact, it is a long-standing open problem if it is possible to implement a wait-free linearizable queue that supports at least three enqueueers and three dequeueers based only on registers and consensus 2 objects. Among all the aforementioned queue implementations, only the one by David [3] has sublinear step complexity.

Using *Compare&Swap*, some practical wait-free queue implementations that support multiple enqueueers and multiple dequeueers have been proposed [5, 12, 14, 16]. Some of these implementations are wait-free [5, 12, 16]; while some are only lock-free [14]. All these solutions have been evaluated empirically and do not have formal complexity analysis. Nonetheless, the worst-case step complexity of either the *Enqueue* or of the *Dequeue* operation is not sublinear.

More recently, relaxed queues have been proposed to overcome the complexity of implementing queues. For instance, in [6], Henzinger et al. formalize the definition of the *c-out-of-order* queue where an element at a distance up to $c - 1$ from the element in the head of the queue, is allowed to be dequeued. A linearizable and lock-free *c-out-of-order* queue with no concurrency constraints is implemented in [11] using the *CAS* primitive. In [1], a lock-free implementation of a queue with *multiplicity* where only concurrent *Dequeue* operations can return the same element, is given under the coherence condition of set-linearizability. This implementation has no concurrency constraint and uses only *Read/Write* primitives. In both these implementations, the *Dequeue* operation's worst-case step complexity is unbounded since it depends on the number of *Enqueue* operations executed. Regarding practical applications, [2] discusses possible applications of the multiplicity relaxation such as relaxed work-stealing for parallel SAT solvers.

Simply by considering an execution where a process only executes *Enqueue* operations, we can show a lower bound on space complexity in the number of elements present in the queue. However, besides this space requirement, there has been some work in optimizing the space complexity of queue implementations using memory reclamation (e.g. [3, 16]). We do not consider the issue of optimizing the space complexity and leave the question for future work.

Paper organization. In Section 2 we present the model. In Section 3, we describe our linearizable wait-free multiple enqueueer multiple dequeuer queue implementation together with its correctness proof. Finally, we present the relaxed queue implementation with multiplicity in Section 4.

2 Preliminaries

We consider a standard asynchronous shared memory model, consisting of a set \mathcal{P} of n crash-prone processes with unique ids, where all n processes can be enqueueers and $k \leq n$ can be dequeuers. We also refer to this set of processes as a set of n enqueueers and k dequeuers.

Processes communicate by applying primitive operations to shared base objects. In particular, we consider *registers*, *Fetch&Inc*, *Compare&Swap*, and *Max registers*. A *register* provides atomic *Read/Write* primitives. The *Fetch&Inc* object provides a *Fetch&Inc* primitive that increments the value of the object by 1 and returns the previous value. The *Compare&Swap* object supports the *Read* and the *CAS* primitives. The *Read* simply returns the value of the object. The call to *CAS(old, new)* writes *new* into the object only if the current value of the object is equal to *old* and in that case, it returns *True*, otherwise, it returns *False*.

The *max register* supports two primitives : *MaxWrite(v)* that writes the value v into the register, and *MaxRead()* that returns the largest value written so far. Modern architectures do not implement the max register object. However, our algorithm uses max registers in a restricted way (essentially, each new value written increments the previous value by one), thus we can easily implement the *MaxWrite(v)* and *MaxRead()* operations by applying a constant number of primitives on *CAS* objects.

The FIFO *queue* provides the two high-level operations *Enqueue(v)* and *Dequeue()*. An *Enqueue(v)* operation adds the element v at the tail of the queue, while the *Dequeue()* operation removes the element at the head of the queue and returns its value, if the queue is not empty, otherwise it returns a special value ϵ .

An *implementation* of a shared object provides a specific data-representation for the object from a set of *base objects*, each of which is assigned an initial value; the implementation also provides algorithms for each process in \mathcal{P} to apply each operation to the object being implemented. To avoid confusion, we call operations on the base objects *primitives* and reserve the term *operations* for the FIFO queue object being implemented.

An *execution* of an implementation of a shared object is a sequence of steps (possibly infinite), where a step is either the application of a primitive operation on a base object or an invocation/response of an operation of the high-level implemented object. An execution is *well-formed* if each process is sequential and if it invokes a new high-level operation only after it has completed the current one. The steps taken by a process during the execution of a high-level operation are defined by the algorithms provided by the implementation of the shared object.

If an operation op_1 returns before another operation op_2 is invoked, we say that op_1 precedes op_2 in real-time order, denoted $op_1 <_{ro} op_2$.

Roughly speaking, an implementation is *linearizable* [8] if each operation appears to take effect atomically at some point between its invocation and response; it is *wait-free* [7] if each process completes its operation if it performs a sufficiently large number of steps.

To define the relaxed FIFO queue, we consider the formalism of *set-linearizability* provided in [15]. Roughly speaking, set-linearizability allows for multiple concurrent operations to be linearized at the same point. Such a linearization point would fall within the execution interval of all the concurrent operations. The set-linearization of an execution E is defined by ordering different sets of the operations in E , such that the operations in a set are executed concurrently. The *FIFO queue with multiplicity* [1] is a relaxed FIFO queue such that its specification allows multiple concurrent *Dequeue()* operations to return the same value.

3 Wait-Free Linearizable Queue

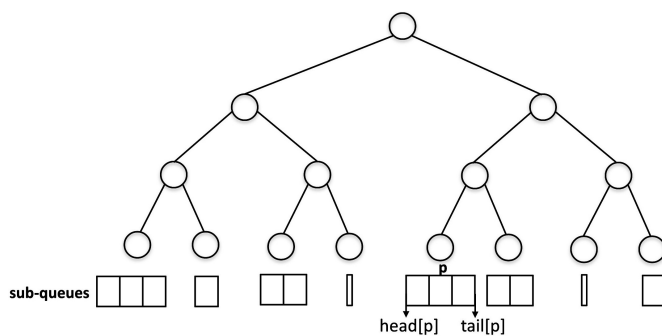
3.1 Algorithm overview

We present hereafter a conceptual overview of the algorithm implementing the k -dequeuer n -enqueuer concurrent queue.

As depicted in Figure 1, the queue object can be seen as n different *sub-queues* such that when an *Enqueue(v)* operation is invoked by an enqueueer process p , the element v is enqueued in the corresponding p -th *sub-queue*. Each enqueued element is associated with a unique timestamp, used by the dequeuers to select the element to be returned (if any).

In particular, each enqueued element is associated to a pair (st, p) where st is the value of a shared counter, and p is the *id* of the process that invoked the corresponding *Enqueue* operation. Two processes executing concurrent *Enqueue(v)* operations can retrieve the same value from the shared counter, but the process *id* makes each timestamp unique. Timestamps are totally ordered according to the lexicographical order. The timestamps associated with the elements in a given sub-queue reflect the real-time order of *Enqueue()* operations by the same process. In particular, if an element e is enqueued in a sub-queue p before another element e' , then e is associated with a smaller timestamp than e' . This also means that the head of the sub-queue has the smallest timestamp among the other elements in the same sub-queue.

For the sake of complexity, the timestamps are organized in a tree structure where the n leaves correspond to the timestamps of the elements at the head of the corresponding n sub-queues, and the root stores the smallest timestamp among the ones in the leaves. Our construction is similar to the one proposed by Jayanti et al. in [9].



■ **Figure 1** Data structure for the k -dequeuer n -enqueuer queue implementation.

Thus, a *Dequeue* operation simply reads the root of the tree and returns the corresponding element in the appropriate sub-queue in the same manner that this is done in the *single dequeuer* queue in [9]. However, to support k different dequeuer processes, we need to manage the concurrency between their operations. This is done by introducing a helping mechanism for the *Dequeue* operation. In particular, each *Dequeue* operation has a unique sequence number. Before executing its instance of *Dequeue* operation, a process will first ensure that the instances with smaller sequence numbers are not more pending. If they are, the process will execute the steps necessary for them to finish, and it will update the tree before executing its own instance of *Dequeue*. Since there are k dequeuer processes, during an instance of *Dequeue*, there could be at most $k - 1$ other processes executing a *Dequeue* operation concurrently.

3.2 Algorithm Pseudocode

In the implementation of the multiple dequeuer and multiple enqueuer queue in Algorithms 1–2, we use two main data structures: a two-dimensional array of registers, called *items*, where each row p together with two integers $head[p]$ and $tail[p]$ represents the sub-queue of process p ; and a balanced binary tree T with n leaves where each node is a *CAS* object used to store the timestamps of enqueued elements.

The sub-queue p contains the elements enqueued by process p that have not been dequeued, i.e. the current sub-queue p is defined by its values h and t of the max register $head[p]$ and the register $tail[p]$ respectively. If $h = t$, the sub-queue p is empty. Otherwise, it is the ordered list of $t - h$ elements: $items[p][h], \dots, items[p][t - 1]$.

Each *Enqueue* operation executed by process p is associated with a unique timestamp (st, p) where st is an integer obtained from the counter *enqCounter*, and p is the process id. The empty queue is associated with a special timestamp $(\epsilon, -1)$, and we consider that $\epsilon > i \forall i \in \mathbb{N}$. $items[p][i] = (val, (st, p))$ means that the i -th *Enqueue* operation by p has enqueued the value val , and that this *Enqueue* has the timestamp (st, p) .

The smallest timestamp of a sub-queue p is the timestamp value of $items[p][h]$ where h is the current value of the head of the sub-queue. This timestamp is stored in the p -th leaf of the tree T associated with p , called p -leaf. The following details the different functions of the implementation in Algorithms 1–2.

- *Enqueue*(v): when process p calls an instance of *Enqueue*(v), it starts by constructing the corresponding timestamp (st, p) by reading the value of *enqCounter*. Process p will then write $(v, (st, p))$ to $item[p][t]$ where t is the value of $tail[p]$. Then, it updates the value of $tail[p]$ to $t + 1$. Afterward, the value $st + 1$ is written to the max register *enqCounter* to

■ **Algorithm 1** Wait-free queue implementation (pseudo-code for process p).

```

1 Shared variables
2   enqCounter : Max register object, initially 0.
3   deqCounter : Fetch&Inc object, initially 1.
4   head[ $n$ ] : Array of Max register objects, initially 0.
5   tail[ $n$ ] : Array of registers where each register contains an integer, initially 0.
6   items[ $n$ ][ $\dots$ ] : Two dimensional array of registers, each register contains the uplet
   (val, (st, it)) initially  $(\perp, (\perp, \perp))$ .
7    $T$  : binary tree of CAS objects with  $n$  leaves, each node contains the pair  $(st, id)$ , all
   initially  $(\epsilon, -1)$ .
8   deqOps[ $\dots$ ] : Array of CAS objects, initially  $(\perp, \perp)$ . deqOps[ $j$ ] =  $(i, id)$  means that
   the  $j$ -th Dequeue operation returns items[id][ $i$ ].val if  $id \neq -1$ , otherwise the
   operation returns  $\epsilon$ .

9 Function Enqueue( $v$ )
10    $st \leftarrow \text{enqCounter.MaxRead}()$ 
11    $t \leftarrow \text{tail}[p]$ 
12    $\text{items}[p][t] \leftarrow (v, (st, p))$ 
13    $\text{tail}[p] \leftarrow \text{tail}[p] + 1$ 
14    $\text{enqCounter.MaxWrite}(st + 1)$ 
15   Propagate( $p$ )
16   return True

17 Function Dequeue()
18    $num \leftarrow \text{deqCounter.Fetch\&Inc}()$ 
19   for ( $i \leftarrow \max(1, num - k + 1); i \leq num; i++$ ) do
20     if  $\text{deqOps}[i].\text{Read}() = (\perp, \perp)$  then
21       if  $i > 1$  then
22          $\text{UpdateTree}(i - 1)$ 
23          $\text{FinishDeq}(i)$ 
24      $(j, id) \leftarrow \text{deqOps}[num].\text{Read}()$ 
25     if  $id = -1$  then
26       return  $\epsilon$ 
27     else
28        $(ret, -) \leftarrow \text{items}[id][j]$ 
29     return  $ret$ 

```

ensure that all subsequent *Enqueue* operations will have a greater timestamp than (st, p) . Finally, process p calls *Propagate*(p) to update the timestamps in the nodes of the tree T from the p -leaf to the root, if necessary.

- *Refresh*(*node*, *isLeaf*): this function is invoked during the execution of an instance of *Propagate* to reset the timestamp stored in a *node*. If the boolean *isLeaf* is equal to *True*, the current node represents a leaf of the tree T . In this case, the operation computes the minimum timestamp in the corresponding sub-queue. This value is either (1) $(\epsilon, -1)$ if the sub-queue is empty (line 16 of Algorithm 2); or a timestamp (2) (st', i) (line 18 of Algorithm 2). If *isLeaf* = *False* then *node* is not a leaf; the operation reads the

■ **Algorithm 2** Auxiliary functions to the queue implementation.

```

1 Function Propagate(id)
2   currentNode  $\leftarrow$  leaf( $\mathbb{T}$ , id)
3   if !Refresh(currentNode, True) then
4     | Refresh(currentNode, True)
5   do
6     | currentNode  $\leftarrow$  parent(currentNode)
7     | if !Refresh(currentNode, False) then
8       | Refresh(currentNode, False)
9   while currentNode  $\neq$  root( $\mathbb{T}$ )

10 Function Refresh(node, isLeaf)
11   (st, id)  $\leftarrow$  node.Read()
12   if isLeaf then
13     | h  $\leftarrow$  head[id].MaxRead()
14     | t  $\leftarrow$  tail[id]
15     | if h = t then
16       | ret  $\leftarrow$  node.CAS((st, id), ( $\epsilon$ , -1))
17     | else
18       | ( $-$ , (st',  $-$ ))  $\leftarrow$  items[id][h]
19       | ret  $\leftarrow$  node.CAS((st, id), (st', id))
20     | return ret
21   else
22     | (min_st, min_id)  $\leftarrow$  read minimum timestamp in current node's children
23     | return node.CAS((st, id), (min_st, min_id))

24 Function FinishDeq(num)
25   ( $-$ , id)  $\leftarrow$  root( $\mathbb{T}$ ).Read()
26   if id = -1 then
27     | deqOps[num].CAS(( $\perp$ ,  $\perp$ ), ( $\epsilon$ , -1))
28   else
29     | h  $\leftarrow$  head[id].MaxRead()
30     | deqOps[num].CAS(( $\perp$ ,  $\perp$ ), (h, id))

31 Function UpdateTree(num)
32   (j, id)  $\leftarrow$  deqOps[num].Read()
33   if id  $\neq$  -1 then
34     | head[id].MaxWrite(j + 1)
35     | Propagate(id)

```

timestamps stored in the children of the current node to compute the minimal timestamp. Then, in both cases, the operation executes the *CAS* primitive on *node* to write the timestamp and returns the resulting boolean.

- *Propagate(id)*: updates the nodes of the tree \mathbb{T} in the path from the *id*-leaf node to the root. Specifically, the function relies on calls to *Refresh* while traversing the path to update each individual *node*. To ensure that the value written into a node is up to date, the call to the function *Refresh*(*node*, $-$) is repeated if the first call fails because a

concurrent instance r_1 of $Refresh(node, -)$ might have written an outdated value since r_1 started before the call to $Refresh(node, -)$ in $Propagate(id)$. However, after the second call to $Refresh(node, -)$, we are certain that the value written is up to date because it can only be written by an instance invoked after $Propagate(id)$. This technique is used in the implementation of the single dequeuer multiple enqueueer queue in [9].

- *Dequeue*: First, an instance of the *Dequeue* operation executed by a process p , computes its unique sequence number num by applying a *Fetch&Inc* primitive on $deqCounter$. Then, p executes the helping mechanism to assist any pending *Dequeue* operation with a sequence number $i \in [max(1, num - k + 1), num]$ in increasing order of i . If the operation with the index i is still pending (i.e. $deqOps[i]$ is still set to its initial value), p executes $UpdateTree(i - 1)$ if $i > 1$, to ensure that the root of the tree is updated to an accurate value. Then, p executes $FinishDeq(i)$ to decide on the operation's return value in $deqOps[i]$. After the return values have been decided for all *Dequeue* operations with indexes in $[max(1, num - k + 1), num]$, p reads $deqOps[num] = (i, j)$ and returns $items[j][i].val$, otherwise p returns ϵ .
- *FinishDeq(num)*: The array $DeqOps$ stores the information regarding the return values of each *Dequeue* operation. A call to *FinishDeq* with the parameter num decides a value and attempts to write it to $DeqOps[num]$ using a *CAS* primitive. *FinishDeq(num)* reads the timestamp at the root of the tree $T : (-, id)$. And if $id = -1$ (i.e. the queue is empty), then $(\epsilon, -1)$ is written to $DeqOps[num]$. Otherwise, the value (h, id) is written to $DeqOps[num]$ where h is the value of the head of the sub-queue id . In either scenario, if the *CAS* instruction fails, another process has succeeded in executing a *CAS* instruction on $DeqOps[num]$ and the return value for the corresponding *Dequeue* has been decided.
- *UpdateTree(num)*: A simple function call that encapsulates the steps necessary before executing the *Dequeue* operation with the sequence number $num + 1$. If the *Dequeue* operation with the sequence number num returns ϵ , then there are no additional steps necessary. Otherwise, it is necessary to update the head of the sub-queue id from which the return value was retrieved; followed by a call to the function $Propagate(id)$ to update the tree accordingly.

3.3 Proof

In this section, we establish that Algorithms 1–2 are a wait-free implementation of a k -dequeuer multi-enqueueer queue. We also establish that an *Enqueue* operation has a worst-case step complexity of $O(\log n)$ and a *Dequeue* operation has a worst-case step complexity of $O(k \log n)$.

3.3.1 Algorithm properties

Each *Dequeue* operation is associated with a unique sequence number that is the value obtained by applying the *Fetch&Inc* primitive on $deqCounter$ at line 18 of Algorithm 1.

► **Lemma 1.** *A total order between Dequeue operations is provided by their sequence number. This order respects the real-time order.*

Proof. Let deq_1 and deq_2 be two *Dequeue* operations by process p_1 and p_2 respectively. Let seq_1 be the sequence number of deq_1 and seq_2 be the sequence number of deq_2 . We prove that if deq_1 precedes deq_2 in real-time order, then $seq_1 < seq_2$.

deq_1 completes before deq_2 is invoked, thus p_1 executes line 18 of Algorithm 1 before the invocation of deq_2 by p_2 . The proof follows from the fact that $deqCounter$ is a linearizable *Fetch&Inc* object. ◀

The *Dequeue* operation with the sequence number i is complete at a given configuration C if $DeqOps[i] \neq (\perp, \perp)$ (i.e.; the value of $DeqOps[i]$ at C is not the initial value). Otherwise, it is incomplete at C .

► **Observation 2.** *Let deq denote a Dequeue operation with the sequence number i . Any call to $FinishDeq(i)$ is executed after the invocation of deq .*

► **Lemma 3.** *Fix an execution E and let C be any configuration of E . $\forall h > 0$ and $\forall i \geq 1$, if the $h + i$ -th Dequeue operation exists and it is complete at C , then the i -th Dequeue operation is complete at C .*

Proof. Consider the first configuration C where the $h + i$ -th Dequeue operation is complete, i.e.; $deqOps[i + h] \neq (\perp, \perp)$. Assume by contradiction that $deqOps[i]$ has its initial value at C .

The value of $deqOps[i]$ is only set during the execution of $FinishDeq(i)$ at line 30 or 27 of Algorithm 2. According to the condition in the for-loop (line 19 of Algorithm 1), only a Dequeue operation with a sequence number $i + h \leq l \leq i + h + k - 1$ may change the value of $deqOps[i + h]$.

According to Lemma 1, the Dequeue operations with a sequence number smaller than or equal to l , and in particular $\in [i, l]$, have started at the configuration immediately before the value of $deqOps[i + h]$ is changed by the l -th Dequeue operation. Also, the Dequeue operations with a sequence number $num \in [i, i + k - 1]$ could not have returned at C otherwise $deqOps[i] \neq (\perp, \perp)$ at C (contradicting our assumption). This is trivially true for $num = i$. For $num \in [i + 1, i + k - 1]$, and since the condition at line 20 of Algorithm 1 is *true* for $deqOps[i]$, the Dequeue operation with sequence number num will execute the $FinishDeq(i)$ function and set $deqOps[i] \neq (\perp, \perp)$ before it returns.

Thus, l should be greater than $i + k - 1$. But this means that there are $k + 1$ pending Dequeue operations, which contradicts the fact that we can have at most k pending Dequeue operations. There is a contradiction. ◀

As $deqOps[num]$ is updated only during the execution of the function $FinishDeq(num)$; the following observation is a consequence of Lemma 3.

► **Observation 4.** *Before the first execution of $FinishDeq(i + h)$, $FinishDeq(i)$ has been executed.*

Each Enqueue operation op has a unique timestamp composed of an integer obtained by reading the Max register $enqCounter$ during the execution of line 10, and the id of the process that executed the operation op .

► **Observation 5.** *For each p , the timestamps of the elements written in the sub-array $items[p]$ are monotonically increasing in accordance with their index in the array. In other terms, we have $items[p][i].ts < items[p][i + 1].ts$.*

At any given configuration, the sub-queue of process p is the sub-array of $items[p]$ in the range $items[p][head[p].MaxRead()], \dots, items[p][tail[p] - 1]$.

► **Lemma 6.** *Let enq_1 and enq_2 be two Enqueue operations such that enq_1 ends before enq_2 is invoked. Let (st_1, id_1) be the timestamp of enq_1 and (st_2, id_2) be the time stamp of enq_2 . We have $st_1 < st_2$.*

Proof. After the execution of line 14 of Algorithm 1 during enq_1 , any value returned by a $enqCounter.MaxRead$ is greater or equal to $st_1 + 1$. The claim follows from the fact that enq_2 executes line 10 of Algorithm 1 after enq_1 returned. ◀

We say that the i -th *Enqueue* operation by a process p matches the *Dequeue* operation with sequence number j , if $deqOps[j] = (i, p)$ at some point in the execution.

Meaning, if the *Dequeue* operation returns, it returns the element enqueued by the i -th *Enqueue* operation of process p (i.e. `items[p][i]`).

► **Lemma 7.** *An Enqueue operation has at most a single matching Dequeue operation.*

Proof. Let enq be the i -th *Enqueue* operation by a process p . Assume by contradiction that there are two *Dequeue* operations, deq_1 and deq_2 that match enq . Let j_1 and j_2 be their corresponding sequence numbers. Then, $deqOps[j_1] = deqOps[j_2] = (i, p)$. By Lemma 1 and without loss of generality, let $j_1 < j_2$. Because of the Observation 4, $FinishDeq(j_1)$ returned before $FinishDeq(j_2)$ is invoked. According to lines 22 to 23 of Algorithm 1, $UpdateTree(j_1)$ is executed before $FinishDeq(j_1 + 1)$. This means that the value $i + 1$ is written in the Max register $head[p]$ at line 34 before that a process read it during the $FinishDeq(j_1 + 1)$. And since $j_2 \geq j_1 + 1$, the claim follows. ◀

► **Lemma 8.** *Let enq denote the i -th Enqueue operation by a process p . Let $ts = (st, p)$ be the timestamp of enq . Let s be any node in the tree T in the path from the p -th leaf to the root of the tree. At any configuration C after enq ends and such that $deqOps[j] \neq (i, p)$ for each $j \geq 0$, we have that the timestamp stored at s is smaller than or equal to ts at C .*

Proof. After enq , we have that $tail[p] \geq i + 1$, because enq is the i -th *Enqueue* operation executed by p .

We first prove that after enq , $head[p]$ is smaller than or equal to i as long as $deqOps[l] \neq (i, p)$ for any $l \geq 0$.

The value of $head[p]$ is updated only during the execution of the function $UpdateTree$ (line 34 of Algorithm 2). In particular, the value of $head[p]$ is set to a value $j + 1$ where j is the value read from some $deqOps[num]$ at line 32. Also, the value of $deqOps[num]$ is updated only during the execution of the function $FinishDeq(num)$ with a value read from $head[p]$ (lines 29 and 30). We prove by induction on j that if the value written in $head[p]$ is j then, all values $0, \dots, j - 1$ have been previously written in $head[p]$ (in increasing order) and to some $deqOps[num]$. The base case is for $j = 1$. Consider the first $MaxWrite()$ that writes 1 to $head[p]$ and let q be the process applying this primitive. According to line 34, q has read the value $(0, p)$ from some $deqOps[num]$, which has been updated with a value read from $head[p]$. The claim follows.

Suppose this is true for a value j , we show that the claim holds for $j + 1$. Consider the first process, denoted q , that writes $j + 1$ into $head[p]$. q has read (j, p) from some $deqOps[num]$ at line 32. By inductive hypothesis, and by the linearizability of $head[p]$ all the values $0, \dots, j$ have been written in $head[p]$ and all the values $0, \dots, j - 1$ have been written in some $deqOps[num]$. The claim follows.

Hence, $head[p] \leq i$ as long as for any $l \geq 0$, we have $deqOps[l] \neq (i, p)$. This is because to write the value $i + 1$ (and then any greater value), a process has to read $deqOps[l] = (i, p)$ for some l .

Base case $k = 0$. s is the p -th leaf. Since enq completes, there is at least one instance of $Propagate(p)$ performed after that process p has written the value i in $tail[p]$. The value of $head[p]$ is smaller than or equal to i , so any instance of $Propagate(p)$ that changes the value of s before C , will write a timestamp read in $items[p][j]$ for some $j \geq i$. By Observation 5, the timestamp read is smaller than or equal to $ts = (st, p)$.

It remains to prove that after an instance of $Propagate(p)$ completes, denoted $prop$, a value smaller than or equal to i has been written in the leaf corresponding to p . An instance of $Propagate(p)$ performs two $Refresh(s)$. Each $Refresh(s)$ reads the state of s , then the $head[p]$ and the corresponding timestamp ts and then applies a CAS to s to modify its value with ts . Suppose that both $Refresh(s)$ fail (and in particular the second one), otherwise the claim is trivial. The second $Refresh(s)$ fails because another instance of $Propagate(p)$, denoted $prop'$ successfully applied a CAS on s . But $prop'$ has read $head[p]$ after $tail[p]$ is set to i . Meaning that it has read a value smaller than or equal to i and it writes in s the corresponding timestamp that is smaller than or equal to ts .

Induction case $k + 1 \leq \log n$. Suppose that the claim holds for $j \leq \log n$: the timestamp stored at s_j is smaller than or equal to ts where s_j is in the path from the p -th leaf to the root at a height of $j \leq k$. We prove that the claim holds for the parent of s_j , denoted s_{j+1} .

Any instance of $Propagate(p)$ updates the nodes in the path from the p -th leaf to the root, one by one, starting from the leaf and following the path to the root. Also, immediately after enq completes, there is at least one $Propagate(p)$ instance that passed through all the nodes in this path. Consider, the first $Propagate(p)$ that updated node s_{j+1} after s_j has been updated, denoted $prop$.

Observe that any process that executes the $Refresh$ function on node s_{j+1} writes the minimum timestamp it reads from the children of s_{j+1} . And that the second $Refresh(s_{j+1})$ fails only if another $Propagate(p)$ has modified the state of this node with a value smaller than or equal to the value at s_j read by $prop$. ◀

► **Lemma 9.** *Let enq be an Enqueue operation with the timestamp ts that enqueued $items[p][i]$. If (i, p) was written to $deqOps[j]$ by a process q , then the execution of line 25 of Algorithm 2 to read ts by q was executed after the invocation of enq .*

Proof. enq is the i -th enqueue operation by p . Let deq be the Dequeue operation executed by q that retrieves ts from the root of the tree (Line 25 of Algorithm 2) before writing (i, p) to $deqOps[j]$. enq must execute the line 13 of Algorithm 1 before ts can be propagated in the tree according to the code of function $Refresh$. The claim follows. ◀

► **Lemma 10.** *Let enq_1 and enq_2 be two Enqueue operations such that enq_1 ends before enq_2 is invoked. If enq_2 has a matching Dequeue operation deq_2 , then enq_1 also has a matching Dequeue operation deq_1 .*

Proof. By contradiction, we suppose that deq_2 exists and deq_1 does not. We denote ts_1 and ts_2 the timestamps associated with enq_1 and enq_2 respectively and num_2 the sequence number of deq_2 . From Lemma 6, $ts_1 < ts_2$ because enq_1 ends before enq_2 begins.

And since enq_1 does not have a matching Dequeue, there is no $j \geq 0$ such that $deqOps[j] = (i, p)$ where $items[i][p]$ is enqueued by enq_1 . Therefore, from Lemma 8, for any node s in the path in T from the p -th leaf to the root, the timestamp stored at s is smaller than or equal to ts_1 . In particular, for the root of the tree, the timestamp stored is smaller or equal to ts_1 . From Lemma 9, the step of line 25 of Algorithm 2 to read the root of the tree before writing $deqOps[num_2]$ is executed after the invocation of enq_2 which is after the invocation of enq_1 . Meaning that during this step, the timestamp at the root was smaller or equal to ts_1 contradicting the fact that $ts_1 < ts_2$. ◀

► **Lemma 11.** *Let enq_1 and enq_2 be two Enqueue operations such that enq_1 ends before enq_2 is invoked and let deq_1 and deq_2 be the matching Dequeue operations to enq_1 and enq_2 respectively. We have that deq_1 has a lower sequence number than deq_2 .*

Proof. We denote num_1 and num_2 the sequence numbers of deq_1 and deq_2 respectively, and ts_1 and ts_2 the timestamps of enq_1 and enq_2 respectively. By contradiction, we suppose that $num_1 > num_2$. Since enq_1 ends before enq_2 begins we have that $ts_1 < ts_2$ (Lemma 6).

And since $deqOps[i]$ are written in an increasing order of i according to Lemma 3, we have that $deqOps[num_2]$ is written before $deqOps[num_1]$. However, from Lemma 8, as long as $deqOps[num_1]$ has its initial value, then the timestamp stored at the root is smaller than or equal to ts_1 . At the execution of line 25 of Algorithm 2 to compute the final value of $deqOps[num_2]$, the root has a timestamp smaller or equal to ts_1 ; contradicting the fact that $ts_1 < ts_2$. ◀

► **Lemma 12.** *Let deq be a Dequeue operation and let enq be an Enqueue operation that ends before deq is complete. Let C be a configuration of E where enq does not have a matching Dequeue operation deq' or deq' is not complete at C . If deq is complete at C , then deq does not return ϵ .*

Proof. By contradiction, we suppose that deq returns ϵ . Let i denote the sequence number of deq and ts denote the timestamp of enq . Since deq returns ϵ , deq reads the value $(\epsilon, -1)$ in $deqOps[i]$ at line 24 of Algorithm 1. Therefore, during the execution of $FinishDeq(i)$, the process that writes $deqOps[i]$, reads $(\epsilon, -1)$ at the root of the tree (line 27 of Algorithm 2). However, By Lemma 8, the timestamp at the root of the tree after the end of enq is smaller than or equal to ts . Meaning that during the execution of line 25 of Algorithm 2 during the instance $FinishDeq(i)$ that writes $deqOps[i]$, the timestamp at the root of the tree was smaller than or equal to ts . We reach a contradiction because $(\epsilon, -1)$ is larger than any timestamp $(h, -) \forall h \in \mathbb{N}$. ◀

3.3.2 Linearizability

First, we construct a permutation L of some of the $Dequeue()$ and $Enqueue()$ operations invoked such that L contains all operations that have terminated. Then, we prove that L preserves the real order as well as the semantics of a queue.

3.3.2.1 Linearization definition

Let E denote a given execution of the wait-free queue implemented in Algorithm 1 and Algorithm 2. We classify every $Dequeue()$ operation deq that appears in E to exactly one of the following types :

1. deq does not execute line 18 of Algorithm 1 in E . Thus deq is not attributed a sequence number.
2. deq executes line 18 of Algorithm 1 in E , its sequence number is j and $deqOps[j]$ has the initial value (\perp, \perp) in E .
3. deq executes line 18 of Algorithm 1 in E , its sequence number is j and $deqOps[j] \neq (\perp, \perp)$ in E .

We remove from E , any $Dequeue()$ operation of type 1 and 2. We denote DEQ the set of $Dequeue()$ operations of type 3. Each operation in DEQ is associated with a unique sequence number $j \in \mathbb{N}_0$. We totally order all the operations in DEQ according to their sequence number. Also, let deq be any incomplete $Dequeue()$ operation in DEQ and let j be its sequence number. We complete deq by returning the value v if $deqOps[j] = (i, id)$ in E and $items[id][i] = (v, -)$. Otherwise, we complete deq by returning the empty queue value ϵ .

We remove every *Enqueue()* operation that does not execute line 13 of Algorithm 1 in E . We denote ENQ the set of *Enqueue()* operations that appear in E and that we do not remove. Every *Enqueue()* operation enq in ENQ is uniquely identified by a pair (i, id) meaning that enq is the i -th *Enqueue()* operation performed by the process id . We associate the *Dequeue()* operation in DEQ with sequence number i with the *Enqueue()* operation (j, id) such that $deqOps[i] = (j, id)$.

Let ENQ_d denote the *Enqueue()* operations in ENQ that have an associated *Dequeue()* operation in DEQ . We associate each *Enqueue()* operations in ENQ_d with the sequence number of the corresponding *Dequeue()*. Thus, *Enqueue()* operations in ENQ_d are totally ordered according to the given sequence number.

We construct the linearization L of the operations in E as follows:

1. First we insert the *Enqueue()* operations in ENQ_d one by one and according to their total order, denoted $enq_{i_1}, enq_{i_2} \dots$ in L . Notice that enq_{i_h} is the *Enqueue()* operation associated with the *Dequeue()* operation having the sequence number i_h . Assuming that $enq_{i_{h+1}}$ exists, we have $i_h < i_{h+1}$; and all the *Dequeue()* operations having a sequence number $i \in [i_h + 1, i_{h+1} - 1]$ return the value ϵ .
2. Then, we insert the *Dequeue()* operations one by one according to their the sequence number. For any sequence number k , If deq_k returns ϵ it is inserted immediately after deq_{k-1} if it exists, or at the beginning otherwise. In the case where deq_k does not return ϵ , it is linearized immediately after the furthest point in L following: (i) the previous deq_{k-1} , (ii) the matching *Enqueue* operation enq_{i_l} with $i_l = k$, and (iii) the last *Enqueue* operation that ends before the invocation of deq_k .
3. Let enq denote an *Enqueue* operation from the remaining *Enqueue()* operations with no matching *Dequeue* operations (i.e. $ENQ \setminus ENQ_d$). We insert enq after the last operation in ENQ_d and before the first *Dequeue()* operation that starts after enq ends (or at the end of L if such *Dequeue()* does not exist). If multiple operations from $ENQ \setminus ENQ_d$ are linearized at the same point, then they are ordered according to their real-time order.

For two operations op_1 and op_2 , we denote $op_1 <_L op_2$ when op_1 precedes op_2 in the linearization L .

3.3.2.2 Linearization and real-time order

We show that the linearization defined in the previous section respects the real-time execution order.

► **Lemma 13.** *Let op_1 and op_2 be two *Enqueue* operations in E such that op_1 ends before op_2 is invoked. op_1 precedes op_2 in L .*

Proof. First, consider the case where both operations do not have matching *Dequeue()* operations. From linearization rule 3, an *Enqueue* operation that does not have a matching *Dequeue* operation is linearized before the first *Dequeue* operation that starts after it ends or at the end of L if such *Dequeue* operation does not exist. If op_1 is linearized at the end of L , then op_2 is also linearized at the end of L after op_1 , because op_2 starts after op_1 ends and there is no *Dequeue* operation that starts after op_1 ends. We suppose that there exists a *Dequeue* operation deq_1 such that op_1 is linearized immediately before deq_1 . If op_2 is linearized at the end of L , the claim is trivial. So let deq_2 be a *Dequeue* operation such that op_2 is linearized immediately before deq_2 . We have $op_1 <_{ro} op_2 <_{ro} deq_2$. Meaning that $deq_2 = deq_1$ or $deq_1 <_L deq_2$, because both operations start after op_1 ends, and deq_1 is the first such operation in L . Therefore, $op_1 <_L op_2$ according to their real time execution order following linearization rule 3.

Next, if op_1 has a matching $Dequeue()$ operation but op_2 does not, we have that op_2 is linearized after the last linearized $Enqueue()$ operation that has a matching $Dequeue()$ operation. The case where op_1 does not have a matching $Dequeue()$ operation but op_2 does, is impossible according to Lemma 10. We suppose that both op_1 and op_2 have matching $Dequeue()$ operations, named respectively deq_1 and deq_2 . From Lemma 11, we have that deq_1 has a smaller sequence number than deq_2 . Therefore, from linearization rule 1, op_1 is linearized before op_2 . ◀

► **Lemma 14.** *Let deq be a Dequeue operation with the sequence number j and let enq be an Enqueue operation invoked after deq returns. If enq has a matching Dequeue operation deq' , then the sequence number of deq' is greater than j .*

Proof. We denote i the sequence number of deq' . By contradiction we suppose that $j > i$. We consider the configuration C where deq completes. According to Lemma 3, deq' also has been completed at C . Meaning that $deqOps[i] \neq (\perp, \perp)$ at C . However, from the hypothesis, enq has not started at C , as enq is not invoked until deq finishes. According to Lemma 9, deq' cannot match enq . The claim follows. ◀

► **Lemma 15.** *Let deq be a Dequeue operation with the sequence number j and let enq be an Enqueue operation invoked after deq returns. If enq has a matching Dequeue operation deq' , then any Dequeue operation with a sequence number $l < j$ is linearized before enq .*

Proof. By contradiction, we suppose that there exists $Dequeue$ operations with sequence numbers strictly smaller than j that are linearized after enq , and let deq_l be the first of these operations in L . Thus, if deq_{l-1} exists, we have that $deq_{l-1} <_L enq$.

If deq_l returns ϵ , from linearization rule 2, deq_l is linearized immediately after deq_{l-1} if it exists, or at the beginning of L . Therefore, $deq_l <_L enq$. There is a contradiction.

Otherwise, deq_l has a matching $Enqueue$ operation denoted enq_l . We denote i the sequence number of deq' . From Lemma 14, we have that $j < i$. Therefore, $l < j < i$. Thus, $enq_l <_L enq$ from linearization rule 1. Furthermore, we have $deq_{l-1} <_L enq$ (if it exists). Therefore, since $enq_l <_L enq$ and $deq_{l-1} <_L enq$, according to linearization rule 2, $enq <_L deq_l$ because $enq <_{ro} deq_l$ (rule 2.3 of linearization). Consequently, $deq_j <_{ro} enq <_{ro} deq_l$. Contradicting the fact that $l < j$ (Lemma 1). ◀

► **Theorem 16.** *Let op_1 and op_2 be two operations in E such that op_1 ends before op_2 is invoked. Then, op_1 precedes op_2 in L .*

Proof. Four cases have to be studied according to the type of operations.

1. op_1 and op_2 are two $Dequeue()$ operations. Since op_1 ends before op_2 begins, the sequence number i_1 of op_1 is strictly smaller than the sequence number i_2 of op_2 (Lemma 1). From linearization rule 2, we have op_1 is before op_2 in L .
2. The case where op_1 and op_2 are $Enqueue()$ operations is proved by Lemma 13.
3. op_1 is an $Enqueue$ operation and op_2 is a $Dequeue()$ operation. First, consider the case that op_2 does not return ϵ . If $op_1 \in ENQ_d$, then from linearization rule 2, op_2 is linearized after op_1 because op_2 is inserted after the last $Enqueue$ operation that ends before op_2 starts. Otherwise, If $op_1 \notin ENQ_d$, from linearization rule 3, it is linearized before the first $Dequeue$ operation that starts after op_1 ends. Thus op_1 is linearized before op_2 .
Next, consider the case where op_2 returns ϵ , and let i denote its sequence number. By Observation 2 and Lemma 12, op_1 has a matching $Dequeue$ operation deq , and deq is complete before op_2 is complete.

Let j is the sequence number of deq . Since deq is complete before op_2 is complete, by Lemma 3, we have that $j < i$. Therefore, from linearization rule 2, deq is linearized before op_2 . Thus, from linearization rule 1, $op_1 <_L deq <_L op_2$. The claim follows.

4. Finally, we suppose that op_1 is a *Dequeue* operation and that op_2 is an *Enqueue* operation. If op_2 does not have a matching *Dequeue* operation, from linearization rule 3, it is linearized before the first *Dequeue* operation that starts after op_2 ends or at the end of L if such operation does not exist. Thus, op_2 is linearized after op_1 because op_1 ends before op_2 starts.

So consider that op_2 has a matching *Dequeue* operation deq and let i be its sequence number and j be the sequence number of op_1 .

If op_1 returns ϵ , from the linearization rule 2, we have $op_1 = deq_j$ is linearized immediately after deq_{j-1} (or beginning of L if it does not exist). And from Lemma 15, for each $l < j$, we have that deq_l is linearized before op_2 . In particular, we have that deq_{j-1} is linearized before op_2 . Therefore, op_1 is linearized before op_2 .

Otherwise, consider enq_j the matching operation of op_1 . From linearization rule 2, op_1 is linearized after (i) deq_{j-1} , (ii) enq_j and after (iii) the last *Enqueue* enq' that ends before op_1 starts. We show that op_2 is linearized after all these three operations. From Lemma 15, we have that deq_{j-1} is linearized before op_2 (i). From Lemma 14, we have that $j < i$ meaning that enq_j is linearized before op_2 according to the total order of the sequence numbers of their matching *Dequeue* operations (ii). And since op_1 ends before op_2 starts, $enq' <_{ro} op_2$. Therefore, $enq' <_L op_2$ because we have shown that the linearization of the *Enqueue* operations respects the real time execution order (Lemma 13) (iii). The claim follows. ◀

3.3.2.3 Linearization and the Queue Sequential Specification

► **Lemma 17.** *Let deq be a *Dequeue* operation that returns $v \neq \epsilon$. There exists an *Enqueue*(v) denoted enq that such that enq is linearized before deq and there is no *Dequeue* operation $deq' \neq deq$ that also returns v .*

Proof. First, we prove that enq exists. Since deq returns $v \neq \epsilon$, it has read a value (j, p) in $deqOps[i]$ where i is the sequence number of deq (line 24 of Algorithm 1). Meaning that $items[p][j] = v$ and the *Enqueue* operation that enqueued v denoted enq , is the j -th instance of *Enqueue* by process p . By linearization rule 2, deq is linearized after enq . And we have shown in Lemma 7 that each *Enqueue* operation has at most a single matching *Dequeue* operation. The claim follows. ◀

► **Lemma 18.** *Let enq_1 and enq_2 be two *Enqueue* operations such that $enq_1 <_L enq_2$. If enq_2 has a matching *Dequeue* deq_2 , then enq_1 has a matching *Dequeue* deq_1 and $deq_1 <_L deq_2$.*

Proof. By contradiction, we suppose that enq_1 does not have a matching *Dequeue* operation. From linearization rule 3, enq_1 is linearized after all *Enqueue* operations in ENQ_d . Especially, enq_1 is linearized after enq_2 . There is a contradiction. And from linearization rule 1, enq_1 and enq_2 are linearized according to the total order of the sequence numbers of their matching *Dequeue* operations. The claim follows. ◀

From the two previous Lemmas 17–18, we have the following theorem.

► **Theorem 19.** *Let deq be a *Dequeue* operation in L . If deq does not return ϵ , then it returns the element enqueued by the first *Enqueue* operation in L that does not have a matching *Dequeue* operation linearized before deq .*

► **Lemma 20.** *Let deq_ϵ be a Dequeue operation that returns ϵ . And let enq be an Enqueue operation linearized before deq_ϵ . We have that enq has a matching Dequeue operation deq that is also linearized before deq_ϵ .*

Proof. First, we show that enq has a matching Dequeue operation deq . By contradiction, we suppose that enq is in $ENQ \setminus ENQ_d$. From linearization rule 3, enq is inserted before the first Dequeue operation deq' that starts after enq ends or at the end of L if deq' does not exist. The case where enq is linearized at the end of L is trivial because it contradicts the fact that enq is linearized before deq_ϵ . So deq' exists. By lemma 12 deq' does not return ϵ . Since $enq <_L deq_\epsilon$, we have $deq' <_L deq_\epsilon$. Hence, deq_ϵ has a greater sequence number than deq' from linearization rule 2. Thus, deq_ϵ is complete after deq' is complete (Lemma 3). We conclude by lemma 12, that deq_ϵ does not return ϵ . There is a contradiction. Thus, enq has a matching Dequeue operation denoted deq .

In the following, we establish that deq is linearized before deq_ϵ . Let i denote the sequence number of deq_ϵ and let j be the sequence number of deq . By contradiction, we assume that $i < j$ (i.e. deq is linearized after deq_ϵ). Let deq_k be the first Dequeue operation linearized after enq with k its sequence number. Such an operation exists as $enq <_L deq_\epsilon$. We have $k \leq i$, according to the linearization rule 2. Assume that deq_k returns ϵ . If $k = 0$ then no operation is linearized before deq_k ; in this case, there is a contradiction. Otherwise ($k \geq 1$), there is no Enqueue operation linearized after deq_{k-1} and before deq_k because deq_k is linearized immediately after deq_{k-1} (linearization rule 2). This contradicts the fact that deq_k is the first Dequeue operation linearized after enq . Hence deq_k does not return ϵ . We conclude that $k < i$. Therefore, deq_k is complete before deq_ϵ is complete (Lemma 3). deq_k does not match enq as we assume that deq is linearized after deq_ϵ . From linearization rule 2, deq_k can only be linearized after enq because enq terminates before the invocation of deq_k . Thus, by Lemma 12, deq_ϵ cannot return ϵ if $j > i$. There is a contradiction. ◀

3.3.3 Step Complexity

We show that the worst-case step complexity of an Enqueue and Dequeue operation is $O(\log n)$ and $O(k \log n)$, respectively. To do so, we establish the following Lemma but omit the detailed proof because of space limitations. The main intuition is that while propagating the timestamp, the process has to read a constant number of nodes per level going from a leaf to the root. Since there are n leaves, the height of the tree is in $O(\log n)$.

► **Lemma 21.** *A process executes $O(\log n)$ steps during a call to the function $Propagate(id)$.*

During the execution of an Enqueue operation there are no loops or function calls aside from a call to the function $Propagate(id)$. And during a Dequeue operation, a process executes at most k instances of $Propagate(id)$. The following corollary ensues.

► **Corollary 22.** *A process executes $O(\log n)$ steps during the execution of an Enqueue operation and $O(k \log n)$ steps during the execution of a Dequeue operation.*

4 Set Linearizable Wait-free Queue Algorithm with multiplicity

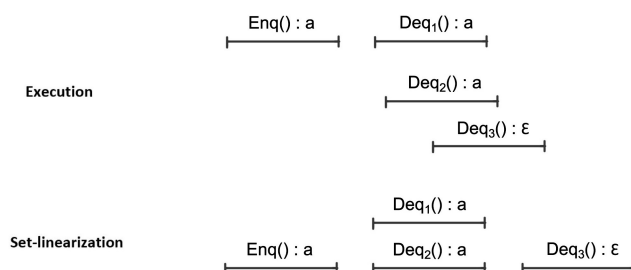
In this section, we propose an implementation of the relaxed queue with multiplicity where the operations have a step complexity of $O(\log n)$. For the relaxed queue with multiplicity, concurrent Dequeue operations are allowed to return the same element from the queue (Figure 2 illustrates such an execution).

■ **Algorithm 3** Relaxed-Queue: implementation of the wait-free queue with multiplicity (Dequeue pseudo-code for process p).

```

1 Function Dequeue()
2    $num \leftarrow \text{deqCounter.MaxRead}()$ 
3   if  $\text{deqOps}[num].\text{Read}() \neq (\perp, \perp)$  then
4      $\text{deqCounter.MaxWrite}(num + 1)$ 
5      $num \leftarrow num + 1$ 
6   if  $num \geq 1$  then
7      $\text{UpdateTree}(num - 1)$ 
8      $\text{FinishDeq}(num)$ 
9      $(h, id) \leftarrow \text{deqOps}[num].\text{Read}()$ 
10    if  $id = \perp$  then
11      return  $\epsilon$ 
12    else
13       $(ret, -) \leftarrow \text{items}[id][h]$ 
14      return  $ret$ 

```



■ **Figure 2** Example of a set-linearizable execution of the relaxed queue with multiplicity.

Only the algorithm of the *Dequeue* operation is different from the Algorithm in Section 3. In the implementation of the relaxed queue, we do not require the unicity of the sequence numbers of the *Dequeue* operations. Therefore, we use a max register object for *deqCounter* instead of the previously used *Fetch&Inc*. Multiple concurrent *Dequeue* operations retrieve the same sequence number num from *deqCounter* as long as $\text{deqOps}[num]$ remains unchanged. A *Dequeue* operation takes the sequence number $num + 1$ only after the *Dequeue* operations with the sequence number num are completed (i.e. $\text{deqOps}[num] \neq (\perp, \perp)$). Thus, we relinquish the need for a helping mechanism for slow *Dequeue* operations since an operation with the same sequence number will need to finish and write to *deqOps* before the next sequence number is assigned.

If the value of *deqCounter* changes between the step a *Dequeue* operation retrieves the value num and the step it reads $\text{deqOps}[num]$, the operation writes $num + 1$ to *deqCounter* and assigns it as its sequence number. Similarly to Algorithm 1, the operation then executes the necessary steps to write $\text{deqOps}[seq]$ where $seq \in \{num, num + 1\}$ is the sequence number of the operation. Meaning that the process executes $\text{UpdateTree}(seq - 1)$ if the *Dequeue* operation with the sequence number $seq - 1$ exists, to ensure that the root of the tree has an accurate value. Then, the process executes $\text{FinishDeq}(seq)$, after which $\text{deqOps}[seq]$ is set to a value different than its initial value. If $\text{DeqOps}[seq] = (i, p)$ the *Dequeue* operation returns $\text{items}[p][i].\text{val}$, otherwise it returns ϵ . Several *Dequeue* operations may have the

same sequence number, and thus return the same value. The design of the algorithm ensures that two *Dequeue* operations can have the same sequence number only if they are concurrent. The full proof of correctness of the relaxed queue implementation is omitted because of space limitations but uses similar techniques as the previous sections.

5 Discussion

We have presented a wait-free implementation of a k -multiple dequeuer n -multiple enqueuer FIFO queue. The worst case step complexity of the *Enqueue* operation is in $O(\log n)$ and the *Dequeue* operation is in $O(k \log n)$. Meaning, that as long as the number k of dequeuer processes is constant, our implementation has logarithmic step complexity, which improves on the previous upper bound of $O(\sqrt{n})$. While we focused on theoretical evaluations of step complexity, it could also be of interest to compare the algorithm empirically to other FIFO implementations to gauge its applicative relevance.

Then, to the best of our knowledge, we presented the first relaxed FIFO queue with logarithmic step complexity where every process can perform both *Enqueue*(v) and *Dequeue*() operations. It remains an open question whether it is possible to implement an exact wait-free linearizable FIFO queue with worst-case logarithmic step complexity without restriction on the number of enqueuers and dequeuers or to implement a relaxed FIFO queue in constant or near-constant step complexity.

References

- 1 Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. Relaxed queues and stacks from read/write operations. In *24th International Conference on Principles of Distributed Systems, OPODIS 2020*, pages 13:1–13:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.OPODIS.2020.13.
- 2 Armando Castañeda and Miguel Piña. Fully read/write fence-free work-stealing with multiplicity, 2020. doi:10.48550/arXiv.2008.04424.
- 3 Matei David. A single-enqueuer wait-free queue implementation. In *Proceedings of 18th International Conference Distributed Computing, DISC 2004*, pages 132–143, Berlin, Heidelberg, 2004. Springer-Verlag. doi:10.1007/978-3-540-30186-8_10.
- 4 David Eisenstat. Two-enqueuer queue in common2, 2008.
- 5 Panagiota Fatourou and Nikolaos D. Kallimanis. Highly-efficient wait-free synchronization. *Theor. Comp. Sys.*, 55(3):475–520, October 2014. doi:10.1007/s00224-013-9491-y.
- 6 Thomas A. Henzinger, Christoph M. Kirsch, Hannes Payer, Ali Sezgin, and Ana Sokolova. Quantitative relaxation of concurrent data structures. *SIGPLAN Not.*, 48(1):317–328, January 2013. doi:10.1145/2480359.2429109.
- 7 Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991. doi:10.1145/114005.102808.
- 8 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.
- 9 Prasad Jayanti and Srdjan Petrovic. Logarithmic-time single deleter, multiple inserter wait-free queues and stacks. In *Proceedings of the 25th International Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS '05*, pages 408–419, Berlin, Heidelberg, 2005. Springer-Verlag. doi:10.1007/11590156_33.
- 10 Pankaj Khanchandani and Roger Wattenhofer. On the importance of synchronization primitives with low consensus numbers. In *Proceedings of the 19th International Conference on Distributed Computing and Networking, ICDCN '18*, pages 18:1–18:10, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3154273.3154306.

- 11 Christoph M. Kirsch, Michael Lippautz, and Hannes Payer. Fast and scalable, lock-free k-fifo queues. In *Proceedings of 12th International Conference Parallel Computing Technologies*, PaCT'13, pages 208–223, Berlin, Heidelberg, 2013. Springer-Verlag. doi:10.1007/978-3-642-39958-9_18.
- 12 Alex Kogan and Erez Petrank. Wait-free queues with multiple enqueueers and dequeuers. *SIGPLAN Not.*, 46(8):223–234, February 2011. doi:10.1145/2038037.1941585.
- 13 Zongpeng Li. Non-blocking implementations of queues in asynchronous distributed shared-memory systems. Master's thesis, Univ. of Toronto, January 2001.
- 14 Adam Morrison and Yehuda Afek. Fast concurrent queues for x86 processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '13, pages 103–112, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2442516.2442527.
- 15 Gil Neiger. Set-linearizability. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '94, page 396, New York, NY, USA, 1994. Association for Computing Machinery. doi:10.1145/197917.198176.
- 16 Chaoran Yang and John Mellor-Crummey. A wait-free queue as fast as fetch-and-add. *SIGPLAN Notices*, 51(8):1–13, February 2016. doi:10.1145/3016078.2851168.

EEMARQ: Efficient Lock-Free Range Queries with Memory Reclamation

Gali Sheffi ✉

Department of Computer Science, Technion, Haifa, Israel

Pedro Ramalhete ✉

Cisco Systems, Zürich, Switzerland

Erez Petrank ✉

Department of Computer Science, Technion, Haifa, Israel

Abstract

Multi-Version Concurrency Control (MVCC) is a common mechanism for achieving linearizable range queries in database systems and concurrent data-structures. The core idea is to keep previous versions of nodes to serve range queries, while still providing atomic reads and updates. Existing concurrent data-structure implementations, that support linearizable range queries, are either slow, use locks, or rely on blocking reclamation schemes. We present EEMARQ, the first scheme that uses MVCC with lock-free memory reclamation to obtain a fully lock-free data-structure supporting linearizable inserts, deletes, contains, and range queries. Evaluation shows that EEMARQ outperforms existing solutions across most workloads, with lower space overhead and while providing full lock freedom.

2012 ACM Subject Classification Software and its engineering → Memory management; Theory of computation → Concurrency

Keywords and phrases safe memory reclamation, lock-freedom, snapshot, concurrency, range query

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2022.5

Related Version *Full Version*: <https://arxiv.org/abs/2210.17086>

Supplementary Material *Software (Source Code)*: <https://github.com/galisheffi/EEMARQ>
archived at `swh:1:dir:4e0ee7f41e81d100392b41c7af384a7896ee9760`

Funding This work was supported by the Israel Science Foundation Grant No. 1102/21.

1 Introduction

Online Analytical Processing (OLAP) transactions are typically long and may read data from a large subset of the records in a database [51, 56]. As such, analytical workloads pose a significant challenge in the design and implementation of efficient concurrency controls for database management systems (DBMS). Two-Phase Locking (2PL) [64] is sometimes used, but locking each record before it is read implies a high synchronization cost and, moreover, the inability to modify these records over long periods. Another way to deal with OLAP queries is to use Optimistic Concurrency Controls [25], where the records are not locked, but they need to be validated at commit time to guarantee serializability [46]. If during the time that the analytical transaction executes, there is any modification to one of these records, the analytical query will have to abort and restart. Aborting can prevent long read-only queries from ever completing.

DBMS designers typically address these obstacles using Multi-Version Concurrency Control (MVCC). MVCC's core idea is to keep previous versions of a record, allowing transactions to read data from a fixed point in time. For managing the older versions, each record is associated with its list of older records. Each version record contains a copy of an older version, and its respective time stamp, indicating its commit time. Each update of the record's values adds a new version record to the top of the version list, and every read of an older version is done by traversing the version list, until the relevant timestamp is reached.



© Gali Sheffi, Pedro Ramalhete, and Erez Petrank;
licensed under Creative Commons License CC-BY 4.0

26th International Conference on Principles of Distributed Systems (OPODIS 2022).

Editors: Eshcar Hillel, Roberto Palmieri, and Etienne Rivière; Article No. 5; pp. 5:1–5:22

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Keeping the version lists relatively short is fundamental for high performance and low memory footprint [11, 19, 33, 34, 36]. However, the version lists should be carefully pruned, as a missing version record can be harmful to an ongoing range query. In the Database setting, the common approach is to garbage collect old versions once the version list lengths exceed a certain threshold [11, 36, 52]. During garbage collection, the entire database is scanned for record instances with versions that are not required for future scanners. However, garbage collecting old versions with update-intensive workloads considerably slows down the entire system. An alternative approach is to use transactions and associate each scan operation with an undo-log [6, 41]. But this requires allocating memory for storing the undo logs and a safe memory reclamation scheme to recycle log entries.

In contrast to the DBMS approach, many concurrent in-memory data-structures implementations do not provide an MVCC mechanism, and simply give up on range queries. Many map-based data-structures provide linearizable [29] insertions, deletions and membership queries of single items. Most data-structures that do provide range queries are blocking. They either use blocking MVCC mechanisms [3, 12, 43], or rely on the garbage collector of managed programming languages [7, 22, 50, 67] which is blocking. It may seem like lock-free data-structures can simply employ a lock-free reclamation scheme with an MVCC mechanism to obtain full lock-freedom, but interestingly this poses a whole new challenge.

Safe manual reclamation (SMR) [40, 55, 58, 61, 66] algorithms rely on *retire()* invocations by the program, announcing that a certain object has been unlinked from a data-structure. The task of the SMR mechanism is to decide which retired objects can be safely reclaimed, making their memory space available for re-allocation. Most SMR techniques [40, 55, 61, 66] heavily rely on the fact that retired objects are no longer reachable for threads that concurrently traverse the data structure. Typically, objects are retired when deleted from the data-structure. However, when using version lists, if we retire objects when they are deleted from the current version of the data-structure, they would still be reachable by range queries via the version list links. Therefore, it is not safe to recycle old objects with existing memory reclamation schemes. Epoch-based reclamation (EBR) [15, 24] is an exception to the rule because it only requires that an operation does not access nodes that were retired before the operation started. Namely, an existing range query prohibits reclamation of any deleted node, and subsequent range queries do not access these nodes, so they can be deleted safely. Therefore, EBR can be used without making any MVCC-specific adjustments, and indeed EBR is used in many in-memory solutions [3, 43, 65]. However, EBR is not robust [58, 61, 66]. I.e., a slow executing thread may prevent the reclamation of an unbounded number of retired objects, which may affect performance, and theoretically block all new allocations.

One lock-free memory reclamation scheme that can be adopted to provide lock-free support for MVCC is the VBR optimistic memory reclamation scheme [58, 59]. VBR allows a program to access reclaimed space, but it raises a warning when accessed data has been re-allocated. This allows the program to retry the access with refreshed data. The main advantages of VBR are that it is lock-free, it is fast, and it has very low memory footprint. Any retired object can be immediately reclaimed safely. The main drawback¹ is that reclaimed memory cannot be returned to the operating system, but must be kept for subsequent node allocations, as program threads may still access reclaimed nodes. Similarly to other schemes, the correctness of VBR depends on the inability of operations to visit previously retired

¹ VBR also necessitates type-preservation. However, this does not constitute a problem in our setting, as all allocated memory objects are of the same type. For more details, see Section 3.

objects. In data structures that do not use multi versions, the deleting thread adequately retires a node after disconnecting it from the data structure. But in the MVCC setting, disconnected nodes remain connected via the version list, foiling correctness of VBR.

In this paper we modify VBR to work correctly in the MVCC setting. It turns out that for the specific case of old nodes in the version list, correctness can be obtained. VBR keeps a slow-ticking epoch clock and it maintains a birth epoch field for each node. Interestingly, this birth epoch can be used to tell whether a retired node in the version list has been re-allocated. As we show in this paper, an invariant of nodes in the version list is that they have non-increasing birth-epoch numbers. Moreover, if one of the nodes in the version list is re-allocated, then this node must foil the invariant. Therefore, when a range query traverses a version list to locate the version to use for its traversal, it can easily detect a node that has been re-allocated and whose data is irrelevant. When a range query detects such re-allocation, it restarts. As shown in the evaluation, restarting happens infrequently with VBR and the obtained performance is the best among existing schemes in the literature. Using the modified VBR in the MVCC setting, a thread can delete a node and retire it after disconnecting it from the data structure (and while it is still reachable via version list pointers).

Two recent papers [43, 65] presented efficient MVCC-based key-value stores. The main new idea is to track and keep a version list of modified fields only and not of entire nodes. For many data-structures, all fields are immutable except for one or two pointer fields. For such data-structures, it is enough to keep a version list of pointers only. The first paper proposed a lock-free mechanism, based on *versioned CAS objects* (vCAS) [65], and the second proposed a blocking mechanism, based on *Bundle objects* (Bundles) [43]. While copying one field instead of the entire node reduces the space overhead, the resulting indirection is harmful for performance: to dereference a pointer during a traversal, one must first move to the top node of the version list, and then use its pointer to continue with the traversal. As we show in Section 4, this indirection suffers from high overheads in read-intensive workloads. Bundles ameliorate this overhead by caching the most recent pointer in the node to allow quick access for traversals that do not use older versions. However, range queries still need to dereference twice as many references during a traversal. The vCAS approach presents a more complicated optimization that completely eliminates indirection (which we further discuss in Section 3). However, its applicability depends on assumptions on the original data-structure that many data structures do not satisfy. Therefore, it is unclear how it can be integrated into general state-of-the-art concurrent data-structures. In terms of memory reclamation, both schemes use the EBR technique (that may block in theory, due to allocation heap exhaustion). This means that even vCAS does not provide a lock-free range query mechanism. A subsequent paper on MVCC-specific garbage collectors [8], provides a robust memory reclamation method for collecting the version records, but it does not deal with the actual data-structure nodes.

In this paper we present EEMARQ (End-to-End lock-free MAP with Range Queries), a design for a lock-free in-memory map with MVCC and a robust lock-free memory management scheme. EEMARQ provides linearizable and high-performance inserts, deletes, searches, and range queries.

The design starts by applying vCAS to a linked-list. The linked-list is simple, and so it allows applying vCAS easily. However, the linked-list does not satisfy the optimization assumptions required by [65], and so a (non-trivial) extension is required to fit the linked-list. In the unoptimized variant, there is a version node for each of the updates (insert or delete) and this node is used to route the scans in the adequate timestamp. But this is costly, because

traversals end up accessing twice the number of the original nodes. A natural extension is to associate the list nodes with the data that is originally kept in the version nodes. We augment the linked-list construction to allow the optimization of [65] for it. They use a clever mapping from the original version nodes to existing list nodes, that allows moving the data from version nodes to list nodes, and then elide the version nodes. Now traversals need no extra memory accesses to version nodes. This method is explained in Section 3.2.

Second, we extend VBR by adding support for reachable retired nodes on the version list. The extended VBR allows keeping retired reachable version nodes in the data structure (which the original VBR forbids) while maintaining high performance, lock-freedom, and robustness.

Finally, we deal with the inefficiency of a linked-list by adding a fast indexing to the linked-list nodes. A fast index can be obtained from a binary search tree or a skip list. But the advantage we get from separating the linked-list from the indexing mechanism is that we do not have to maintain versions for the index (e.g., for the binary search tree or the skip list), but only for the underlying linked-list. This separation between the design of the versioned linked-list and the non-versioned index, simplifies each of the sub-designs, and also obtains high performance, because operations on the index do not need to maintain versions. Previous work uses this separation idea between the lower level of the skip list or leaves of the tree from the rest for various goals (e.g., [43, 69]).

The combination of all these three ideas, i.e., optimized versioned linked-list, extended VBR, and independent indexing, yields a highly performant data structure design with range queries. Evaluation shows that EEMARQ outperforms both vCAS and Bundles (while providing full lock-freedom).

2 Related Work

Existing work on linearizable range queries in the shared-memory setting includes many solutions which are not based on MVCC techniques. Some data-structure interfaces originally include a tailor-made range query operation. E.g., there are trees [12, 13, 22, 67], hash tries [54], queues [44, 45, 53], skip lists [5] and graphs [30] with a built-in range query mechanism. Other and more general solutions execute range queries by explicitly taking a snapshot of the whole data-structure, followed by collecting the set of keys in the given range [1, 4]. The Snapcollector [50] forces the cooperation of all executing threads while a certain thread is scanning the data-structure. Despite being lock-free and general, the Snapcollector’s memory and performance overheads are high. The Snapcollector was enhanced to support range queries that do not force a snapshot of the whole data-structure [16]. However, this solution still suffers from major time and space overheads.

Another way to implement range queries is to use transactional memory [23, 31, 48, 49]. Transactions can either be implemented in software or in hardware, allowing range queries to take effect atomically. Although transactions may seem as ideal candidates for long queries, software transactions incur high performance overheads and hardware transactions frequently abort when accessing big memory chunks. Read-log-update (RLU) [38] borrows software transaction techniques and extends read-copy-update (RCU) [39] to support multiple updates. RLU yields relatively simple implementations, but its integration involves re-designing the entire data-structure. In addition, similarly to RCU, it suffers from high overheads in write-extensive workloads.

Arbel-Raviv and Brown exploited the EBR manual reclamation to support range queries [3]. Their technique uses EBR’s global epoch clock for associating data-structure items with their insertion and deletion timestamps. EEMARQ uses a similar technique for associating

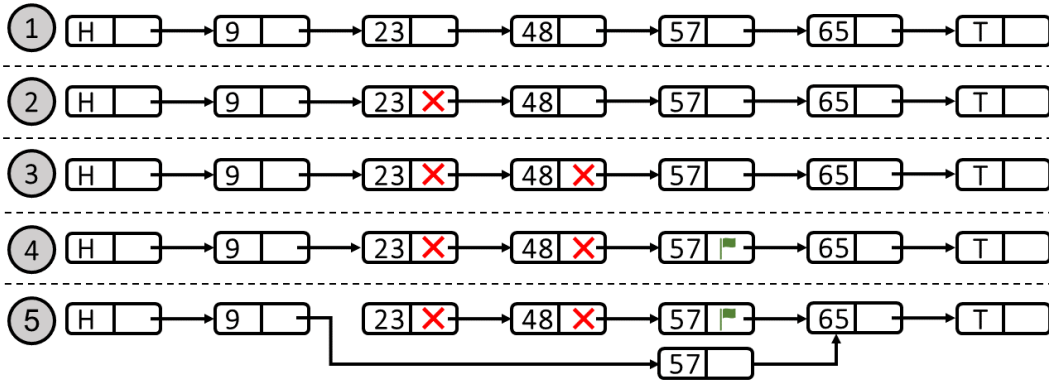
data-structure modifications with respective timestamps. They also took advantage of EBR's retire-lists, in order to locate nodes that were deleted during a range query scan. However, while their solution indeed avoids extensive scanning helping when deleting an item from the data-structure (as imposed by the Snapcollector [50]), it may still impose significant performance overheads (as shown in Section 4). EEMARQ minimizes these overheads by keeping a reachable path to deleted nodes (for more details, see Section 3.2).

Multi-Version Concurrency Control. MVCC easily provides isolation between concurrent reads and updates. I.e., range queries can work on a consistent view of the data, while not interfering with update operations. This powerful technique is widely used in commercial systems [19, 21], as well as in research-oriented DBMS [32, 52], in-memory shared environments [43, 65] and transactional memory [23, 31, 35, 48, 49]. MVCC has been investigated in the DBMS setting for the last four decades, both from a theoretical [9, 10, 34, 47, 68] and a practical [19, 32, 36, 41, 52] point of view. A lot of effort has been put in designing MVCC implementations and addressing the unwanted side-effects of long version lists. This issue is crucial both for accelerating version list scans and for reducing the contention between updates and garbage collection. Accelerating DBMS version list scans (independently of the constant need to prune them) has been investigated in [34]. Most DBMS-related work that focuses on this issue, tries to minimize the problem by eagerly collecting unnecessary versions [2, 11, 21, 33, 37, 41, 52].

Safe Memory Reclamation. Most existing in-memory environments that enable linearizable range queries, avoid managing their memory, by relying on automatic (and blocking) garbage collection of old versions [7, 22, 50, 67]. Solutions that do manually manage their allocated memory [3, 43, 65], use EBR for safe reclamation. In EBR, a shared epoch counter is incremented periodically, and upon each operation invocation, the threads announce their observed epoch. The epoch clock can be advanced when no executing thread has an announcement with a previous epoch. During reclamation, only nodes that had been retired at least two epochs ago are reclaimed. EBR is safe for MVCC because a running scan prevents the advance of the epoch clock, and also the reclamation of any node in the data-structure that was not deleted before the scan.

The MVCC-oriented garbage collector from [8] incorporates reference counting (RC) [17, 18, 27], in order to maintain lock-freedom while safely reclaiming old versions. Any object can be immediately reclaimed once its reference count reaches zero, without the need in invoking explicit *retire()* calls. While RC simplifies reclamation, it incurs high performance overheads and does not guarantee a tight bound on unreclaimed garbage (i.e., it is not robust).

Other reclamation schemes were also considered when designing EEMARQ. NBR [61] was one of the strongest candidates, as it is fast and lock-free (under some hardware assumptions). However, it is not clear whether NBR can be integrated into a skip list implementation (which serves as one of our fast indexes). Pointer-based reclamation methods (e.g., Hazard Pointers [40]) allow threads to protect specific objects (i.e., temporarily prevent their reclamation), by announcing their future access to these objects, or publishing an announcement indicating the protection of a bigger set of objects (e.g., Hazard Eras [55], Interval-Based Reclamation [66], Margin Pointers [62]). Although these schemes are robust (as opposed to EBR), it is unclear how they can be used in MVCC environments. They require that reaching a reclaimed node from a protected one would be impossible (even if the protected node is already retired). I.e., they require an explicit unlinking of old versions before retiring them. Besides the obvious performance overheads, it may affect robustness



■ **Figure 1** Removing nodes 23 and 48 from the linked-list. At stages 1–4, the list logically contains 5 nodes and at stage 5, it logically contains 3 nodes. The logical deletions of nodes 23 and 48 are executed via marking them (stages 2 and 3, respectively), flagging node 57 (stage 4) and inserting a new 57 representative instead of the three of them (stage 5). Nodes 23, 48, and the flagged 57 are then retired.

(as very old versions would not be reclaimed). The garbage collector from [8] uses Hazard Eras for unlinking old versions (to be eventually collected using an RC-based strategy), but it has not been evaluated in practice.

3 The Algorithm

In this Section we present EEMARQ’s design. We start by introducing a new lock-free linearizable linked-list implementation in Section 3.1. The list implementation is based on Harris’s lock-free linked-list [26], and includes the standard *insert()*, *remove()* and *contains()* operations. In Section 3.2 we explain how to add a linearizable and efficient range query operation. We describe the integration of the designated robust SMR algorithm in Section 3.3, and explain how to improve performance by adding an external index in Section 3.4. A full linearizability and lock-freedom proof for our implementation (including the range queries mechanism and the SMR and fast indexing integration) appears in the full version of this paper [60].

As discussed in [65], node-associated version lists introduce an extra level of indirection per node access. Methods that use designated version objects for recording updates, suffer from high overheads, especially in read-intensive workloads. For avoiding this level of indirection, we introduce a new variant of Harris’s linked-list. In our new variant, there is no need to store any update-related data in designated version records, since it can be stored directly inside nodes, in a well-defined manner. Associating each node with a single data-structure update (i.e., an insertion or a removal) is challenging. Typically, an insert operation includes a single update, physically inserting a node into the list. A remove operation involves a marking of the target node’s *next* pointer (serving as its logical deletion, and the operation’s linearization point in many existing implementations) and a following physical removal from the list. In other words, each node may be associated with multiple list updates. Since the target node’s physical deletion is not the linearization point of any operation, there is no need to record this update. However, each node may still be associated with either one or two updates throughout an execution (i.e., its logical insertion and deletion).

In our linked-list implementation, some new node is inserted into the list during every physical update of the list (either a physical insertion or deletion), which obviously yields the desirable association between nodes and data-structure updates (each node is associated with the update that involved its insertion into the list). The node inserted during a physical insertion is simply the inserted node, logically inserted into the list. The node inserted during a physical removal is a designated new node that replaces the deleted node's successor, and is physically inserted together with the physical removal of the deleted node ².

Figure 1 shows an example for inserting a new node during deletion. This list illustration shows the list layout throughout the deletion procedure of two nodes. At the first stage, the list contains five nodes, ordered by their keys (together with the *head* and *tail* sentinels). Then, some thread marks node 23 for deletion. At this point, before physically removing it, some other thread marks node 48 for deletion. I.e., both nodes must be physically unlinked together, as successive marked nodes. In order to remove the marked nodes, node 57 is flagged, as it is the successor of the last marked node in the sequence. Node 57 is flagged for making sure that its *next* pointer does not change. Finally, all three nodes are physically unlinked from the list, together with the physical insertion of a new node, representing the old flagged one. I.e., although all three nodes were physically removed from the list, node 57 was not logically removed, as it was replaced by a new node with the same key. After the physical deletion at stage 5, the three nodes are retired (for more details, see Section 3.3). As opposed to Harris's implementation, the linearization point of both deletions is this physical removal, which atomically inserts the new representative into the list. Our mapping from modifications to nodes, maps this deletion to the new node, inserted at stage 5 (which represents the deletion of both nodes). In Section 3.2 we explain how this mapping is used for executing range queries.

3.1 The Linked-List Implementation

Our linked-list implementation, together with the list node class, is presented in Algorithm 1. The simple pointer access methods implementation (e.g., *mark()* in line 23 and *getRef()* in line 40) appear in Appendix A. In a similar way to Harris's list, the API includes the *insert()*, *remove()* and *contains()* operations ³. The *insert()* operation (lines 7–17) receives a key and a value. If there already exists a node with the given key in the list, it returns its value (line 11). Otherwise, it adds a new node with the given key and value to the list, and returns a designated NO_VAL answer (line 17). The *remove()* operation (lines 18–25) receives a key. If there exists a node with the given key in the list, it removes it and returns its value (line 25). Otherwise, it returns NO_VAL (line 22). The *contains()* operation (lines 26–30) receives a key. If there exists a node with the given key in the list, it returns its value (line 30). Otherwise, it returns NO_VAL (line 29).

All three API operations use the *find()* auxiliary method (lines 31–58), which receives a key and returns pointers to two nodes, *pred* and *curr* (line 58). As in Harris's implementation, it is guaranteed that at some point during the method execution, both nodes are consecutive reachable nodes in the list, *pred*'s key is strictly smaller than the input key, and *curr*'s key is equal or bigger than the given key. I.e., if *curr*'s key is strictly bigger than the input key, it is guaranteed that there is no node with the given input key in the list at this point. The method traverses the list, starting from the *head* sentinel node (line 33),

² When multiple nodes are physically removed together, it replaces the successor of the last node in the sequence of deleted nodes.

³ The *rangeQuery()* operation is added in Section 3.2. In addition, lines marked in blue in Algorithm 1 can be ignored at this point. They will also be discussed in Section 3.2

■ **Algorithm 1** Our Linked-List Implementation.

```

1: class Node
2:   Long ts
3:   K key
4:   V value
5:   Node* next
6:   Node* prior
7: procedure INSERT(key, val)
8:   while (true) do
9:     pred, curr ← FIND(key)
10:    if (curr → key == key)
11:      return curr → val
12:    n := alloc(key, val, ⊥)
13:    n → next := curr
14:    n → prior := curr
15:    if (CAS(&pred → next, curr, n))
16:      CAS(&n → ts, ⊥, getTS())
17:      return NO_VAL
18: procedure REMOVE(key)
19:   while (true) do
20:     pred, curr ← FIND(key)
21:     if (curr → key ≠ key)
22:       return NO_VAL
23:     if (!mark(curr)) continue
24:     FIND(key)      ▷ physical deletion
25:     return curr → val
26: procedure CONTAINS(key)
27:   pred, curr ← FIND(key)
28:   if (curr → key ≠ key)
29:     return NO_VAL
30:   else return curr → val
31: procedure FIND(key)
32:   retry:
33:     pred := head
34:     pNext := pred → next
35:     curr := getRef(pNext)
36:     while (true) do
37:       while(isMarkdOrFlagged(curr →
next))
38:         if (!getRef(curr → next))
39:           break
40:         curr := getRef(curr → next)
41:         if (curr → key ≥ key) break
42:         pred := curr
43:         pNext := pred → next
44:         if (isMarkdOrFlagged(pNext))
45:           goto retry
46:         curr := getRef(pNext)
47:         CAS(&pred → ts, ⊥, getTS())
48:         if (pNext ≠ curr)
49:           if (!TRIM(pred, getRef(pNext)))
50:             goto retry
51:         pNext := pred → next
52:         if (isMarkdOrFlagged(pNext))
53:           goto retry
54:         curr := pNext
55:         if ( isMarkedOrFlagged(curr →
next) ∨ curr → key < key)
56:           goto retry
57:         CAS(&curr → ts, ⊥, getTS())
58:         return pred, curr
59: procedure TRIM(pred, victim)
60:   curr := victim
61:   while (isMarked(curr → next)) do
62:     curr := getRef(curr → next)
63:     CAS(&curr → ts, ⊥, getTS())
64:     if (!flag(curr) ∧ !isFlagged(curr →
next))
65:       return false
66:     succ := getRef(curr → next)
67:     if (succ) CAS(&succ → ts, ⊥, getTS())
68:     newCurr := alloc(curr → key, curr → val,
⊥)
69:     newCurr → next := succ
70:     newCurr → prior := victim
71:     if (CAS(&pred → next, victim,
newCurr))
72:       CAS(&newCurr → ts, ⊥, getTS())
73:       return true
74:     return false

```

and until it gets to an unmarked and unflagged node with a key which is at least the input key (line 41). Recall that the two output variables are guaranteed to have been reachable, adjacent, unmarked and not flagged at some point during the method execution. Therefore, as long as the current traversed node is either marked or flagged (checked in line 37), the traversal continues, regardless of the current key (lines 37-40). Once the traversal terminates (either in line 38 or 41), if the current two nodes, saved in the *pred* and *curr* variables, are adjacent (the condition checked in line 48 does not hold), then the method returns them in line 58. Otherwise, similarly to the original implementation, the method is also in charge of physically removing marked nodes from the list.

As we are going to discuss next, our physical removal procedure, as depicted in Figure 1, is slightly different from the original one [26]. Physical deletions are executed via the *trim()* auxiliary method (lines 59–74). Although nodes are still marked for deletion in our implementation (line 23), their successful marking does not serve as the removal linearization point. I.e., reachable marked nodes are still considered as list members. The *trim()* method receives two nodes as its input parameters, *pred* and *victim*. *victim* is the physical removal candidate, and is assumed to already be marked. *pred* is assumed to be *victim*'s predecessor in the list, and to be neither marked nor flagged. As depicted in Figure 1, consecutive marked nodes are removed together. Therefore, the method traverses the list, starting from *victim*, for locating the first node which is not marked (lines 61–62). When such a node is found, the method tries to flag its *next* pointer, for freezing it until the removal procedure is done. In general, pointers are marked and flagged using their two least significant bits (which are practically redundant when reading node address aligned to a word). Both marked and flagged pointers are immutable, and a pointer cannot be both marked and flagged. Therefore, the flagging trial in line 64 fails if *curr*'s *next* pointer is either marked or flagged. If the flagging trial is unsuccessful, and not because some other thread has already flagged *curr*'s *next* pointer, the method returns in line 65. Otherwise, a new node is created in order to replace the flagged one (lines 68–70). Note that since this node's *next* pointer is flagged (i.e., immutable), it is guaranteed that the new node points to the original one's current successor. The actual trimming is executed in line 71. If the compare-and-swap (CAS) is successful, then the sequence of marked nodes, together with the single flagged one (at the end of the sequence), are atomically removed from the list, together with the insertion of the new copy of the flagged node (the new copy is neither flagged nor marked).

As the physical removal is necessary for linearizing the removal (as will be further discussed in Section 3.2), a remover must physically remove the deleted node before it returns from a *remove()* call. The marking of a node in line 23 only determines the remover's identity and announces its intention to delete the marked node. Therefore, the remover must additionally ensure that the node is indeed unlinked, by calling the *find()* method in line 24. In the full version of this paper [60] we formally prove that the list implementation, presented in Algorithm 1, is linearizable and lock-free.

3.2 Adding Range Queries

Given Algorithm 1, adding a linearizable range queries mechanism is relatively straight forward. We use a method which is similar to the vCAS technique [65]. As discussed in Section 1, the vCAS scheme introduces an extra level of indirection for the linked-list per node access. Indeed, we show in Section 4 that the vCAS implementation suffers from high overheads. The original vCAS paper provides a technique for avoiding this level of indirection. The suggested optimization relies on the following (very specific) assumption: a certain node can be the third input parameter to a successful CAS operation only once throughout the entire execution. That successful CAS is considered as the *recording* of this node, and the property is referred to as *recorded-once* in [65].

Although the recorded-once property yields a linearizable solution, which reduces memory and time overheads, this assumption does not hold in the presence of physical deletions, as they usually set the deleted node's predecessor to point to the deleted node's successor [26], or to another, already reachable node [14, 42], and then, this reachable node is recorded more than once. This makes the suggested technique inapplicable to Harris's linked-list [26] and most other concurrent data-structures (e.g., [12, 28, 42]). The original vCAS paper implemented a recorded-once binary search tree, based on [20], which we compare against in

Section 4. We extend the recorded-once condition and make it fit for the linked-list and other data-structures. We claim that associating each node with the data of a single data-structure update (as provided by our list) is enough for avoiding indirection in this setting. Given such an association, there is no need to store update-related data in designated version records, since it can be stored directly inside nodes, in a well-defined manner.

First, in a similar way to [3, 43, 65], we add a shared clock, for associating each node with a timestamp. The shared clock is read and updated using the *getTS()* and *fetchAddTS()* methods, respectively (see Appendix A). The shared clock is incremented whenever a range query is executed (e.g., see line 2 in Algorithm 2), and is read before setting a new node’s timestamp (e.g. see lines 16, 47, 57, 63, 67 and 72 in Algorithm 1). Next, we change the nodes layout (see our node class description in Algorithm 1). On top of the standard fields (i.e., key, value and *next* pointer), we add two extra fields to each node. The first field is the node’s timestamp (denoted as *ts*), representing its insertion into the list. Nodes’ *ts* fields are always initialized with a special \perp value (see lines 12 and 68 in Algorithm 1), to be given an actual timestamp after being inserted into the list. The second field, *prior*, points to the previous successor of this node’s first predecessor in the list (its predecessor when being inserted into the list). Both fields are set once and then remain immutable. E.g., consider the new node, inserted into the list at stage 5 in Figure 1. Its *prior* field points to the node whose key is 23, as this is the former successor of the node whose key is 9, which is the first predecessor of the newly inserted node. By its specification, once the *prior* field is set (see line 14 and 70 in Algorithm 1), it is immutable. These two new fields are not used during the list operations from Algorithm 1, but we do specify their proper initialization, in order to support linearizable range queries. Moreover (and in a similar way to [3, 65]), list inserts and deletes are linearized during the execution of the *getTS()* method, as follows. Let *n* be the node inserted into the list in line 15, during a successful *insert()* operation. *n*’s timestamp is set at some point, not later than the CAS in line 16 (it may be updated earlier, by a different thread). The *getTS()* invocation that precedes the successful update of *n*’s timestamp is the operation’s linearization point. In a similar way, consider a successful *remove()* operation. The removed node is unlinked from the list during a successful *trim()* execution. Let *newCurr* be the node successfully inserted into the list in line 71, during this successful *trim()* execution. *newCurr*’s timestamp is set at some point, not later than the CAS in line 72 (it may be updated earlier, by a different thread). The *getTS()* invocation that precedes the successful update of *newCurr*’s timestamp is the operation’s linearization point.

Our range queries mechanism is presented in Algorithm 2. The *rangeQuery()* operation receives three input parameters (see line 1): the lowest and highest keys in the range, and an output array for returning the actual keys and associated values in the range. In addition to filling this array, it also returns its accumulated size in the *count* variable. The operation starts by fetching and incrementing the global timestamp counter (line 2), which serves as the range query’s linearization point. I.e., the former timestamp is the one associated with the range query. This way, the range query is indeed linearized between its invocation and response, along with guaranteeing that the respective view is immutable during the operation (as new updates will be associated with the new timestamp).

After incrementing the global timestamp counter, the operation uses the *find()* auxiliary method in order to locate the first node in range (lines 4–12). As opposed to the vCAS mechanism [65], and in a similar way to the Bundles mechanism [43], we observe that until the traversal reaches the target range, there is no need to take timestamps into consideration. This observation is crucial for performance, as there is no need to traverse nodes via the

Algorithm 2 The Range Queries Mechanism.

```

1: procedure RANGEQUERY(low, high, 16:   while (succ → ts > ts) do
   *arr)                               17:       succ := succ → prior
2:   ts := fetchAddTS()                 18:       curr := succ
3:   currKey := low                      19:   count := 0
4:   while (true) do                   20:   while (curr → key ≤ high) do
5:     pred, curr ← FIND(currKey)       21:     arr[count] → key := curr → key
6:     currKey := pred → key            22:     arr[count] → value := curr → value
7:     while (pred → ts > ts) do
8:       pred := pred → prior           23:     count := count + 1
9:     if (pred → key ≤ low)             24:     succ := getRef(curr → next)
10:      curr := pred                    25:     CAS(&succ → ts, ⊥, getTS())
11:      break                            26:     while (succ → ts > ts) do
12:      ts := getTS() - 1                27:       succ := succ → prior
13:   while (curr → key < low) do         28:     curr := succ
14:     succ := getRef(curr → next)       29:   return count
15:     CAS(&succ → ts, ⊥, getTS())

```

prior fields (which produce longer traversals in practice). In addition, it enables using the fast index (described in Section 3.4) for enhancing the search. During each loop iteration, we first find a node with a key which is smaller than the lowest key in the range (saved as the *pred* variable in line 5). Then, we optimistically try to find a relatively close node, following *prior* pointers, until we get to a small enough timestamp (lines 7–8). Since this search may result in a node with a bigger key (e.g., see line 14 in Algorithm 1), the next iteration sends a smaller key as input to the *find()* execution in line 5. Note that in the worst case scenario, the loop in lines 4–12 stops after the *find()* execution in line 5 outputs the *head* sentinel node (as its timestamp is necessarily smaller than *ts*). Therefore, it never runs infinitely. The purpose of updating *ts* in line 12 will be clarified in Section 3.3, as it is related to the VBR mechanism. Note that in any case, this update does not foil correctness, since it is still guaranteed that the range query is linearized between the operation’s invocation and response.

When *pred* has a key which is smaller than the range lower bound, the operation moves on to the next step (the loop breaks in line 11). At this point, the traversal continues according to the respective timestamp⁴, until getting to a node with a key which is at least the range lower bound (lines 13–18). Once a node with a big enough key is found, the traversal continues in lines 20–28. At this stage, the *count* output variable and the output array are updated according to the data accumulated during the range traversal. Finally, the *count* output variable, indicating the total number of keys in range, is returned in line 29. Note that throughout the traversals in lines 16–17 and lines 26–27, there is no need to update *succ*’s timestamp (as done in lines 15 and 25), since it serves as a node’s *prior* and thus, is guaranteed to already have an updated timestamp (for more details, see the full version of this paper [60]).

⁴ In the full version of this paper [60] we prove that a node’s successor at timestamp *T* can be found by starting from its current successor and then following *prior* references until reaching a node with a timestamp which is not greater than *T*.

3.3 Adding A Safe Memory Reclamation Mechanism

Before integrating our list with a manual memory reclamation mechanism, we must first install *retire()* invocations, for announcing that a node’s memory space is available for re-allocation. Naturally, nodes are retired after unsuccessful insertions, or after they are unlinked from the list. I.e., n is retired if the CAS in line 15 is unsuccessful, *newCurr* is retired if the CAS in line 71 is unsuccessful, and upon a successful trimming in line 71, the unlinked nodes are retired, starting from *victim*. The last retired node is *curr*, which is replaced by its new representative in the list, *newCurr*. Note that we do not handle physical removals of *prior* links. Handling them is unnecessary, and might cause significant overheads, both to the list operations and to the reclamation procedure. Therefore, retired nodes are still reachable from the list head during retirement: *newCurr*’s *prior* field points to *victim*, making all of the unlinked nodes reachable via this pointer (and their *next* pointers).

To add a safe memory reclamation mechanism to our list, we use an improved variant of Version Based Reclamation (VBR) [58]. VBR cannot be integrated as is. Similarly to most safe memory reclamation techniques, it assumes that retired objects are not reachable via the data-structure links. This assumption is crucial to the correctness of VBR, as retired objects may be immediately reclaimed. In addition, VBR uses a slow ticking epoch clock, and ensures that the clock ticks at least once between the retirement and future re-allocation of the same node. During execution, the operating threads constantly check that the global epoch clock has not changed. Upon a clock tick, they conservatively treat all data read from shared memory as stale, and move control to an adequate previous point in the code in order to read a fresh value. As long as the clock does not tick, threads may continue executing without worrying about use-after-free issues. The intuition is that if a node is accessed during a certain epoch, then it must have been reachable during this epoch. I.e., even if this node has already been retired, its retirement was during the current epoch, which means that it has not been re-allocated yet (as the clock has not ticked yet).

Our list implementation poses a new challenge in this context. Suppose that the current epoch is E , and that a certain node, n , is currently in the list (i.e., it has not been unlinked using the *trim()* method yet). In addition, suppose that n ’s *prior* field points to another node, m , that has been retired during an earlier epoch. Then m may be reclaimed and re-allocated during E . A traversing thread may access n ’s *prior* field during E , without getting any indication to the fact that the referenced node is a stale value. Another problem, which does not affect correctness, but may cause frequent thread starvations, is that the global epoch clock is likely to tick during a long range query. In the original VBR scheme, a clock tick forces the executing thread to start its traversal from scratch, even if it has not encountered any reclaimed node in practice.

In order to overcome the above problems, we made some small adjustments to the original VBR scheme. First, we kept the global epoch clock of VBR and the timestamp clock of the range queries separated. We separated the two, as VBR works best with a (very) slow ticking clock. Read-intensive workloads (in which range queries dominate the execution) incur high overheads when combining the two clocks. The separation of the two independent clocks helps overcome the potential aborts. The second step was to modify the nodes’ layout (The VBR-integrated node layout appears in the full version of this paper [60]). Recall that the VBR scheme adds a *birth epoch* to every node, along with a version per mutable field. Non-pointer mutable fields are associated with the node’s birth epoch, and pointers are associated with a version which is the maximum between the birth epoch of the node and the birth epoch of its successor. Our list nodes have two mutable fields, their timestamp ts (changes only once), and their *next* pointer. The *prior* pointers are immutable. Accordingly, we

associated the node's timestamp with its VBR-integrated birth epoch, serving as its version (there was no need to add an extra *ts* version), and added a designated *next* pointer version. Writes to the mutable fields are handled exactly as in the original VBR scheme. Accordingly, upon allocation, a node's timestamp is initialized to \perp , along with the current VBR epoch as its associated birth epoch. When the timestamp is updated (see line 16, 47, 57, 63, 67 and 72 in Algorithm 1, or lines 15 and 25 in Algorithm 2), the birth epoch (also serving as the timestamp's version) does not change, as the two fields are accessed together, via a wide-compare-and-swap (WCAS) instruction. Similarly, *next* pointers are associated with the maximum between the two respective birth epochs, and are also updated using WCAS.

Reads are handled in a different manner from the original VBR, as the problems we mentioned above must be treated with special care. The original VBR repeatedly reads the global epoch in order to make sure that it has not changed. In our extended VBR variant, it is read once. After reading the global epoch, and as long as the executing thread does not encounter a birth epoch or a version which is bigger than this epoch, it may continue executing its code. The motivation behind this behavior is that even if a certain node in the system has meanwhile been reclaimed, this node does not pose a problem as long as the current thread does not encounter it. Therefore, traversing threads follow three guidelines: (1) a node's birth epoch is read again after each read of another field, (2) after dereferencing a *next* pointer, the reader additionally makes sure that the successor's birth epoch is not greater than the pointer's version, and (3) after dereferencing a *prior* pointer (which is not associated with a version), the reader additionally makes sure that the successor's birth epoch is not greater than the predecessor's birth epoch. If any of these conditions does not hold, then the reader needs to proceed according to the original VBR's protocol. In the full version of this paper [60] we prove that these three guidelines are sufficient for maintaining correctness. Upon an epoch change, the original VBR enforces a rollback to a predefined checkpoint in the code. Accordingly, we install code checkpoints. Whenever a check that our guidelines impose fails, the executing thread rolls-back to the respective checkpoint. Checkpoints are installed in the beginning of each API operation (i.e., *insert()*, *remove()*, *contains()* and *rangeQuery()*). Another checkpoint is installed after a successful marking in line 23 of Algorithm 1, as the identity of the marking thread affects linearizability (and therefore, a rollback to the beginning of the operation would foil linearizability). Note that a successful insertion in line 15 does not force a checkpoint (although it affects linearizability, by setting the inserter identity), as it is not followed by any reads of potentially reclaimed memory.

Another issue that needs to be dealt with is the guarantee that life-cycles of nodes, allocated from the same memory address, do not overlap. The original VBR scheme does so by associating each node with a retire epoch. A node's retire epoch is set upon retirement. During re-allocation, if the current global epoch is equal to the node's retire epoch, then the global epoch is incremented before re-allocation. We chose to optimize over the original VBR, discarding the retire epoch field, as it adds an extra field per allocated node. Instead, each retire list is associated with the epoch, recorded once it is full (right before it is returned to the global pool of nodes). Upon pulling such a list from the global pool, if its associated epoch is equal to the current one, then the global epoch is incremented.

Finally, consider the following scenario. Suppose that a thread T_1 is running a range query, the current epoch is E and the current global timestamp is t . Next, suppose that another thread, T_2 , reclaims a node n that has been retired during E , and that is relevant for T_1 's range query. Starting from this point, whenever T_1 accesses the newly allocated node, it rolls back and starts its traversal from scratch (as the new node has a birth epoch which is

greater than its predecessor through the *prior* pointer, foiling guideline 3). As long as T_1 's *ts* variable does not change, T_1 will infinitely get to the new allocated node and then roll-back to the beginning. We reduce the probability of such scenarios in practice, by updating the *ts* variable in line 12 of Algorithm 2. We further ensure that the current global timestamp is up-to-date by incrementing it upon each re-allocation, if necessary.

3.4 Adding A Fast Index

Our linked-list implementation encapsulates the key-value pairs and enables the timestamps mechanism. However, when key ranges are large, the linked-list does not perform as good as other concurrent data-structures. It forces a linear traversal per operation, as opposed to skip lists [24, 28] and binary search trees [14, 42]. We observe that the index links in such data-structures (e.g., the links connecting the upper levels in a skip list or the inner levels in a tree) are only required for fast access. The actual data exists only in the lowest level of the skip list (or the leaves of an external tree). Therefore, we allow a simple integration of an external index, enabling fast access instead of long traversals. The index should provide an *insert(key, node)* operation, receiving a key and a node pointer as its associated value. It additionally should provide a *remove(key)* operation. Finally, instead of providing a *contains(key)* operation, it should provide a *findPred(key)* operation, receiving a key and returning a pointer to the node associated with some key which is smaller than the given one.

The *findPred(key)* can be naively implemented by calling the data-structure search method (there usually exists such method. E.g., [26, 28, 42]) with a smaller key as input. I.e., it is possible to search for key minus 2 or minus 10. Obviously, this does not guarantee that a suitable node will indeed be returned. However, if the selected smaller key is not small enough, it is possible to start a new trial and search for a smaller key. Our experiments showed that limiting the number of such trials per search to a small constant (e.g., 5 in our experiments) is negligible in terms of performance, and is usually enough for locating a relevant node. In addition, the index is used only for fast access, so correctness is not affected even if all trials fail. Specifically, the *findPred(key)* operation can be easily implemented for a skip list, using its built-in search auxiliary method [24, 28] (as it returns a predecessor with a smaller key). Examples for using a skip list as a fast index have already been introduced for linearizable data-structures [57] and in the transactional memory setting [63]. The *findPred(key)* operation can also be implemented for some binary search trees, by traversing the left child, instead of the right child, at some point during the search path. We applied this method when implementing our tree index, based on Natarajan and Mittal's BST [42].

■ **Algorithm 3** Starting a new traversal using the index.

<pre> 1: currKey := key 2: attempts := MAX_ATTEMPTS 3: while (attempts \neq 0) do 4: attempts-- 5: pred := index \rightarrow findPred(currKey) 6: predTS := pred \rightarrow timestamp 7: predNext := pred \rightarrow next 8: predKey := pred \rightarrow key 9: if (pred \rightarrow birth > currEpoch) 10: rollback </pre>	<pre> 11: else if (predKey \geq key \vee predTS == \perp) 12: continue 13: else if (isMarkedOrFlagged(predNext)) 14: currKey := predKey 15: else break 16: if (attempts == 0) 17: pred := head 18: predNext := pred \rightarrow next </pre>
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

We update the fast index as follows: New Nodes are inserted into the index after being inserted into the list (i.e., right before the *insert()* operation returns in line 17 of Algorithm 1). Nodes are removed from the index after being removed from the list, and right before being retired (see Section 3.3). Note that the *curr* node, replaced by *newCurr* via the CAS in line 71, should be removed from the index, followed by an insertion of *newCurr*. In our implemented index, we have implemented an *update()* operation instead. This operation receives as input a node reference, and uses it to replace a node with the same key in the index⁵. In case the external index does not provide an *update()* operation, this also may be executed via the standard *remove()* operation, followed by a respective *insert()* operation. The index is read only during the *find()* auxiliary method. Instead of starting each list traversal from the *head* sentinel node (see line 33 in Algorithm 1), the traversing thread tries to shorten the traversal by accessing the fast index. I.e., instead of executing the code in lines 33-34 of Algorithm 1, each thread executes the code from Algorithm 3. It starts by initializing the searched key to the input key (received as input in line 31 of Algorithm 1). Then, after finding the alleged predecessor, using the *findPred()* operation (line 5), if its birth epoch is bigger than the last recorded one, the executing thread rolls-back to its last recorded checkpoint (see Section 3.3). Otherwise, if *pred*'s key is not smaller than the given input key, or its timestamp is not initialized yet, the thread starts another trial, with the same key. Otherwise, if *pred* is either marked or flagged (line 13), the thread starts another trial, with a smaller key. Otherwise, it is guaranteed that *pred* and *predNext* hold a valid node and its (unmarked and unflagged) *next* pointer, respectively, and the thread may start its list traversal from line 35 of Algorithm 1. Note that the code presented in Algorithm 3 always terminates, as in the worst case scenario, the loop breaks after a predefined number of attempts.

4 Evaluation

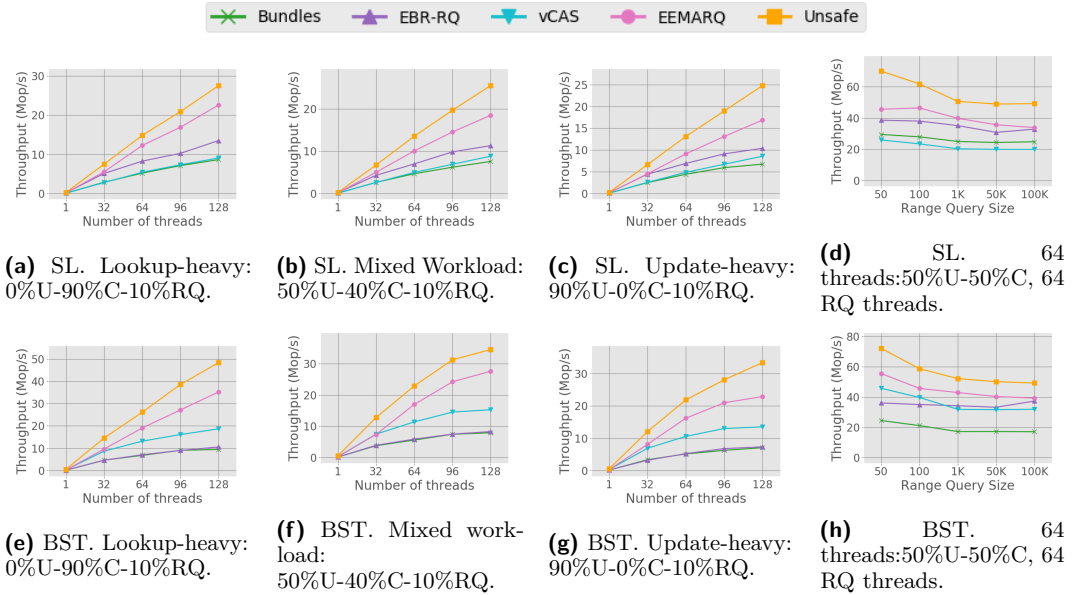
For evaluating throughput of EEMARQ, we implemented⁶ the linked-list presented in Section 3⁷, including the extended VBR variant, as described in Section 3.3. In addition, we implemented the lock-free skip list from [24] and the lock-free BST from [42]. Both the skip list and the tree were used on top of the linked list, and served as fast indexes, as described in Section 3.4. Deleted nodes from both indexes were manually reclaimed, using the original VBR scheme, according to the integration guidelines from [58] (without the adjustments described in Section 3.3). Each data-structure had its own objects pool (as VBR forces type preservation). Retire lists had 64 entries. Since VBR allows the immediate reclamation of retired nodes, retire lists were reclaimed as a whole every time they contained 64 nodes. I.e., at most 8192 (64 retired nodes X 128 threads) objects were over-provisioned per data-structure at any given moment.

We compared EEMARQ against four competitors, all using epoch-based reclamation. EBR-RQ is the lock-free epoch-based range queries technique by Arbel-Raviv and Brown [3], vCAS is the lock-free technique by Wei et al. [65], Bundles is the lock-based bundled references technique by Nelson et al. [43], and Unsafe uses a naive non-linearizable scan of the nodes in the range without synchronizing with concurrent updates (used as our baseline). We

⁵ The implemented *update()* operation also takes the node's birth epoch into account, and does not replace a node with a reclaimed node or with a node with a smaller birth epoch

⁶ The code is available here.

⁷ For avoiding unnecessary accesses to the global timestamps clock, the *ts* field updates from Algorithm 1 and Algorithm 2 were executed only after the *ts* field was read, and only if it was still equal to \perp .



■ **Figure 2** Throughput evaluation under various workloads for the skip list (2a–2d) and the tree (2e–2h). The key range is 1M. In Figures 2a–2c and 2e–2g, the range query size is 1000. Y axis: throughput in million operations per second. X axis: #threads in Figures 2a–2c and 2e–2g, and range query size in Figures 2d and 2h.

did not compare EEMARQ against RLU [38], as its mechanism is not linearizable, and it was also shown to be slower than our competitors [43]. For EBR-RQ and Bundles, we used the implementation provided by the authors. The Unsafe code was provided by the EBR-RQ authors. The vCAS authors provided a vCAS-based lock-free BST, including their optimization for avoiding indirection (see Section 3.2 for more details). Since there does not exist any respective skip list implementation, we implemented a vCAS-based skip list according to the guidelines from [65]. The vCAS-based skip list was not optimized, as the optimization technique, suggested in [65], does not fit to this data-structure. For our competitors’ memory reclamation, we used the original implementations, provided by [3, 43, 65], without any code or object pools usage changes. In particular, memory was not returned to the operating system. I.e., all implementations used pre-allocated object pools [58, 61] for reclaiming memory.

Setup. We conducted our experiments on a machine running Linux (Ubuntu 20.04.4), equipped with 2 Intel Xeon Gold 6338 2.0GHz processors. Each processor had 32 cores, each capable of running 2 hyper-threads to a total of 128 threads overall. The machine used 256GB RAM, an L1 data cache of 3MB and an L1 instruction cache of 2MB, an L2 unified cache of 80MB, and an L3 unified cache of 96MB. The code was written in C++ and compiled using the GCC compiler version 9.4.0 with `-std=c++11 -O3 -mcx16`. Each test was a fixed-time micro benchmark in which threads randomly call the `insert()`, `remove()`, `contains()` and `rangeQuery()` operations according to different workload profiles. We ran the experiments with a range of 1M keys. Each execution started by pre-filling the data-structure to half of its range size, and lasted 10 seconds (longer experiments showed similar results). Each experiment was executed 10 times, and the average throughput across all executions

is reported. Figure 2 shows the skip list and tree scalability under various workloads. The updates (half *insert()* and half *remove()*), *contains()* and *rangeQuery()* percentiles appear under each graph. Figures 2a-2c and 2e-2g show the skip list and tree scalability as a function of the number of executing threads, under a variety of workloads. All queries have a fixed range of 1K keys (following [65]). Figures 2d and 2h show the effect of varying range query size on the skip list and tree performance. In these experiments, 64 threads perform 25% *insert()*, 25% *remove()* and 50% *contains()*, and 64 threads perform range queries only. In the full version of this paper [60] we present the respective range queries and updates throughput for Figures 2d and 2h, along with additional results for other workloads.

Discussion. The EEMARQ skip list surpasses the next best algorithm, EBR-RQ, by up to 65% in the lookup-heavy workload (Figure 2a), by up to 50% in the mixed workload (Figure 2b), and by up to 70% in the update-heavy workload (Figure 2c). The EEMARQ tree surpasses its next best algorithm, vCAS, by up to 75% in the lookup-heavy workload (Figure 2e), by up to 65% in the mixed workload (Figure 2f), and by up to 70% in the update-heavy workload (Figure 2g).

The results show that avoiding indirection is crucial to performance. In particular, EEMARQ outperforms its competitors in the lookup-heavy workloads (Figures 2a and 2e), in which memory is never reclaimed. I.e., its range query mechanism, which completely avoids traversing separate version nodes, has a significant advantage under such workloads. It can also be seen when examining EEMARQ’s competitors. While the vCAS-based tree (which avoids indirection) is EEMARQ’s next best competitor, the vCAS-based skip list (that involves the traversal of designated version nodes) is the weakest among the skip list implementations. In addition, the Bundles technique, which employs such a level of indirection when range queries are executed, also performs worse than most competitors, under most workloads.

Under update-dominated workloads, EEMARQ is faster also thanks to its efficient memory reclamation method. While all other algorithms use EBR as their memory reclamation scheme, EEMARQ enjoys VBR’s inherent locality and fast reclamation process. Moreover, EEMARQ avoids the original VBR’s frequent accesses to the global epoch clock, as described in Section 3.3. Although using VBR forces rollbacks when accessing reclaimed nodes, our fast index mechanism allows a fast retry, which makes the impact of rollbacks small. Indeed, experiments with longer retire lists (i.e., fewer rollbacks) showed similar results. This is clearly shown in Figures 2d and 2h: EEMARQ outperforms its competitors when the query ranges are big (10% of the data-structure range). I.e., possible frequent rollbacks do not prevent EEMARQ from outperforming all other competitors.

5 Conclusion

We presented EEMARQ, a design for a lock-free data-structure that supports linearizable inserts, deletes, contains, and range queries. Our design starts from a linked-list, which is easier to use with MVCC for fast range queries. We add lock-free memory reclamation to obtain full lock-freedom. Finally, we facilitate an easy integration of a fast external index to speed up the execution, while still providing full linearizability and lock-freedom. As the external index does not require version maintenance, it can remain simple and fast. We implemented the design with a skip list and a binary search tree as two possible fast indexes, and evaluated their performance against state-of-the-art solutions. Evaluation shows that EEMARQ outperforms existing solutions across read-intensive and update-intensive

workloads, and for varying range query sizes. In addition, EEMARQ’s memory footprint is relatively low, thanks to its tailored reclamation scheme, enabling the immediate reclamation of deleted objects. EEMARQ is the only technique that provides lock-freedom, as other existing methods use blocking memory reclamation schemes.

References

- 1 Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM (JACM)*, 40(4):873–890, 1993.
- 2 Panagiotis Antonopoulos, Peter Byrne, Wayne Chen, Cristian Diaconu, Raghavendra Thallam Kodandaramaih, Hanuma Kodavalla, Prashanth Purnananda, Adrian-Leonard Radu, Chaitanya Sreenivas Ravella, and Girish Mittur Venkataramanappa. Constant time recovery in azure sql database. *Proceedings of the VLDB Endowment*, 12(12):2143–2154, 2019.
- 3 Maya Arbel-Raviv and Trevor Brown. Harnessing epoch-based reclamation for efficient range queries. *ACM SIGPLAN Notices*, 53(1):14–27, 2018.
- 4 Hagit Attiya, Rachid Guerraoui, and Eric Ruppert. Partial snapshot objects. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 336–343, 2008.
- 5 Hillel Avni, Nir Shavit, and Adi Suissa. Leaplist: lessons learned in designing tm-supported range queries. In *Proceedings of the 2013 ACM symposium on Principles of distributed computing*, pages 299–308, 2013.
- 6 Daniel Bartholomew. *MariaDB cookbook*. Packt Publishing Ltd, 2014.
- 7 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.
- 8 Naama Ben-David, Guy E Blelloch, Panagiota Fatourou, Eric Ruppert, Yihan Sun, and Yuanhao Wei. Space and time bounded multiversion garbage collection. *arXiv preprint arXiv:2108.02775*, 2021.
- 9 Philip A Bernstein and Nathan Goodman. Concurrency control algorithms for multiversion database systems. In *Proceedings of the first ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 209–215, 1982.
- 10 Philip A Bernstein and Nathan Goodman. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems (TODS)*, 8(4):465–483, 1983.
- 11 Jan Böttcher, Viktor Leis, Thomas Neumann, and Alfons Kemper. Scalable garbage collection for in-memory mvcc systems. *Proceedings of the VLDB Endowment*, 13(2):128–141, 2019.
- 12 Nathan G Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. *ACM Sigplan Notices*, 45(5):257–268, 2010.
- 13 Trevor Brown and Hillel Avni. Range queries in non-blocking k-ary search trees. In *International Conference On Principles Of Distributed Systems*, pages 31–45. Springer, 2012.
- 14 Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 329–342, 2014.
- 15 Trevor Alexander Brown. Reclaiming memory for lock-free data structures: There has to be a better way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 261–270, 2015.
- 16 Bapi Chatterjee. Lock-free linearizable 1-dimensional range queries. In *Proceedings of the 18th International Conference on Distributed Computing and Networking*, pages 1–10, 2017.
- 17 Andreia Correia, Pedro Ramalhete, and Pascal Felber. Orcgc: automatic lock-free memory reclamation. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 205–218, 2021.

- 18 David L Detlefs, Paul A Martin, Mark Moir, and Guy L Steele Jr. Lock-free reference counting. *Distributed Computing*, 15(4):255–271, 2002.
- 19 Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: Sql server’s memory-optimized oltp engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1243–1254, 2013.
- 20 Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 131–140, 2010.
- 21 Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. The sap hana database—an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.
- 22 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.
- 23 Sérgio Miguel Fernandes and Joao Cachopo. Lock-free and scalable multi-version software transactional memory. *ACM SIGPLAN Notices*, 46(8):179–188, 2011.
- 24 Keir Fraser. Practical lock-freedom. Technical report, University of Cambridge, Computer Laboratory, 2004.
- 25 Theo Härder. Observations on optimistic concurrency control schemes. *Information Systems*, 9(2):111–120, 1984.
- 26 Timothy L Harris. A pragmatic implementation of non-blocking linked-lists. In *International Symposium on Distributed Computing*, pages 300–314. Springer, 2001.
- 27 Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Transactions on Computer Systems (TOCS)*, 23(2):146–196, 2005.
- 28 Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. *The art of multiprocessor programming*. Newnes, 2020.
- 29 Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.
- 30 Nikolaos D Kallimanis and Eleni Kanellou. Wait-free concurrent graph objects with dynamic traversals. In *19th International Conference on Principles of Distributed Systems (OPODIS 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.
- 31 Idit Keidar and Dmitri Perelman. Multi-versioning in transactional memory. In *Transactional Memory. Foundations, Algorithms, Tools, and Applications*, pages 150–165. Springer, 2015.
- 32 Alfons Kemper and Thomas Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*, pages 195–206. IEEE, 2011.
- 33 Jongbin Kim, Hyunsoo Cho, Kihwang Kim, Jaeseon Yu, Sooyong Kang, and Hyungsoo Jung. Long-lived transactions made less harmful. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 495–510, 2020.
- 34 Jongbin Kim, Kihwang Kim, Hyunsoo Cho, Jaeseon Yu, Sooyong Kang, and Hyungsoo Jung. Rethink the scan in mvcc databases. In *Proceedings of the 2021 International Conference on Management of Data*, pages 938–950, 2021.
- 35 Priyanka Kumar, Sathya Peri, and K Vidyasankar. A timestamp based multi-version stm algorithm. In *International Conference on Distributed Computing and Networking*, pages 212–226. Springer, 2014.
- 36 Juchang Lee, Hyungyu Shin, Chang Gyoo Park, Seongyun Ko, Jaeyun Noh, Yongjae Chuh, Wolfgang Stephan, and Wook-Shin Han. Hybrid garbage collection for multi-version concurrency control in sap hana. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1307–1318, 2016.

- 37 Li Lu and Michael L Scott. Generic multiversion stm. In *International Symposium on Distributed Computing*, pages 134–148. Springer, 2013.
- 38 Alexander Matveev, Nir Shavit, Pascal Felber, and Patrick Marlier. Read-log-update: a lightweight synchronization mechanism for concurrent programming. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 168–183, 2015.
- 39 Paul E McKenney and John D Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, volume 509518, 1998.
- 40 Maged M Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, 2004.
- 41 AB MySQL. Mysql, 2001.
- 42 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.
- 43 Jacob Nelson-Slivon, Ahmed Hassan, and Roberto Palmieri. Bundling linked data structures for linearizable range queries. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 368–384, 2022.
- 44 Yiannis Nikolakopoulos, Anders Gidenstam, Marina Papatriantafidou, and Philippas Tsigas. A consistency framework for iteration operations in concurrent data structures. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 239–248. IEEE, 2015.
- 45 Yiannis Nikolakopoulos, Anders Gidenstam, Marina Papatriantafidou, and Philippas Tsigas. Of concurrent data structures and iterations. In *Algorithms, Probability, Networks, and Games*, pages 358–369. Springer, 2015.
- 46 Christos H Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM (JACM)*, 26(4):631–653, 1979.
- 47 Christos H Papadimitriou and Paris C Kanellakis. On concurrency control by multiple versions. *ACM Transactions on Database Systems (TODS)*, 9(1):89–99, 1984.
- 48 Dmitri Perelman, Anton Byshevsky, Oleg Litmanovich, and Idit Keidar. Smv: Selective multi-versioning stm. In *International Symposium on Distributed Computing*, pages 125–140. Springer, 2011.
- 49 Dmitri Perelman, Rui Fan, and Idit Keidar. On maintaining multiple versions in stm. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 16–25, 2010.
- 50 Erez Petrank and Shahar Timnat. Lock-free data-structure iterators. In *International Symposium on Distributed Computing*, pages 224–238. Springer, 2013.
- 51 Hasso Plattner. A common database approach for oltp and olap using an in-memory column database. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 1–2, 2009.
- 52 Behandelt PostgreSQL. Postgresql. *Web resource: <http://www.PostgreSQL.org/about>*, 1996.
- 53 Aleksandar Prokopec. Snapqueue: lock-free queue with constant time snapshots. In *Proceedings of the 6th ACM SIGPLAN Symposium on Scala*, pages 1–12, 2015.
- 54 Aleksandar Prokopec, Nathan Grasso Bronson, Phil Bagwell, and Martin Odersky. Concurrent tries with efficient non-blocking snapshots. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 151–160, 2012.
- 55 Pedro Ramalhete and Andreia Correia. Brief announcement: Hazard eras-non-blocking memory reclamation. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 367–369, 2017.
- 56 G Satyanarayana Reddy, Rallabandi Srinivasu, M Poorna Chander Rao, and Srikanth Reddy Rikkula. Data warehousing, data mining, olap and oltp technologies are essential elements to support decision-making process in industries. *International Journal on Computer Science and Engineering*, 2(9):2865–2873, 2010.

- 57 Gali Sheffi, Guy Golan-Gueta, and Erez Petrank. A scalable linearizable multi-index table. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 200–211. IEEE, 2018.
- 58 Gali Sheffi, Maurice Herlihy, and Erez Petrank. VBR: Version Based Reclamation. In Seth Gilbert, editor, *35th International Symposium on Distributed Computing (DISC 2021)*, volume 209 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 35:1–35:18, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.DISC.2021.35.
- 59 Gali Sheffi, Maurice Herlihy, and Erez Petrank. Vbr: Version based reclamation. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 443–445, 2021.
- 60 Gali Sheffi, Pedro Ramalhete, and Erez Petrank. Eemarq: Efficient lock-free range queries with memory reclamation. *arXiv preprint arXiv:2210.17086*, 2022.
- 61 Ajay Singh, Trevor Brown, and Ali Mashtizadeh. Nbr: neutralization based reclamation. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 175–190, 2021.
- 62 Daniel Solomon and Adam Morrison. Efficiently reclaiming memory in concurrent search data structures while bounding wasted memory. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 191–204, 2021.
- 63 Alexander Spiegelman, Guy Golan-Gueta, and Idit Keidar. Transactional data structure libraries. *ACM SIGPLAN Notices*, 51(6):682–696, 2016.
- 64 Alexander Thomasian and In Kyung Ryu. Performance analysis of two-phase locking. *IEEE Transactions on Software Engineering*, 17(5):386, 1991.
- 65 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.
- 66 Haosen Wen, Joseph Izraelevitz, Wentao Cai, H Alan Beadle, and Michael L Scott. Interval-based memory reclamation. *ACM SIGPLAN Notices*, 53(1):1–13, 2018.
- 67 Kjell Winblad, Konstantinos Sagonas, and Bengt Jonsson. Lock-free contention adapting search trees. *ACM Transactions on Parallel Computing (TOPC)*, 8(2):1–38, 2021.
- 68 Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proceedings of the VLDB Endowment*, 10(7):781–792, 2017.
- 69 Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. Efficient lock-free durable sets. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–26, 2019.

A Auxiliary Methods and Initialization

As described in Section 3.1 and Algorithm 1 and 2, the linked-list represents a map of key-value pairs. Each pair is represented by a node, consisting of five fields: the mutable *ts* field represents its associated timestamp, the immutable *key* and *value* fields represent the pair’s key and value, respectively, the mutable *next* field holds a pointer to the node’s successor in the list, and the immutable *prior* field points to the previous successor of its first predecessor in the list.

The list is initialized as follows: The global timestamps clock is initialized to 2, and the \perp constant, representing an uninitialized timestamp, is set to 1. The list has a single entry point, which is a pointer to the *head* sentinel node. *head*’s key is the minimal key in the key range (denoted as $-\infty$). Its timestamp is set to the initial system timestamp (2 in our implementation) and its next pointer points to the *tail* sentinel node. *tail*’s key is

the maximal key in the key range (denoted as ∞), its timestamp is equal to *head*'s, and its next pointer points to null. Both *prior* fields point to null, as *head* has no predecessor, and *tail*'s predecessor (which is *head*) has no previous successor. After initialization, the list is considered as empty (the sentinel nodes do not represent map items).

■ **Algorithm 4** Our Auxiliary Methods Implementation.

<pre> 1: getTS() 2: return globalTS.load() 3: fetchAddTS() 4: return globalTS.fetch&add() 5: MARK_MASK := 0x1 6: FLAG_MASK := 0x2 7: AUX_MASK := 0x3 8: isMarked(ptr) 9: if (ptr & MARK_MASK) return true 10: return false 11: isFlagged(ptr) 12: if (ptr & FLAG_MASK) return true 13: return false 14: isMarkedOrFlagged(ptr) 15: if (ptr & AUX_MASK) return true 16: return false </pre>	<pre> 17: getRef(ptr) 18: return ptr & ~ AUX_MASK 19: mark(node) 20: ptr := node → next 21: if (isMarkedOrFlagged(ptr)) return false 22: markedPtr := ptr ∨ MARK_MASK 23: return CAS(&node → next, ptr, markedPtr) 24: flag(node) 25: ptr := node → next 26: if (isMarkedOrFlagged(ptr)) return false 27: flaggeddPtr := ptr ∨ FLAG_MASK 28: return CAS(&node → next, ptr, flagged- dPtr) </pre>
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

The pseudo code for the global timestamps clock and node pointers access auxiliary methods appears in Algorithm 4. The *getTS()* method (lines 1-2) is used to atomically read the global timestamps clock, and the *fetchAddTS()* method (lines 3-4) is used to atomically update it.

We use the pointer's two least significant bits for encapsulating the mark and flag bits (see lines 5-7). The *isMarked()* (lines 8-10), *isFlagged()* (lines 11-13), and *isMarkedOrFlagged()* (lines 14-16) methods receive a pointer and return an answer using the relevant bit mask. The *getRef()* method (lines 17-18) receives a (potentially marked or flagged) pointer and returns the actual reference, ignoring the mark and flag bits. The *mark()* (lines 19-23) and *flag()* (lines 24-28) methods receive a node reference (the input pointer is assumed to be unmarked and unflagged), dereference it, and mark or flag the node's next pointer (respectively), assuming it is neither marked nor flagged.

The Step Complexity of Multidimensional Approximate Agreement

Hagit Attiya  

Department of Computer Science, Technion, Israel

Faith Ellen  

Department of Computer Science, University of Toronto, Canada

Abstract

Approximate agreement allows a set of n processes to obtain outputs that are within a specified distance $\epsilon > 0$ of one another and within the convex hull of the inputs.

When the inputs are real numbers, there is a wait-free shared-memory approximate agreement algorithm [16] whose step complexity is in $O(n \log(S/\epsilon))$, where S , the *spread* of the inputs, is the maximal distance between inputs. There is another wait-free algorithm [17] that avoids the dependence on n and achieves $O(\log(M/\epsilon))$ step complexity where M , the *magnitude* of the inputs, is the absolute value of the maximal input.

This paper considers whether it is possible to obtain an approximate agreement algorithm whose step complexity depends on neither n nor the magnitude of the inputs, which can be much larger than their spread. On the negative side, we prove that $\Omega\left(\min\left\{\frac{\log M}{\log \log M}, \frac{\sqrt{\log n}}{\log \log n}\right\}\right)$ is a lower bound on the step complexity of approximate agreement, even when the inputs are real numbers. On the positive side, we prove that a polylogarithmic dependence on n and S/ϵ can be achieved, by presenting an approximate agreement algorithm with $O(\log n(\log n + \log(S/\epsilon)))$ step complexity. Our algorithm works for multidimensional domains. The step complexity can be further restricted to be in $O(\min\{\log n(\log n + \log(S/\epsilon)), \log(M/\epsilon)\})$ when the inputs are real numbers.

2012 ACM Subject Classification Theory of computation \rightarrow Shared memory algorithms; Theory of computation \rightarrow Distributed algorithms

Keywords and phrases approximate agreement, conflict detection, shared memory, wait-freedom, step complexity

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2022.6

Funding *Hagit Attiya*: Supported by the Israel Science Foundation (grants 380/18 and 1425/22).

Faith Ellen: Supported by the Natural Science and Engineering Research Council of Canada (grant RGPIN-2020-04178).

Acknowledgements We thank Sasho Nikolov for useful discussion. We also appreciate the helpful comments of the anonymous reviewers.

1 Introduction

Approximate agreement allows a set of n processes, each starting with an input from a domain, to obtain outputs (in the same domain) that are close to each other and in the convex hull of the inputs. A parameter ϵ represents an upper bound on how close the outputs are. Originally, Dolev, Lynch, Pinter, Stark and Weihl [7] considered the one-dimensional case, where the domain is \mathbb{R} , the *real numbers*, and motivated the problem by clock synchronization and the stabilization of inputs from sensors. More recently, Mendes, Herlihy, Vaidya, and Garg [14, 15, 18] considered *multidimensional* approximate agreement, also called *approximate vector consensus*, where the domain of inputs and outputs is \mathbb{R}^k , for some integer $k \geq 2$. The multidimensional variant was motivated by distributed algorithms for optimization problems.



© Hagit Attiya and Faith Ellen;
licensed under Creative Commons License CC-BY 4.0

26th International Conference on Principles of Distributed Systems (OPODIS 2022).

Editors: Eshcar Hillel, Roberto Palmieri, and Etienne Rivière; Article No. 6; pp. 6:1–6:12

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Wait-free shared-memory algorithms to reach approximate agreement for asynchronous processes that may fail by crashing are evaluated by their (individual) *step complexity*, that is, the maximum number of reads and writes to shared memory a process performs until it obtains an output. There is a wait-free one-dimensional approximate agreement algorithm with $O(\log(S/\epsilon))$ iterations due to Moran [16], where S , the *spread* of the inputs, is the maximum distance between any two inputs. Each iteration requires one *update* and one *scan* of an atomic snapshot object [2]. Using Attiya and Rachman's implementation of a snapshot object from *single-writer* registers [6], this leads to $O(n \log n \log(S/\epsilon))$ step complexity and, using Inoue, Masuzawa, Chen, and Tokura's implementation from *multi-writer* registers [13], this leads to $O(n \log(S/\epsilon))$ step complexity.

A simple argument shows that $\Omega(n)$ is a lower bound on the step complexity of approximate agreement using single-writer registers [4]. The dependence on n can be avoided by using multi-writer registers: An algorithm by Schenk [17] for inputs in \mathbb{R} achieves $O(\log(M/\epsilon))$ step complexity, where M , the *magnitude* of the inputs, is the largest of the absolute values of the inputs.

The spread, S , of a set of inputs is at most twice its magnitude, M , but the magnitude might be significantly larger than the spread. This raises the challenge, addressed in this paper, of obtaining a wait-free approximate agreement algorithm whose step complexity depends on S , but not on n or M . If the same upper and lower bounds on the domain of the input values are known to all processes, then the bounded version of Schenk's algorithm has $O(\log(R/\epsilon))$ step complexity, where R is the difference between these two bounds. However, R can also be much larger than the spread of the inputs.

This paper provides both negative and positive answers. On the negative side, we prove that $\Omega\left(\min\left\{\frac{\log M}{\log \log M}, \frac{\sqrt{\log n}}{\log \log n}\right\}\right)$ is a lower bound on the step complexity of approximate agreement, even using multi-writer registers. Thus, for values of M that are larger than n (actually, even for $M \in 2^{\Omega(\sqrt{\log n})}$), step complexity that depends on n cannot be avoided. This lower bound is proved by reduction from the *conflict detection* problem [3]. We also prove a lower bound of $\frac{1}{2} \log_{\sqrt{2}+1}(S/\epsilon)$ on the step complexity of multidimensional approximate agreement using multi-writer registers. This extends and improves Herlihy's lower bound for one-dimensional approximate agreement among two or more processes using single-writer registers [11]. Related lower bounds were proved by Attiya, Lynch and Shavit [5] and Hoest and Shavit [12]. Ellen, Gelashvili, and Zhu [8] gave a lower bound on the number of registers needed to solve approximate agreement from a lower bound on its wait-free step complexity.

On the positive side, we prove that a polylogarithmic dependence on n can be achieved, by presenting a multidimensional approximate agreement algorithm with $O(\log n(\log n + \log(S/\epsilon)))$ step complexity. The algorithm repeatedly solves *two group approximate agreement*: combining one group of processes whose values are close together with another such group to form a larger group whose values can be slightly further apart. Two group approximate agreement is solved with a variant of approximate agreement, where processes may know slightly different domains for their inputs. A key step is showing how to solve this subproblem using Schenk's approximate agreement algorithm for inputs in $[0,1]$.

In the one-dimensional case, we can run Schenk's algorithm in parallel with our new algorithm to achieve the best of both algorithms: an approximate agreement algorithm whose step complexity is $O(\min\{\log n(\log n + \log(S/\epsilon)), \log(M/\epsilon)\})$.

The one-dimensional problem was initially studied in message-passing systems [7]. Many different approximate agreement algorithms were subsequently developed for these models. Most of these algorithms are asynchronous and tolerate Byzantine failures. A good example is an approximate agreement algorithm that tolerates f Byzantine failures, when $n > 3f$ [1].

This algorithm works in $O(\log(S/\epsilon))$ (asynchronous) rounds. In each round, each process obtains information from $n - f$ processes. In the shared-memory model, this would translate into an algorithm with $O(n \log(S/\epsilon))$ step complexity.

Multidimensional approximate agreement was previously studied only in message-passing systems. The first algorithm for this problem in \mathbb{R}^d uses $O(d \log(dS/\epsilon))$ rounds [15]. Later, dependency on d was eliminated: there is an algorithm by Függer and Nowak [9] that uses $O(\log(S/\epsilon))$ rounds. As with one-dimensional approximate agreement, this would translate into $O(n \log(S/\epsilon))$ step complexity. Results of Függer, Nowak and Schwarz [10] imply a lower bound of $\Omega(\log(S/\epsilon))$ on the number of rounds for solving multidimensional approximate agreement among two processes. Note that, like all prior lower bounds, their lower bound does not depend on the magnitude of the inputs, but only on their spread.

2 Model

We consider a system where n deterministic asynchronous processes, p_0, \dots, p_{n-1} , communicate by reading and writing to shared multi-reader, multi-writer registers. In this model, all processes can read from and write to all registers. A *configuration* consists of the state of every process and the value of every shared register. In an *initial configuration*, each process starts in an initial state, which includes its input value, and all registers contain an initial value. Each process can be modelled as a deterministic state machine, specifying an algorithm that the process follows until it outputs a value.

Two configurations C and C' are *indistinguishable to process p_i* if p_i has the same state and all shared registers have the same values in both configurations.

A process p_i is *active in configuration C* if process p_i has not yet output a value. In this case, configuration Cp_i is the configuration reached from C when p_i performs the next step of its algorithm.

A *schedule* is a (finite or infinite) sequence of processes, specifying the order in which processes take steps. A non-empty schedule $\sigma = p_{i_1}p_{i_2}\dots$ is *applicable* to a configuration C if process p_{i_1} is active in C and, for every prefix $p_{i_1}\dots p_{i_k}$ of σ of length $k \geq 2$, process p_{i_k} is active in configuration $C_{k-1} = Cp_{i_1}\dots p_{i_{k-1}}$. Suppose σ is a schedule that is applicable to configuration C . If σ has length k , then $C, p_{i_1}, C_1, \dots, p_{i_k}, C_k$ is the *execution from C induced by σ* , where $C_k = Cp_{i_1}\dots p_{i_k}$. If σ is an infinite schedule, then C, p_{i_1}, C_1, \dots is the *execution from C induced by σ* . The *solo execution of process p_i from C* is the execution induced by the longest schedule containing only process p_i that is applicable to C .

We assume each process p_i starts with an input value $x_i \in \mathbb{R}^d$ and, after performing some number of steps according to its algorithm, produces an output value $y_i \in \mathbb{R}^d$. An algorithm (or, more precisely, an algorithm for each process) *solves multidimensional approximate agreement with parameter ϵ* if (a) the distance between output values is at most ϵ , and (b) the output values are in the convex hull of the input values. An algorithm is *wait-free* if it ensures that every process that does not crash terminates within a finite number of its own steps.

The (individual) *step complexity* of an algorithm is the maximum number of steps taken by any one process in any possible execution of the algorithm.

3 A Multidimensional Approximate Agreement Algorithm

In our multidimensional approximate agreement algorithm, described in Section 3.4, processes repeatedly solve instances with increasing values of the accuracy parameter among increasingly

many processes, using their output from one instance as their input to the next instance. The algorithm begins by solving instances among pairs of processes. This produces groups of (two) processes whose inputs are close to one another. These groups are repeatedly combined, two at a time, in a tree-like manner, creating larger groups whose inputs can be slightly further apart.

The subproblem of combining two groups (where, in each group, the inputs are close together) is a restricted version of multidimensional approximate agreement, which we call *two group approximate agreement*. In Section 3.3, we show how to solve two group approximate agreement using another variant of multidimensional approximate agreement, which we call *approximate agreement with domain uncertainty*. This problem is defined in Section 3.2. There, we reduce it to a variant of approximate agreement for inputs in $[0, 1]$, which was introduced and efficiently solved by Schenk [17] and is described in Section 3.1.

3.1 Schenk's Algorithm

Schenk's wait-free approximate agreement algorithm $\text{r-agree}(x, \epsilon)$ [17] assumes that each process p_i has an input $x_i \in [0, 1]$ and all processes have a common accuracy parameter $\epsilon > 0$. It ensures that each non-faulty process outputs a value y_i such that $\min\{x_1, \dots, x_n\} \leq y_i \leq \max\{x_1, \dots, x_n\}$ and all outputs are within distance ϵ of one another.

When $\epsilon = 1/2$, r-agree uses two single-bit multi-writer registers, which are both initially 0. Processes with inputs in the interval $[0, 1/2]$ write 1 to one of these registers and processes with inputs in the interval $[1/2, 1]$ write 1 to the other register. Processes with input $1/2$ output $1/2$. A process with any other input reads the register it didn't write to and, if it sees 1, it also outputs $1/2$. Otherwise it outputs its input. All outputs will lie in the interval of size $1/2$ corresponding to the register that is written to first.

When $\epsilon = 1/4$, the same approach can be applied to this interval, using the outputs as inputs, to obtain new outputs in a subinterval of size $1/4$. The only difficulty is that processes with output $1/2$, which is on the boundary between both intervals $[0, 1/2]$ and $[1/2, 1]$, don't know which interval this is. The solution is for these processes to participate in the subproblems for both these intervals. In at least one of these two subproblems, it will output $1/2$, so it can use its output from the other subproblem. When $\epsilon = 1/2^k$, where $k > 1$, this is done k times, each time reducing the size of the interval containing the inputs by a factor of 2. More generally, this is done $\lceil \log_2(1/\epsilon) \rceil$ times.

A useful generalization of approximate agreement is to allow each process p_i to have a different value $\epsilon_i > 0$ for its accuracy parameter. For this problem, which is called *ϵ -unknown approximate agreement*, all non-faulty processes must output a value within distance $\max\{\epsilon_1, \dots, \epsilon_n\}$ of each other. As in approximate agreement, each output must lie between the smallest and largest inputs. This problem can also be solved using r-agree with $O(\log(\max\{1/\epsilon_1, \dots, 1/\epsilon_n\}))$ step complexity.

Schenk used ϵ -unknown approximate agreement to solve approximate agreement for any real valued inputs x_1, \dots, x_n with $O(\log(\max\{|x_1|, \dots, |x_n|\}/\epsilon))$ step complexity: First, each process p_i finds a value r_i such that (1) the interval $[-r_i, r_i]$ contains at least one of the original inputs and (2) these values differ from one another by at most a factor of 2. Then the processes solve this bounded version of approximate agreement by mapping their inputs to $[0, 1]$ and solving approximate agreement within this interval using accuracy parameters that can differ by at most a factor of 2.

3.2 Approximate Agreement with Domain Uncertainty

We consider a closely related problem, *approximate agreement with domain uncertainty*. In this problem, each process p_i has two points $u_i, v_i \in \mathbb{R}^k$ and a point $x_i \in \mathbb{R}^k$ on the line segment between them, expressed as $x_i = u_i + t_i(v_i - u_i)$, where $t_i \in [0, 1]$. We call this line segment the *domain* for process p_i . The domains of all processes are assumed to be close to one another. Specifically, there exists a constant $\delta > 0$ known to all processes such that $\|u_i - u_j\| \leq \delta$ and $\|v_i - v_j\| \leq \delta$ for all processes p_i and p_j . Each process p_i that does not crash must produce an output $y_i = u_i + t'_i(v_i - u_i)$ on the line between u_i and v_i such that $\min\{t_1, \dots, t_n\} \leq t'_i \leq \max\{t_1, \dots, t_n\}$ and the difference between any two outputs is at most ϵ , which is known to all processes.

Algorithm 1 solves approximate agreement with domain uncertainty for $\epsilon \geq 5\delta$. If the size of the domain of process p_i is small, then it simply sets $y_i = x_i$. Otherwise, it uses r-agree to solve approximate agreement with input t_i and uses the result t'_i to determine its output $y_i = u_i + t'_i(v_i - u_i)$.

■ **Algorithm 1** Code for a process with inputs $u, v \in \mathbb{R}^k$, $t \in [0, 1]$, and parameters ϵ and δ .

```

ApproxAgreeDU( $u, v, t, \epsilon, \delta$ )
1:  $s \leftarrow \|v - u\|$ 
2: if  $s \leq 2\delta$  then return  $u + t(v - u)$ 
3:  $\epsilon' \leftarrow \epsilon/5s$ 
4:  $t' \leftarrow \text{r-agree}(t, \epsilon')$ 
5: return  $u + t'(v - u)$ 

```

Consider an execution where process p_i calls $\text{ApproxAgreeDU}(u_i, v_i, t_i, \epsilon, \delta)$ with $t_i \in [0, 1]$ for $1 \leq i \leq n$. If p_i outputs $y_i = u_i + t_i(v_i - u_i) \in \mathbb{R}^k$ on line 2, then t_i lies between $\min\{t_1, \dots, t_n\}$ and $\max\{t_1, \dots, t_n\}$ and y_i is a point on the line segment between u_i and v_i . If p_i outputs the point $y_i = u_i + t'_i(v_i - u_i) \in \mathbb{R}^k$ on line 5, then t'_i is the value output by r-agree on line 4. The specifications of ϵ -unknown approximate agreement ensure that $0 \leq \min\{t_1, \dots, t_n\} \leq t'_i \leq \max\{t_1, \dots, t_n\} \leq 1$, so y_i is a point on the line segment between u_i and v_i .

To prove that ApproxAgreeDU is correct, it remains to show that all outputs are within ϵ of one another.

► **Lemma 1.** For $1 \leq i, j \leq n$, if process p_i outputs $y_i = u_i + t'_i(v_i - u_i)$ and process p_j outputs $y_j = u_j + t'_j(v_j - u_j)$, then $\|y_i - y_j\| \leq \epsilon$.

Proof. For $1 \leq i \leq n$, let $s_i = \|u_i - v_i\|$ be the size of the domain of process p_i . Since $\|u_i - u_j\| \leq \delta$ and $\|v_i - v_j\| \leq \delta$, it follows from the triangle inequality that $s_i = \|u_i - v_i\| = \|u_i - u_j + u_j - v_j + v_j - v_i\| \leq \|u_i - u_j\| + \|u_j - v_j\| + \|v_j - v_i\| \leq 2\delta + s_j$. By the triangle inequality,

$$\begin{aligned}
\|y_i - y_j\| &= \|u_i + t'_i(v_i - u_i) - u_j - t'_j(v_j - u_j)\| \\
&= \|u_i - u_j + t'_i v_i - t'_i u_j - t'_j v_j + t'_j u_j - t'_i u_j + t'_i u_j - t'_i u_i\| \\
&\leq \|u_i - u_j\| + t'_i \cdot \|v_i - v_j\| + |t'_i - t'_j| \cdot \|v_j - u_j\| + t'_i \cdot \|u_j - u_i\| \\
&\leq \delta + 1 \cdot \delta + |t'_i - t'_j| \cdot \|v_j - u_j\| + 1 \cdot \delta \\
&= 3\delta + |t'_i - t'_j| \cdot s_j.
\end{aligned}$$

Let $I \subseteq \{1, \dots, n\}$ be the set of identifiers of processes that perform line 4 and let $s' = \min\{s_m \mid m \in I\}$. Note that, by the test on line 2, if $I \neq \emptyset$, then $s' > 2\delta$. Hence $s_m \leq s' + 2\delta < 2s'$ for all $m \in I$. For each $m \in I$, let $\epsilon'_m = \epsilon/5s_m$, so $\max\{\epsilon'_m \mid m \in I\} = \max\{\epsilon/5s_m \mid m \in I\} = \epsilon/5 \min\{s_m \mid m \in I\} = \epsilon/5s'$.

First consider the case when $i, j \in I$. From the specifications of ϵ -unknown approximate agreement, $|t'_i - t'_j| \leq \max\{\epsilon'_m \mid m \in I\} = \epsilon/5s'$. Then $\|y_i - y_j\| \leq 3\delta + |t'_i - t'_j| \cdot s_j \leq 3\epsilon/5 + (\epsilon/5s') \cdot 2s' = \epsilon$.

Otherwise, without loss of generality, suppose $j \notin I$. Then $s_j \leq 2\delta$ and $\|y_i - y_j\| \leq 3\delta + |t'_i - t'_j| \cdot s_j \leq 3\delta + 1 \cdot 2\delta = 5\delta = \epsilon$. ◀

If $t_1 = \dots = t_n = t$, then each nonfaulty process p_i outputs $u_i + t(v_i - u_i)$ since $t = \min\{t_1, \dots, t_n\} \leq t'_i \leq \max\{t_1, \dots, t_n\} = t$. In particular, if $t = 0$, then each nonfaulty process outputs its first argument and, if $t = 1$, then each nonfaulty process outputs its second argument.

3.3 Two Group Approximate Agreement

The *two group approximate agreement problem* is a restricted version of the approximate agreement problem in which the processes are divided into two groups, 0 and 1, such that, within each group, the inputs of the processes are guaranteed to be points in \mathbb{R}^k that are within distance $\epsilon/5$ of one another. We will use ApproxAgreeDU (Algorithm 1) to solve this problem.

TwoGroupApproxAgree (Algorithm 2) uses two arrays of multi-writer registers $A[0..1]$ and $B[0..1]$, each with two components. The components of A are initially \perp and can store any point in \mathbb{R}^k . The components of B are single bits and are initially 0. Only processes in group g write to component g of these arrays.

As in Schenk's approximate agreement algorithm, each process writes to one register ($A[g]$, where g is the group to which it belongs) and reads from the other register ($A[1-g]$). If a process in group g sees that $A[1-g]$ has not yet been written to, it informs the processes in group $1-g$ of this fact by writing 1 into $B[g]$ and reads from $A[1-g]$ again. If it sees that $A[1-g]$ has still not been written to, the process outputs its input.

Otherwise, the process participates in an instance of ApproxAgreeDU, using its input as one endpoint of its domain and the point it read as the other endpoint. The endpoints are ordered so that an input from group 0 is the first endpoint and an input from group 1 is the second point. Then the preconditions ensure that the domains of all processes are close to one another.

If a process in group g saw that $A[1-g]$ was first written to between its first and second reads, it uses its input for two group approximate agreement as its input point in this domain. However, if the process saw that $A[1-g]$ had been written to before its first read, it checks the bit $B[1-g]$. If it is 0, processes in the other group will participate in the instance of ApproxAgreeDU and the process also uses its input for two group approximate agreement as its input point in this domain. If it is 1, some processes in the other group may simply output their inputs. In this case, the process uses the other endpoint of its domain as its input point.

Consider an execution where TwoGroupApproxAgree(g_i, x_i, ϵ) is called by process p_i in group $g_i \in \{0, 1\}$, for $1 \leq i \leq n$. Furthermore, suppose $\|x_i - x_j\| \leq \epsilon/5$ for every pair of processes p_i and p_j that are in the same group.

■ **Algorithm 2** Code for a process in group G_g with input x .

```

TwoGroupApproxAgree( $g, x, \epsilon$ )
1:  $a[g] \leftarrow x$ 
2:  $A[g] \leftarrow \text{write}(a[g])$ 
3:  $a[1-g] \leftarrow \text{read}(A[1-g])$ 
4: if  $a[1-g] = \perp$  then
5:    $B[g] \leftarrow \text{write}(1)$ 
6:    $a[1-g] \leftarrow \text{read}(A[1-g])$ 
7:   if  $a[1-g] = \perp$  then
8:     return  $x$ 
9:   else
10:    return  $\text{ApproxAgreeDU}(a[0], a[1], g, \epsilon, \epsilon/5)$ 
11: else
12:    $b \leftarrow \text{read}(B[1-g])$ 
13:   if  $b = 0$  then
14:     return  $\text{ApproxAgreeDU}(a[0], a[1], g, \epsilon, \epsilon/5)$ 
15:   else
16:     return  $\text{ApproxAgreeDU}(a[0], a[1], 1-g, \epsilon, \epsilon/5)$ 

```

► **Observation 2.** *The value 1 is written to most one component of B .*

Proof. Suppose the first step of the execution is by a process in group $1-g$. Then when any process from group g performs line 3, it does not see \perp in $A[1-g]$ and, hence, it does not write 1 to $B[g]$ on line 5. ◀

Next, we show that the outputs are within the convex hull of the inputs.

► **Lemma 3.** *If process p_i outputs y_i , then y_i is in the convex hull of $\{x_1, \dots, x_n\}$.*

Proof. If process p_i returns $y_i = x_i$ on line 8, the claim is true since x_i is in the convex hull of $\{x_1, \dots, x_n\}$. Otherwise, y_i is the point returned by $\text{ApproxAgreeDU}(a_i[0], a_i[1], t_i, \epsilon, \epsilon/5)$, where $t_i \in \{0, 1\}$, g_i is the group to which p_i belongs, $a_i[g_i] = x_i$, and $a_i[1-g_i] \neq \perp$ is the value it read from $A[1-g_i]$ on line 3 or 6.

The only points written to $A[1-g_i]$ are elements of $\{x_1, \dots, x_n\}$, so $a_i[1-g_i] \in \{x_1, \dots, x_n\}$. From the specifications of ApproxAgreeDU , y_i is on the line segment between $a_i[0]$ and $a_i[1]$. Hence y_i is in the convex hull of $\{x_1, \dots, x_n\}$. ◀

Finally, we show that all the outputs are sufficiently close to one another.

► **Lemma 4.** *Suppose that the inputs to all processes in group g are within $\epsilon/5$ of one another, for all $g \in \{0, 1\}$. For $1 \leq i, j \leq n$, if process p_i outputs the point y_i and process p_j outputs the point y_j , then $\|y_i - y_j\| \leq \epsilon$.*

Proof. First consider the processes that return on lines 10, 14, or 16. Each such process p_i returns the result from $\text{ApproxAgreeDU}(a_i[0], a_i[1], t_i, \epsilon, \epsilon/5)$, where $t_i \in \{0, 1\}$. Note that $a_i[g_i]$ is the input of a process in group g_i and only processes in group $1-g_i$ write to $A[1-g_i]$. Hence $a_i[0]$ is the input of a process in group 0 and $a_i[1]$ is the input of a process in group 1. Since the inputs of all processes in the same group are within $\epsilon/5$ of one another, Lemma 1 implies that all these output points differ from one another by at most ϵ .

Now suppose that some process in group g returns its input on line 8. Since this process returns on line 8 only after writing 1 to $B[g]$, Observation 2 implies that no process writes 1 to $B[g-1]$. This implies that no processes in group $1-g$ return on line 8.

Let C be the configuration immediately following the first write to $B[g]$. Any process that returns on line 8 read \perp from $A[1-g]$ on line 6 following its write to $B[g]$. Thus, the first write to $A[1-g]$ occurs after configuration C . Let C' be the configuration immediately following the first write to $A[1-g]$. Note that the first step of every process in group g is a write to $A[g]$, so $A[g] \neq \perp$ in configuration C and all subsequent configurations. Thus, each process p_j in group $1-g$ reads $a_j[g] \neq \perp$ from $A[g]$ on line 3 and reads 1 from $B[g]$ on line 12. Hence, it will call $\text{ApproxAgreeDU}(a_j[0], a_j[1], g, \epsilon, \epsilon/5)$ on line 16 with $a_j[1-g] = x_j$.

Each process p_i in group g that returns on line 10 performs its read of $A[1-g]$ on line 3 prior to C' and its read on line 6 after C' . Each process p_i in group g that returns on line 14 performs its read of $A[1-g]$ on line 3 after C' . In either case, it will call $\text{ApproxAgreeDU}(a_i[0], a_i[1], g, \epsilon, \epsilon/5)$ with $a_i[g] = x_i$.

Hence, in all calls to ApproxAgreeDU , the first argument is an input of a process in group 0, the second argument is an input of a process in group 1, and the third argument is $g \in \{0, 1\}$. Every process p_i that returns on line 10, 14, or 16 outputs $a_i[0] + g(a_i[1] - a_i[0]) = a_i[g]$, which is an input of a process in group g . If process p_i returns on line 8, it is in group g and it returns its own input. By assumption, the inputs of all processes in group g differ from one another by at most $\epsilon/5$. Hence, all outputs differ from one another by at most $\epsilon/5 < \epsilon$. \blacktriangleleft

Note that, if there are only two processes, then $\text{TwoGroupApproxAgree}$ can be used to solve approximate agreement with $O(\log(S/\epsilon))$ step complexity using 1-bit multiwriter registers plus two single-writer registers to which the processes write their inputs.

3.4 Putting the Pieces Together

We can construct an algorithm for approximate agreement in \mathbb{R}^k , where the accuracy parameter ϵ is known by all processes and each process has no information about the inputs of the other processes. The step complexity of our algorithm depends on $(\log n$ and) the spread, the maximum distance between any two inputs, rather than the input with the largest magnitude, as in Schenk's algorithm. The idea is to use a binary tree of height $\lceil \log_2 n \rceil$, with one leaf for each process and with a separate instance of two group approximate agreement at every other node. The accuracy parameter is ϵ at the root and $\epsilon/5^d$ at internal nodes of depth d . Each process p_i traverses a path from its leaf to the root, using its input x_i as its input to the first instance of two group approximate agreement and, for each subsequent instance, using its output from the previous instance as its input. If its leaf is in the left subtree of a node, a process will be in group 0 of the instance of two group approximate agreement at the node and, if its leaf is in the right subtree, it will be in group 1.

We show that the input requirements for each instance of two group approximate agreement is satisfied.

► **Lemma 5.** *For $0 \leq d < \lceil \log_2 n \rceil$, for each instance of two group approximate agreement at each node of depth d and in each group, the inputs of the processes are within distance $\epsilon/5^{d+1}$ of one another.*

Proof. First, consider any node with a leaf as a child. Since the group corresponding to that child consists of only one process, the inputs of the processes in this group are all equal to one another and, hence, within distance 0 of one another.

Now consider any node which has at least one child that is not a leaf and assume the claim is true for those children. Let d be the depth of the node, so its children are at depth $d+1$. For each child that is not a leaf and for each group of that child, the inputs in the group are within distance $\epsilon/5^{d+2}$ of one another. By Lemma 4, the outputs of the instance

at this child are within distance $\epsilon/5^{d+1}$ of one another. Hence the inputs to the instance at the node from the group corresponding to this child are within distance $\epsilon/5^{d+1}$ of one another. ◀

► **Theorem 6.** *The algorithm described above solves ϵ -unknown approximate agreement with $O(\log n(\log n + \log(S/\epsilon)))$ step complexity among n processes, where S is the spread of the inputs.*

Proof. By Lemma 5, for the instance of two group approximate agreement at the root, the inputs in each group are within distance $\epsilon/5$ of one another. By Lemma 4, the outputs of the instance at the root and, hence, the algorithm are within distance ϵ of one another.

By Lemma 3, the outputs of any instance of two group approximate agreement are in the convex hull of its inputs. Thus, it follows by induction that the outputs of the algorithm are in the convex hull of $\{x_1, \dots, x_n\}$.

Each process participates in at most one instance of two group approximate agreement at depth d for $0 \leq d < \lceil \log_2 n \rceil$. Since the inputs to this instance are all within the convex hull of $\{x_1, \dots, x_n\}$, they are within distance S of one another. At level d , the accuracy parameter for TwoGroupApproxAgree is $\epsilon/5^d$. It involves at most one call to ApproxAgreeDU with accuracy parameter $\epsilon/5^d$, which, in turn, involves at most one call to r-agree with accuracy parameter at least $\epsilon/5^{d+1}S$. Each such call to r-agree takes $O(\log(S5^{d+1}/\epsilon)) = O(d + \log(S/\epsilon))$ steps. Thus, the total step complexity is $O(\log n(\log n + \log(S/\epsilon)))$. ◀

The step complexity of Schenk's algorithm for domain \mathbb{R} depends only on the magnitude of the inputs, whereas the step complexity of our algorithm depends on the spread of the inputs and the number of processes. To get the best of both worlds with domain \mathbb{R} , one can run Schenk's algorithm and our algorithm in parallel, both with accuracy parameter $\epsilon/5$. Specifically, a separate part of shared memory is used for each algorithm and each process alternately performs steps of the two algorithms. If a process first completes Schenk's algorithm with output y , then it performs TwoGroupApproxAgree(0, y , ϵ). If it first completes our algorithm with output y , then it performs TwoGroupApproxAgree(1, y , ϵ). In either case, it returns the output it obtains from TwoGroupApproxAgree.

Note that, from the output specifications of Schenk's algorithm and our algorithm, in each group, all processes have inputs that are within distance $\epsilon/5$ of one another. Hence, from the output specifications of TwoGroupApproxAgree, all outputs will be within distance ϵ of one another. Since all three algorithms satisfy validity, the resulting algorithm also satisfies validity.

4 Lower bound on the Step Complexity as a Function of the Magnitude and the Number of Processes

In the *conflict detection* problem, each process p_i has an input $x_i \in \{1, \dots, m\}$. If a process doesn't crash, it must output either **true** or **false**. If two processes have different input values and neither crashes, at least one of them returns **true**, indicating that there is a conflict. If all processes have the same input value, they must all output **false**, indicating no conflict. Aspnes and Ellen [3] proved that the step complexity of this problem when implemented using only registers is $\Omega\left(\min\left\{\frac{\log m}{\log \log m}, \frac{\sqrt{\log n}}{\log \log n}\right\}\right)$, where n is the number of processes.

There is a simple reduction from conflict detection to approximate agreement, where each process has an input in $\{1, \dots, m\}$ and $\epsilon = 1/2$. Specifically, given inputs $x_i \in \{1, \dots, m\}$, the processes perform approximate agreement to determine outputs y_i . If $y_i = x_i$, then process

6:10 The Step Complexity of Multidimensional Approximate Agreement

p_i outputs **false** and, if $y_i \neq x_i$, then process p_i outputs **true**. If all the inputs have the same value, then, by validity, all the y_i 's have this value and all processes output **false**, as required. However, if $x_i \neq x_j$, then either $y_i \neq x_i$ or $y_j \neq x_j$, since $|y_i - y_j| \leq \epsilon < 1 \leq |x_i - x_j|$. In this case, either p_i outputs **true** or p_j outputs **true**, as required. It follows that the step complexity of approximate agreement among n processes with inputs in $\{1, \dots, m\}$ and $\epsilon = 1/2$ is $\Omega\left(\min\left\{\frac{\log m}{\log \log m}, \frac{\sqrt{\log n}}{\log \log n}\right\}\right)$.

5 Lower Bound on the Step Complexity as a Function of the Spread

Consider a wait-free algorithm for approximate agreement. For any reachable configuration C and any process p_i active in C , let $m_i(C)$ denote the value that process p_i outputs in its solo execution starting from configuration C . If p_i has already decided in configuration C , then $m_i(C)$ denotes the value that p_i decided.

► **Observation 7.** *If p_i is active in C , then $m_i(C) = m_i(Cp_i)$.*

► **Observation 8.** *If C and C' are indistinguishable to process p_i , then $m_i(C) = m_i(C')$.*

Herlihy [11] proved a lower bound for approximate agreement among 2 or more processes that communicate using single-writer registers. We present a corrected version of his proof, together with an extension to multi-writer registers. We begin with a technical lemma.

► **Lemma 9.** *If processes p_0 and p_1 are active in configuration C , then there exists $\sigma \in \{p_0, p_1, p_0p_1, p_1p_0\}$ such that $\|m_0(C\sigma) - m_1(C\sigma)\| \geq (\sqrt{2} - 1)^{|\sigma|} \|m_0(C) - m_1(C)\|$.*

Proof. We consider different cases depending on the operations p_0 and p_1 are poised to perform.

If p_0 is poised to perform a read, then configurations C and Cp_0 are indistinguishable to p_1 , so, by Observation 8, $m_1(Cp_0) = m_1(C)$. By Observation 7, $m_0(Cp_0) = m_0(C)$. Thus $\|m_0(Cp_0) - m_1(Cp_0)\| = \|m_0(C) - m_1(C)\| \geq (\sqrt{2} - 1)^1 \|m_0(C) - m_1(C)\|$.

Similarly, if p_1 is poised to perform a read, then $\|m_0(Cp_1) - m_1(Cp_1)\| = \|m_0(C) - m_1(C)\| \geq (\sqrt{2} - 1)^1 \|m_0(C) - m_1(C)\|$.

If p_0 and p_1 are poised to perform writes to the same location, then Cp_0p_1 and Cp_1 are indistinguishable to process p_1 , so Observation 8 implies that $m_1(Cp_0p_1) = m_1(Cp_1)$. By Observation 7, $m_1(Cp_0p_1) = m_1(Cp_0)$ and $m_1(Cp_1) = m_1(C)$. Thus $m_1(Cp_0) = m_1(C)$. However, by Observation 7, $m_0(Cp_0) = m_0(C)$. Hence, we get $\|m_0(Cp_0) - m_1(Cp_0)\| = \|m_0(C) - m_1(C)\| \geq (\sqrt{2} - 1)^1 \|m_0(C) - m_1(C)\|$.

If p_0 and p_1 are poised to perform writes to different locations, then $Cp_0p_1 = Cp_1p_0$. By the triangle inequality,

$$\begin{aligned} \|m_0(C) - m_1(C)\| &= \|m_0(C) - m_1(Cp_0) + m_1(Cp_0) - m_0(Cp_1) + m_0(Cp_1) - m_1(C)\| \\ &\leq \|m_0(C) - m_1(Cp_0)\| + \|m_1(Cp_0) - m_0(Cp_1)\| + \|m_0(Cp_1) - m_1(C)\|. \end{aligned}$$

Since $(\sqrt{2} - 1) + (\sqrt{2} - 1) + (\sqrt{2} - 1)^2 = 1$, it follows that either

$$\begin{aligned} \|m_0(C) - m_1(Cp_0)\| &\geq (\sqrt{2} - 1) \|m_0(C) - m_1(C)\|, \\ \|m_0(Cp_1) - m_1(C)\| &\geq (\sqrt{2} - 1) \|m_0(C) - m_1(C)\|, \text{ or} \\ \|m_1(Cp_0) - m_0(Cp_1)\| &\geq (\sqrt{2} - 1)^2 \|m_0(C) - m_1(C)\| \end{aligned}$$

In the first case, $m_0(Cp_0) = m_0(C)$ by Observation 7, so $\|m_0(Cp_0) - m_1(Cp_0)\| \geq (\sqrt{2} - 1)^1 \|m_0(C) - m_1(C)\|$.

Similarly, in the second case, $\|m_0(Cp_1) - m_1(Cp_1)\| \geq (\sqrt{2} - 1)^1 \|m_0(C) - m_1(C)\|$.

In the third case, by Observation 7, $m_0(Cp_1p_0) = m_0(Cp_1)$ and $m_1(Cp_0p_1) = m_1(Cp_0)$. Since $Cp_1p_0 = Cp_0p_1$, it follows that $\|m_0(Cp_1p_0) - m_1(Cp_1p_0)\| = \|m_1(Cp_0) - m_0(Cp_1)\| \geq (\sqrt{2} - 1)^2 \|m_0(C) - m_1(C)\|$. ◀

► **Theorem 10.** *Any approximate agreement algorithm for two or more processes and accuracy ϵ has step complexity at least $\frac{1}{2} \log_{\sqrt{2}+1}(S/\epsilon)$, where S is the spread of the inputs.*

Proof. Consider an initial configuration C_0 in which process p_0 has input x_0 , process p_1 has input x_1 , and $S = \|x_0 - x_1\|$. Suppose an adversary schedules steps of processes p_0 and p_1 by repeatedly choosing schedules that satisfy Lemma 9 until both have output values. Let σ' be the resulting schedule, let $C' = C\sigma'$, and let $t = |\sigma'|$. Then $\|m_0(C') - m_1(C')\| \geq (\sqrt{2} - 1)^t \|m_0(C_0) - m_1(C_0)\|$. To satisfy agreement, $\|m_0(C') - m_1(C')\| \leq \epsilon$. To satisfy validity, $m_0(C_0) = x_0$ and $m_1(C_0) = x_1$, so $\|m_0(C_0) - m_1(C_0)\| = \|x_0 - x_1\| = S$. Hence $t \geq \log_{1/(\sqrt{2}-1)}(S/\epsilon)$. This is equal to $\log_{\sqrt{2}+1}(S/\epsilon)$, since $1/(\sqrt{2} - 1) = \sqrt{2} + 1$. There are only two processes taking steps, so at least one of them must take at least $t/2$ steps. ◀

6 Conclusion

This paper studies wait-free multidimensional approximate agreement in the shared memory model, using only read and write operations. The step complexities of our algorithms have poly-logarithmic dependency on S/ϵ and n , where S is the maximum distance between inputs, ϵ is a parameter bounding the distance between outputs, and n is the number of processes.

There is still a gap between our upper and lower bounds, and it would be interesting to bring them closer together. In particular, it might be possible to increase the lower bound for the conflict detection problem.

Another possible avenue for future research is to explore whether our ideas can be applied in other models, in particular, to obtain approximate agreement algorithms for asynchronous message-passing systems.

References

- 1 Ittai Abraham, Yonatan Amit, and Danny Dolev. Optimal resilience asynchronous approximate agreement. In *Proceedings of the 8th International Conference On Principles Of Distributed Systems (OPODIS)*, pages 229–239, 2004.
- 2 Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890, 1993.
- 3 James Aspnes and Faith Ellen. Tight bounds for adopt-commit objects. *Theory of Computing Systems*, 55(3):451–474, 2014.
- 4 Hagit Attiya and Faith Ellen. *Impossibility Results for Distributed Computing*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2014.
- 5 Hagit Attiya, Nancy Lynch, and Nir Shavit. Are wait-free algorithms fast? *Journal of the ACM*, 41(4):725–763, 1994.
- 6 Hagit Attiya and Ophir Rachman. Atomic snapshots in $o(n \log n)$ operations. *SIAM Journal on Computing*, 27(2):319–340, 1998.
- 7 Danny Dolev, Nancy Lynch, Shlomit Pinter, Eugene Stark, and William Weihl. Reaching approximate agreement in the presence of faults. *Journal of the ACM*, 33(3):499–516, 1986.
- 8 Faith Ellen, Rati Gelashvili, and Leqi Zhu. Revisionist simulations: A new approach to proving space lower bounds. In *Proceedings of the 37th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 61–70, 2018.
- 9 Matthias Függer and Thomas Nowak. Fast multidimensional asymptotic and approximate consensus. In *Proceedings of the 32nd International Symposium on Distributed Computing (DISC)*, pages 27:1–27:16, 2018.

6:12 The Step Complexity of Multidimensional Approximate Agreement

- 10 Matthias Függer, Thomas Nowak, and Manfred Schwarz. Tight bounds for asymptotic and approximate consensus. *Journal of the ACM*, 68(6):46:1–46:35, 2021.
- 11 Maurice Herlihy. Impossibility results for asynchronous PRAM. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 327–336, 1991.
- 12 Gunnar Hoest and Nir Shavit. Toward a topological characterization of asynchronous complexity. *SIAM Journal on Computing*, 36(2):457–497, 2006.
- 13 Michiko Inoue, Toshimitsu Masuzawa, Wei Chen, and Nobuki Tokura. Linear-time snapshot using multi-writer multi-reader registers. In *Proceedings of the 8th International Workshop on Distributed Algorithms (WDAG)*, pages 130–140, 1994.
- 14 Hammurabi Mendes and Maurice Herlihy. Multidimensional approximate agreement in Byzantine asynchronous systems. In *Proceedings of the 45th Annual ACM Symposium on Theory of Computing (STOC)*, pages 391–400, 2013.
- 15 Hammurabi Mendes, Maurice Herlihy, Nitin Vaidya, and Vijay K. Garg. Multidimensional agreement in Byzantine systems. *Distributed Computing*, 28(6):423–441, 2015.
- 16 Shlomo Moran. Using approximate agreement to obtain complete disagreement: The output structure of input-free asynchronous computations. In *Proceedings of the 3rd Israel Symposium on the Theory of Computing and Systems (ISTCS)*, pages 251–257, 1995.
- 17 Eric Schenk. Faster approximate agreement with multi-writer registers. In *Proceedings of the 36th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 714–723, 1995.
- 18 Nitin H. Vaidya and Vijay K. Garg. Byzantine vector consensus in complete graphs. In *Proceedings of the 32nd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 65–73, 2013.

Performance Anomalies in Concurrent Data Structure Microbenchmarks

Rosina F. Kharal ✉ 🏠
University of Waterloo, Canada

Trevor Brown ✉ 🏠
University of Waterloo, Canada

Abstract

Recent decades have witnessed a surge in the development of concurrent data structures with an increasing interest in data structures implementing concurrent sets (CSets). Microbenchmarking tools are frequently utilized to evaluate and compare the performance differences across concurrent data structures. The underlying structure and design of the microbenchmarks themselves can play a hidden but influential role in performance results. However, the impact of microbenchmark design has not been well investigated. In this work, we illustrate instances where concurrent data structure performance results reported by a microbenchmark can vary 10-100x depending on the microbenchmark implementation details. We investigate factors leading to performance variance across three popular microbenchmarks and outline cases in which flawed microbenchmark design can lead to an inversion of performance results between two concurrent data structure implementations. We further derive a set of recommendations for best practices in the design and usage of concurrent data structure microbenchmarks and explore advanced features in the Setbench microbenchmark.

2012 ACM Subject Classification Computing methodologies → Concurrent computing methodologies; Theory of computation → Concurrency; Computing methodologies → Massively parallel and high-performance simulations

Keywords and phrases concurrent microbenchmarks, concurrent data structures, concurrent performance evaluation, PRNGs, parallel computing

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2022.7

Related Version *Full Version*: <https://arxiv.org/abs/2208.08469>

Supplementary Material

Interactive Resource (Website): https://cs.uwaterloo.ca/~t35brown/setbench_example_www/
Software (Repository): https://github.com/rkharal/prng_experiments, archived at `swh:1:dir:3c60f562296ce0d0636dab4ff6e8493553f4c601`

Funding This work was supported by: the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Program grant: RGPIN-2019-04227, the Canada Foundation for Innovation John R. Evans Leaders Fund with equal support from the Ontario Research Fund CFI Leaders Opportunity Fund: 38512, NSERC Discovery Launch Supplement: DGECR-2019-00048, and the University of Waterloo.

Acknowledgements We thank the reviewers for their helpful comments and suggestions.

1 Introduction

The execution efficiency of highly parallelizable data structures for concurrent access has received significant attention over the past decade. An extensive variety of data structures have appeared, with a particular focus on data structures implementing concurrent sets (CSets) [8, 11, 21, 37, 51]. CSets have applications in many areas including distributed systems, database design, and multicore computing. A CSet is an abstract data type (ADT) which stores keys and provides three primary operations on keys: search, insert, and delete. Insert and delete operations modify the CSet and are called *update* operations. There are numerous concurrent data structures that can be used to implement CSets, including trees,



© Rosina F. Kharal and Trevor Brown;

licensed under Creative Commons License CC-BY 4.0

26th International Conference on Principles of Distributed Systems (OPODIS 2022).

Editors: Eshcar Hillel, Roberto Palmieri, and Etienne Rivière; Article No. 7; pp. 7:1–7:24

Leibniz International Proceedings in Informatics

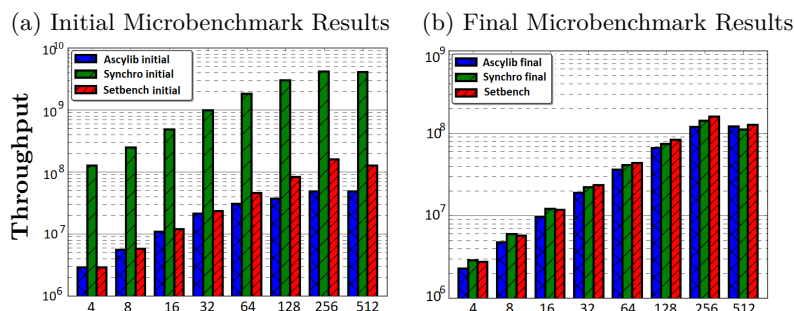


LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

skip-lists, and linked-lists. A CSet data structure refers to the implementation of a CSet. Microbenchmarks are commonly used to evaluate the performance of CSet data structures, essentially performing a stress test on the CSet across varying search/update workloads and thread counts. A typical microbenchmark runs an experimental loop bombarding the CSet with randomized operations performed by threads until the duration of the experiment expires. *Throughput*, number of operations performed by a CSet, is a key performance metric. Multiple platforms for microbenchmarking exist to support CSet research. The accuracy and reliability of performance results generated from microbenchmark experiments is fundamental to concurrent data structure research. Researchers must be able to assess the performance benefit vs loss of varying concurrent implementation strategies and their overall impact on performance. Microbenchmarks are also an important tool for comparative performance analysis between different implementations of CSets. While CSet implementations have been well studied [3, 8, 11, 37], the popular microbenchmarks used to evaluate them have not been scrutinized to the same degree. Microbenchmarking idiosyncrasies exist that can significantly distort performance results. The goal of our work is to better understand the role of microbenchmark design in performance results and attempt to minimize factors present within the microbenchmark that misrepresent performance.

When testing a CSet implementation on three different microbenchmarks with identical parameters, one would expect to observe similar performance results within a reasonable margin of error. However, we found 10-100x performance differences on the same CSet data structure tested across the Ascylib [11], Setbench [7], and Synchrobench [14] microbenchmarks. These microbenchmarks are often employed for evaluation of high performance CSets. In Figure 1(a) we observe a range of varying performance results on the popular lock-free BST by Natarajan et al. [37] across the three microbenchmarks displayed using a logarithmic y-axis in order to capture wide performance gaps on a single scale. We performed a systematic review of the design intricacies within each microbenchmark. We found discrepancies in microbenchmark implementation leading to CSet data structures underperforming in one microbenchmark and over performing in another. We found one popular microbenchmark duplicating the entire benchmark code for each CSet implementation. This renders the code highly prone to errors related to updates or modifications to the benchmark, and may inevitably result in reporting skewed experimental results. Our investigations led to the discovery that seemingly minor differences in the architecture and experimental design of a microbenchmark can cause a 10-100x performance boost, erroneously indicating high performance of the data structure when the underlying cause is the microbenchmark itself. We performed successive updates to two of the microbenchmarks, adjusting where errors or discrepancies were discovered, until performance is approximately equalized (Figure 1(b)). In this work, we discuss the primary factors leading to microbenchmark performance variance and provide a set of recommended best practices for microbenchmark experiment design.

Microbenchmarks rely heavily on pseudo random number generators (PRNGs) to generate randomized keys and/or select randomized operations on a CSet. In this work, in addition to investigating microbenchmark design differences, we delve into a deeper investigation of PRNG usage in microbenchmarks. Deleterious interactions between a PRNG and a microbenchmark that uses it can go undetected for years. We present examples where (mis)use of PRNGs can cause substantial performance anomalies and generate misleading results. We illustrate how using a problematic PRNG can lead to an inversion of throughput results on pairs of CSet data structures. We discuss the pitfalls of common PRNG usage in microbenchmark experiments. Our experiments are limited to concurrent tree data structures that were present in the three microbenchmarks in our study. We believe the lessons learned



■ **Figure 1** Throughput results across three microbenchmarks, Ascylib, Synchrobench and Setbench executing on a 256-core system testing the lock-free BST [37]. Thread count is displayed on the x-axis, y-axis is a logarithmic scale. Figure (a) results from unmodified microbenchmarks (as written by their authors). Figure (b) results for modified versions of Synchrobench and Ascylib correcting for pitfalls in microbenchmark design.

in our investigations related to microbenchmark design apply broadly to the experimental process and are not limited to specific CSet implementations. We leave the investigation of microbenchmark performance on varying CSet implementations for future work.

Contributions. In this work, we perform the first rigorous comparative analysis of three widely used microbenchmarks for CSet performance evaluation. We present an overview of related work in Section 3. The three microbenchmarks evaluated in our work are described in detail in Section 4. We investigate the source of performance differences reported by each microbenchmark when testing equivalent tree-based data structures in Sections 4 and 5. We study the role of memory reclamation and its impact on CSet performance in Section 5. In Section 6 we investigate commonly employed methods of fast random number generation and the pitfalls of each. We describe a set of derived recommendations for best practice in microbenchmark design at the ends of Sections 4, 5 and 6. Additional recommendations for further improvements in microbenchmark design are discussed in Section 7. Advanced features of the Setbench microbenchmark are described in Section 7.2 followed by concluding remarks in Section 8. In the next section, we begin with a background on the principles of microbenchmark design with concrete examples from the Setbench microbenchmark.

2 Background

In this work we test three concurrent synthetic microbenchmarks, Setbench [7], Ascylib [11], and Synchrobench [14] for high speed CSet analysis. The key properties of each microbenchmark are summarized in Table 1. The microbenchmarks report the total operations per second performed on the CSet by n threads based on a specified workload. In particular, we study data structures that implement *sets* of keys and provide operations to *search*, *insert* or *delete* a key. The microbenchmarks allow users to specify parameters that include the number of threads (τ), the experiment duration (d), the update rate (u), and the key range (r) contained within the set (i.e. $[1, 200,000]$).

Evaluating performance operations on an initially empty data structure will generate results that are misleading and not representative of average performance on a non-empty CSet. Therefore, microbenchmarks typically prefill the CSet before the experiment begins to contain a subset of keys less than or equal to the total range. The prefill size may be specified

■ **Table 1** Summary of properties within each microbenchmark. 1: steady state depends on experiment parameters. Setbench allows varying insert and delete ratios; this determines what the data structure will fill to in steady state. (*2: performance tracking. *3: statistics tracking, performance recording, automated graph generation, range query searches, varying distributions of keys/operations are possible, independent insert and delete rates possible. *4: track effective updates, alternating updates possible).

Benchmark Properties	Prefill Size	Threads used to Prefill	Prefill Ops	PRNG for key generation	PRNG for update choice	
Ascylib	half-full	single/n	inserts	✓	✓	
Setbench	steady state ¹	n	ins/dels	✓	✓	
Synchrobench	half-full	single	inserts	✓		
Benchmark Properties	Centralized Test loop	Test file per DS	Range Queries Available	Effective Upd Option	Thread Pinning	Unique Features
Ascylib	✓	✓		✓	✓	*2
Setbench	✓		✓		✓	*3
Synchrobench		✓		✓	✓	*4

by the user as the *initial* (*i*) prefill amount, or the microbenchmark may decide the prefill size using its own algorithm. For a duration *d*, a microbenchmark runs in an experiment loop where *n* threads are assigned keys from the specified key range based on a random uniform distribution, though other distributions are also possible. Threads perform a combination of search or update operations based on experimental parameters. For example, if the specified update rate (*u*) is 10%, the search rate is 90%. The microbenchmark either randomly splits the update rate across insert and delete operations, or employs its own algorithm to attempt to divide insert and delete operations equally. Microbenchmarks may offer the ability to specify independent insert and delete rates. This is discussed further in Section 4.3.

2.1 Microbenchmark Setup

We use the *Setbench* microbenchmark as a case study to explain some underlying design principles in concurrent microbenchmarks. A typical Setbench experiment involves *n* threads accessing a CSet for a fixed duration. During this time, each thread performs search or update operations that are chosen according to a specified probability distribution on keys randomly drawn from another probability distribution over a *fixed* key range. For example, threads might choose an *operation* to perform *uniformly* (1/3rd insert, 1/3rd delete and 1/3rd search operations), and then choose a *key* to insert, delete or search for from a *Zipfian* distribution. Each thread has a PRNG object, and the same object is used to select a random operation and generate random keys.

To ensure that an experiment measures performance as it would in the steady state (after the experiment has been running for a sufficiently long time), performance measurements are not taken until the data structure is warmed up by performing insertions and deletions until the CSet converges to approximately its steady state. This step is called prefilling. If the key range is $[1, 10^6]$, and threads do 50% insertions and 50% deletions, then the size of the CSet in steady state will be approximately 500,000 (half full). Different microbenchmarks will employ varying methods of prefilling the data structure prior to experimental evaluation. This is discussed in the next section. In this work, we evaluate performance results across three microbenchmarks and analyse the underlying subtleties in microbenchmark design which lead to varying performance on equivalent CSet data structures. An example of this is

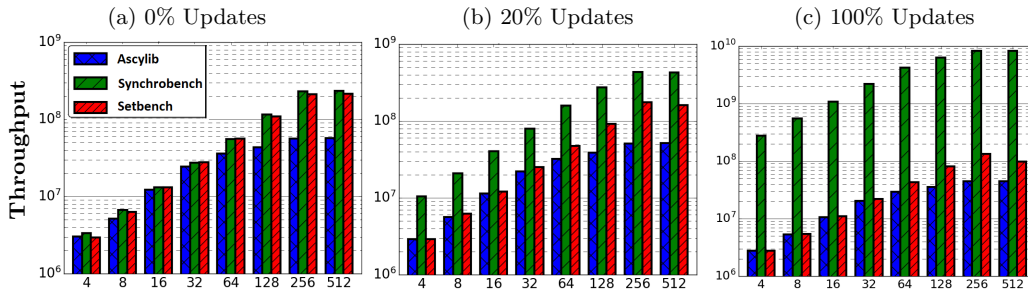
illustrated in Figure 1(a) where microbenchmark experiments are performed on the lock-free BST by Natarajan et al. [37]. The initial comparative throughput results are very different. We apply successive modifications to the microbenchmarks where required in an attempt to minimize large performance gaps. This process is outlined step-by-step in Section 4.

Experiments performed in this work execute on a dual socket, AMD EPYC 7662 processor with 256 logical cores and 256 GB of RAM. DRAM is equally divided across two NUMA nodes. We use a scalable memory allocator, *jemalloc* [13], to prevent memory allocation bottlenecks. Each microbenchmark employs its own PRNG for generating random numbers during the experiment loop. This is discussed further in Section 4. We test key ranges between 2000 and 2 million keys using thread counts of up to 512, which gives an indication of the effects of oversubscribing the cores. All figures in Sections 4 and 5 in this work are displayed using a logarithmic y-axis in order to allow visual comparison between algorithms with large differences.

3 Related Work

There are previous efforts in the literature to better understand the underlying structure and design of benchmarks used to evaluate concurrent applications. In their work on the comparative evaluation of transactional memory (TM) systems, Nguyen et al. discuss the unexpected low performance results observed when using benchmarks to evaluate various hardware transactional memory (HTM) and software transactional memory (STM) systems [38]. They argue that the observed limited performance results are a consequence of the programming model and data structure design used within the benchmarks and are not necessarily indicative of true performance results of the TM systems themselves. In related work by Ruan et al. [41], the STAMP benchmark suite [33] used for evaluating transactional memory was identified as being out-of-date. The authors present several suggested modifications to the benchmark suite to boost the reliability of performance results for more accurate TM evaluation. McSherry et al. discuss the COST (Configuration that Outperforms a Single Thread) [32] associated with scaling applications to support multi-threaded execution, and the need to measure performance gains without rewarding the substantial overhead costs of parallelization.

Recent microbenchmarks exist that were not tested in our work, such as the Synchron framework [21] for concurrent data structures evaluation. We leave this for future study. There has been some prior investigation of microbenchmark design for concurrent data structure performance evaluation. Microbenchmark experiments executing a search-only workload on CSets have been tested in previous work by Arbel et al. [3]. They considered differences in concurrent tree implementations of CSets and their impact on performance. It was discovered that subtle differences in concurrent tree implementations can play a pivotal role in microbenchmark performance results. Our work concentrates on the impact of *microbenchmark implementation differences* on CSet data structure performance for workloads that include updates. Mytkowicz et al. in their work, “*Producing Wrong Data Without Doing Anything Obviously Wrong!*” [34], illustrate how subtle changes to an experiment’s setup can lead to enormous performance differences and ultimately to incorrect conclusions. Tim Harris’ presentation, “*Benchmarking Concurrent Data Structures*” [17], is closely related to our work. Harris explains the need for sound experimental methodology in performance evaluation tools and discusses some noted pitfalls in the Synchron benchmark in [18]. Important considerations in the design of good concurrent data structure experiments have been previously discussed in seminars presented by Trevor Brown [6]. Brown discusses



■ **Figure 2** Initial throughput comparisons across three unmodified microbenchmarks testing the lock-free BST [37] on varying update rates.

subtle aspects of microbenchmark testing configurations and underlying memory and thread distributions that can play a crucial role in performance results. This is discussed further in Section 7.1. In our work, we provide an investigative approach to microbenchmark design by comparing design strategies employed in three popular microbenchmarks.

4 Comparison Lock-Free Binary Search Tree

As mentioned above, a key performance indicator in the evaluation of CSet data structure performance is the total number of operations per second (throughput). This is computed by summing the total number of operations performed per thread and dividing by the duration of the experiment. A key indicator of memory reclamation efficiency is the *maximum resident memory* occupied in RAM by the microbenchmark program during the duration of the experiment. We use this measure to evaluate the memory reclamation capabilities of each microbenchmark in Section 5. We perform a comparative study on the lock-free BST data structure by Natarajan et al. [37] which implements a CSet. The lock-free BST stores keys in leaf nodes; internal nodes contain repeated leaf values to provide direction for searches. Not all microbenchmarks implement the lock-free BST with memory reclamation. Therefore, our initial comparisons turn memory reclamation off. Table 1 describes the properties of each microbenchmark tested in this work.

4.1 Synchronbench

The Synchronbench synthetic microbenchmark allows the evaluation of popular C++ and Java-based CSet implementations. Synchronbench is a popular microbenchmark used for performance evaluation of CSet data structures [5, 11, 14, 16, 47, 48, 50]. Synchronbench allows users to specify an *alternate option* (-A) or an *effective option* (-f) as input parameters to the microbenchmark. The -A option can be used to force threads to alternate between a key being inserted and the same key being deleted. The -f option sets total throughput calculations to count failed update operations as search operations and not as update operations. We do not use either of these options in our experiments. Synchronbench performs single threaded prefilling with insert-only operations. Each data structure directory contains a test file (`test.c`) that runs the basic test loop of the microbenchmark, performing a timed search/update workload on the CSet. In our evaluation of Synchronbench, we found the repetition of the `test.c` file in each data structure directory. This is discussed further in Section 4.5.1. Synchronbench allows users to specify a single update rate that is divided between insert and delete operations, though the division is not necessarily equal. This is discussed further in Section 4.5.4.

■ **Table 2** Summary of Synchrobench and Ascylib modifications tested in this work. Ascylib required fewer changes, Setbench did not require any modifications for the comparative experiments performed in Section 4. The original installed implementations are labelled Synchro and Ascylib without a version number. Ascylib' and Synchro' are versions of each microbenchmark where only the lock-free BST implementation is modified (imported from Setbench).

Synchro Version	Synchro	Synchro1	Synchro2	Synchro3	Synchro4	Synchro5	Synchro'
insert & delete		✓	✓	✓	✓	✓	
random seeds/thread			✓	✓	✓	✓	
MM3 RNG				✓	✓	✓	
randomized updates					✓	✓	
common DS impl						✓	✓
Ascylib Version	Ascylib	Ascylib1	Ascylib2	Ascylib3	Ascylib'		
disable thread pin		✓	✓	✓			
MM3 RNG			✓	✓			
common DS impl				✓	✓		

4.2 Ascylib

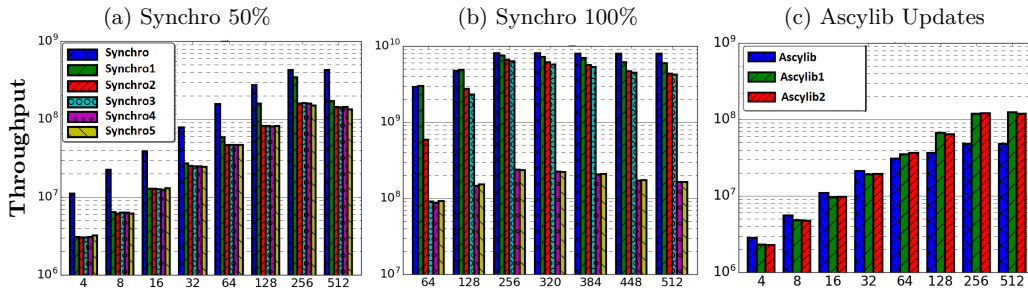
The Ascylib synthetic microbenchmark is another microbenchmark used to compare performance of concurrent data structures [3, 11, 12, 14, 23, 40, 54]. Ascylib also performs an initial prefilling step using single threaded insert-only operations. However, Ascylib has a setting to allow multi-threaded prefilling using insert-only operations. The range and initial values are updated to the closest power of two. This is a necessary condition for the Ascylib test algorithm to generate randomly distributed keys. The experiment testing algorithm (*test_simple.c*) is also repeated in each data structure directory. However, the main test loop is implemented in one common macro and is shared across each CSet data structure implemented in Ascylib. The update rate is randomly distributed among insert and delete operations and updates are not required to be effective. Ascylib allows additional user inputs to define profiling parameters which are not tested in this work. Additional properties of Ascylib can be seen in Table 1.

4.3 Setbench

The Setbench synthetic microbenchmark is another benchmarking tool employed in concurrent data structure literature [3, 7, 8, 9, 24, 42]. Setbench employs a directory structure per CSet implementation. However, each CSet utilizes a single experiment test loop via an adapter class which imports each specific CSet implementation into the main experimental algorithm. This allows a single point of update for the testing algorithm and avoids software update errors. Setbench allows specification of independent insert and delete rates. Setbench uses per thread PRNGs initialized with unique seeds. Although Setbench has multiple choices of PRNGs, we employed the *murmurhash3* (MM3) [1] PRNG for comparative microbenchmark experiments in this section of our work. Setbench employs multi-threaded prefilling using randomized insert and delete operations. The benefits of this are discussed in Section 4.8. We delve into further details regarding the Setbench microbenchmark in Section 7.2.

4.4 Throughput Comparisons

We test the initial installed implementations of the three aforementioned microbenchmarks in order to compare performance results on the lock-free BST data structure. To standardize experiments across the three microbenchmarks, we performed single threaded prefilling



■ **Figure 3** Throughput results for successive modifications to Synchronbench (a), (b) and successive modifications to Ascylib (c). Synchron1 to Synchron4 involve updates to the Synchronbench microbenchmark design. Synchron5 updates the data structure implementation to that of Setbench. Figure (a) uses a key range of 2 million. Figure (b) uses a key range of 20,000 keys and displays the impact of successive modifications to Synchronbench at a 100% update rate.

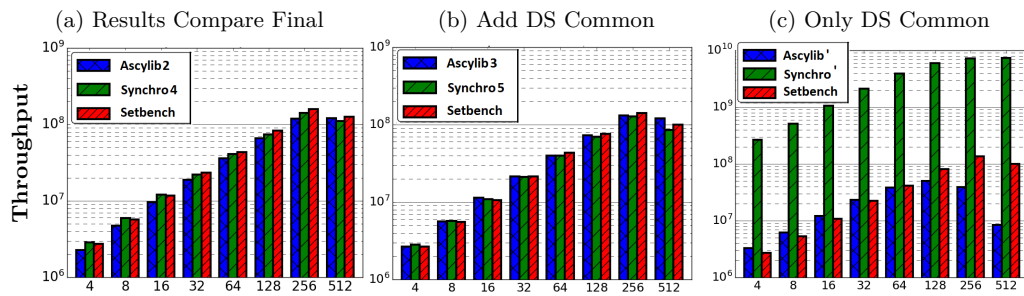
using insert-only operations to reach a start state where the data structure contains exactly half of the keys from the specified input range. Memory reclamation was turned off in all microbenchmarks. Attempted updates and effective updates are both counted towards the total operation throughput. We examine throughput results for experiments running for 20 seconds with update rates varying from 0% to 100% and a specified range of 2 million keys unless stated otherwise. Enforcing the range to a power of 2 is turned off in Ascylib experiments to match the other microbenchmarks. Initial results across the three microbenchmarks can be seen in Figure 2 where throughput values are displayed on a logarithmic y-axis. We observe a range of varying performance results on the lock-free BST across the three microbenchmarks. In particular, across all experiments, Synchronbench throughput results are one to two orders of magnitude higher than Setbench or Ascylib. Ascylib results are notably lower than those of Setbench and Synchronbench. We also observe that Ascylib throughput results tend to plateau at about 128 threads and do not indicate growth as is expected and seen with Setbench and Synchronbench. We investigate further to understand the role of individual microbenchmark design on performance results.

4.5 Performance Factors: Synchronbench

Further investigation is required to understand the underlying causes of comparatively spiked performance results from the Synchronbench microbenchmark seen in Figure 2. In the following set of experiments we aim to equalize the performance results of Setbench and Synchronbench through various adjustments made to Synchronbench where errors or bugs were discovered. We modify the original installation of Synchronbench and title each updated version as `SynchroX`, where `X` is the adjustment number. With each successive modification, for both Ascylib and Synchronbench, all previous modifications are maintained unless stated otherwise. A summary of modifications performed in our work are listed in Table 2.

4.5.1 Missing Insertions

Synchronbench utilizes a file (`test.c`) in each data structure implementation in order to run the microbenchmark experiment loop. Each thread executes in the loop for the duration of the experiment, and all threads are joined prior to termination. Insert operations occur only following a successful delete operation which indicates success by setting a variable `last` to `-1`. This value is checked on the next update operation; if `last` is negative an insert operation



■ **Figure 4** (a) Throughput results display the final comparison across three microbenchmarks with all successive modifications (Synchro4 and Ascylib2). Figure (b) tests Synchro5 and Ascylib3 which maintain all microbenchmark changes and also update the data structure (DS) implementation to that of Setbench. Figure (c) tests Ascylib' and Synchro' which do not contain any modifications to the microbenchmarks and only equalize the DS implementation.

will proceed. However, the `test.c` file in the lock-free BST directory contained a bug in which `last` was an unsigned type and could never take on negative values. As a result, all experiments on the lock-free BST were performing update operations comprised of deletions and never insertions. The data structure is initially prefilled to half of the specified range, but following prefiling, insert operations never take place due to this particular bug in the experiment loop. Delete-only update operations generate notably higher throughput results since the data structure becomes empty very quickly; essentially all operations reduce to searches as the duration of the experiment increases. Upon correction of this bug, throughput results lowered significantly. Performance results of progressive adjustments to Synchrobench are illustrated in Figure 3. This adjustment was the first of a series of modifications made to the original version of Synchrobench for the lock-free BST and is labelled `Synchro1` in the figure. There is a drop in throughput from the original installation of `Synchro` to `Synchro1`. We note that missing insert operations in the experiment loop was not a common occurrence in other data structure directories of Synchrobench.

4.5.2 Thread PRNG seeds

The test algorithm (`test.c`) for the lock-free BST data structure did not assign each thread a unique initialization seed for use in the PRNG employed to generate random keys. Having a PRNG initialized with the same seed per thread resulted in threads utilizing the same set of keys for search/update operations, resulting in an overall high throughput. As the duration of the experiment and the number of threads increase, updates are again essentially reduced to search operations due to other threads having previously completed the requested operation on the given key. Inserts fail because the key is already there, deletes fail because the key was removed by another thread. With `Synchro2` we correct this problem with the addition of randomly generated seeds to initialize each thread's PRNG. The impact of this update can be seen more prominently in Figure 3(b) where there is a drop in throughput with `Synchro2` on 100% updates operating on a 20,000 key range. This is not so visible when the key range is much larger. At a key range of 2 million, the dominant overhead in operations is traversing a large tree; therefore, we see less variation in throughput from `Synchro2` to `Synchro5` in Figure 3(a). The probability of contention on the same set of keys is lower at 2 million keys, therefore, the impact of `Synchro2` is more prominent in smaller key ranges.

4.5.3 Standardized PRNG

As discussed earlier, Synchronbench utilizes a standard built in C++ PRNG, `rand()` to supply randomly generated keys. Setbench and Ascylib use XOR-shift based PRNGs. The Synchronbench microbenchmark is adapted to support the XOR-shift based PRNG employed in Setbench (MM3). This adjustment is labeled **Synchro3** in Figure 3. The adjustment does reduce overall throughput as MM3 uses a more complicated random number generation algorithm, using multiply and XOR-shifts, than what was previously employed in Synchronbench.

4.5.4 Effective Insert and Delete Operations

In attempt to equally distribute insert and delete operations across threads, Synchronbench uses an effective update strategy. Effective updates require threads to perform one type of update successfully before the other type of update is attempted. For example, a thread must perform an insert operation that successfully modifies the data structure before it can attempt a delete operation. This is considered an *effective* update, an approach we found to offer no tangible benefit and can be unforgiving of data structure specific bugs. Effective updates should not be confused with the `-f` (effective) option. The `-f` option in Synchronbench controls only how failed update operations will count towards total throughput, but an effective update strategy for insertions and deletions is used regardless.

Enforcing effective updates is problematic because, for example, in an almost full data structure, to perform an effective insert, one may need to repeatedly attempt to insert many random keys until one succeeds. Essentially, a number of search operations are inserted in between insert and delete operations, thereby inflating the total number of operations. The implementation of the lock-free BST in Synchronbench has a known concurrency bug contained in the original algorithm [3]; modified nodes are not always correctly updated in the tree. The requirement for effective updates in the experiment can generate results which erroneously indicate performance gains in the presence of errors in the implementation. The approach followed in Setbench is to randomize insert and delete operations using per thread PRNGs. This will generate more accurate performance results in spite of possible errors in the implementation. This adjustment is added to Synchronbench and is labelled **Synchro4**.

It may also be noted that a *checksum validation* step would prove beneficial in Synchronbench to catch data structure related concurrency bugs. A checksum validation verifies that the sum of keys inserted minus the sum of keys deleted into the CSet *during* an experiment should equal the final sum of keys contained in the CSet *following* the experiment. Incidentally, the implementation of the lock-free BST in Synchronbench was failing checksum validation. **Synchro4** is the final correction to the Synchronbench microbenchmark design. The data structure specific concurrency bug is updated in the next modification.

4.5.5 Equalizing the Lock-Free BST Implementation

The final update to Synchronbench is a modification of the data structure implementation and equalizing the three microbenchmarks to use the lock-free BST implementation provided in Setbench. The Setbench implementation corrects the concurrency bug and adds checksum validation, which does not exist in the other microbenchmarks. This adjustment is labeled **Synchro5**. We do not see a large difference in performance from **Synchro4** to **Synchro5** in Figure 3(b), which highlights the need for randomized insert and delete operations in concurrent microbenchmark experiments. By employing a randomized update operation assignment, we mitigate the impact of concurrency bugs on overall CSet data structure

performance. We also assess an implementation of Synchronbench, *Synchro'* (Synchro prime), with the imported lock-free BST implementation from Setbench which does not include any modifications to the Synchronbench microbenchmark given in *Synchro1* to *Synchro4*. This comparison is given in Figure 4.

4.6 Performance Factors: Ascylib

The Ascylib microbenchmark test algorithm and underlying default settings lead to a few factors that impact performance results on the lock-free BST. Each successive modification to Ascylib is labeled *AscylibX*.

4.6.1 Thread Pinning

The Ascylib general installation enables thread pinning by default. With further investigation, we found that built-in thread pinning settings were under utilizing the 256 available cores during experimentation. Ascylib captures the underlying core and NUMA node count at compile time; we updated build settings to ensure the correct number of cores were detected. Although the Ascylib build displays that the correct number of cores have been detected, we found the Ascylib throughput results in Figure 2 were based on under 50% core utilization. The default settings were unable to utilize the full set of cores. To remove the underlying thread pinning settings, and disable thread pinning entirely, we recompiled with `SET_CPU=0`. This adjustment is labelled *Ascylib1*. Results for *Ascylib1* indicate full core utilization and improve performance in Figure 3(c). A user that is unaware of Ascylib's default setting may unknowingly generate misleading results. Rather than modifying the three benchmarks to perform identical thread pinning, we disabled thread pinning in all three for consistency. This is perhaps not ideal for microbenchmark experiments. Recommendations for thread pinning in microbenchmark experiments are discussed further in Section 7.

4.6.2 Standardized PRNG

As was the case with the Synchronbench microbenchmark, we use the same PRNG across all three microbenchmarks. Ascylib is updated to use the MM3 PRNG employed in Setbench. The update is labelled *Ascylib2*. We do not see a significant observable change in performance on a logarithmic scale between *Ascylib1* and *Ascylib2*. The MM3 algorithm is a more complicated PRNG (multiply, XOR-shifts) than what was previously used in Ascylib (Marsaglia XOR-shift [30]). Additional testing reveals a slight drop in performance when switching the PRNG to MM3.

4.6.3 Equalizing the Lock-Free BST Implementation

Last, for a comparison that evaluates a standard data structure implementation on each microbenchmark, we implement the lock-free BST implementation from Setbench into Ascylib. This is labelled *Ascylib3*. *Ascylib3* maintains all previous benchmark adjustments whereas *Ascylib'* only updates the common data structure implementation from Setbench into the original installation of Ascylib (Table 2).

4.7 Final Comparisons

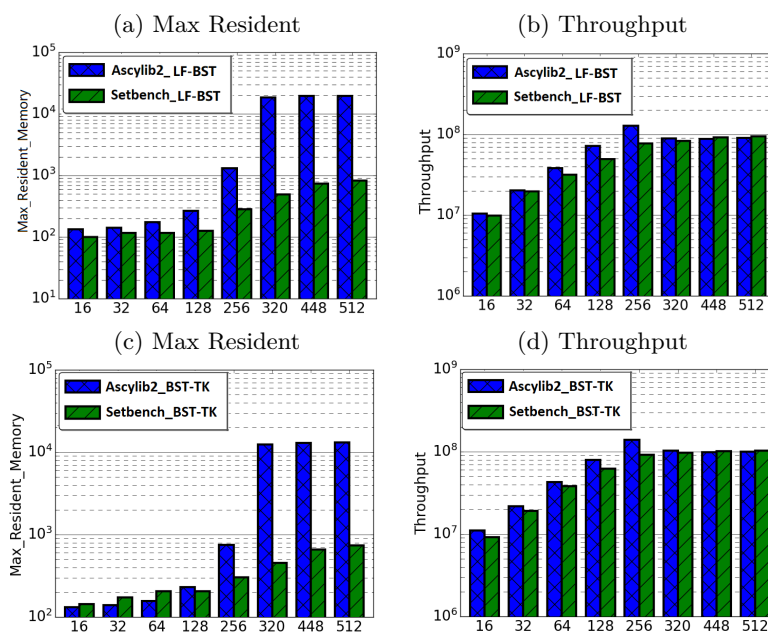
The final comparative results following successive modifications to Ascylib and Synchronbench are given in Figure 4(a), which tests *Ascylib2*, *Synchro4* and the original Setbench implementation. These implementations use the built in data structures of each microbench-

mark while adjusting for microbenchmark design differences in an attempt to equalize the throughput results. We have achieved throughput results that are fairly consistent across microbenchmarks. There are slight discrepancies in throughput results, but these are not nearly as drastic as the performance differences across the original implementations of Ascylib and Synchrobench in Figure 2. Additional final comparisons are provided in Figure 4(b) and (c), which also equalize the lock-free BST implementation across all microbenchmarks on a 100% update workload across 2 million keys. Figure 4(b) includes all microbenchmark modifications for both Synchrobench and Ascylib, whereas Figure 4(c) does not include any microbenchmark modifications from the original installed versions of Synchrobench and Ascylib. The results in Figure 4(c) illustrate the variations in throughput that occur on account of microbenchmark implementation differences. We see that once microbenchmark idiosyncrasies have been ironed out in (a) and (b), the performance results are much more consistent. This highlights again the crucial role of microbenchmark design in the performance of CSet data structures.

4.8 Microbenchmark Design Considerations

In this section we investigated microbenchmark idiosyncrasies between three microbenchmarks. We performed successive modifications to two of the microbenchmarks to account for design differences. During our experiments, we discovered the following factors in microbenchmark design which lead to the greatest impact on performance: (1) Repeated benchmark code is prone to error. In Synchrobench where the algorithm running performance experiments is duplicated for each data structure, errors in the algorithm led Synchrobench results to exceed other microbenchmarks by 100x. The microbenchmark testing algorithm should exist in one centralized location and provide easy adaptation to new data structures. (2) Microbenchmarks use a variety of techniques for splitting the update rate between insert and delete operations. Recommended practice is to randomly distribute update operations between inserts and deletes using per thread PRNGs. (3) Synchrobench introduced a setting to enforce effective updates. We note in Section 4.5.4, effective updates unnecessarily inflate throughput results and are not recommended. (4) Our recommended best practice for microbenchmark design includes strategies to detect and mitigate errors in the microbenchmark. We certainly recommend a checksum validation in microbenchmark experiments. In our work, adding checksum validation assisted in discovering microbenchmark and data structure implementation errors.

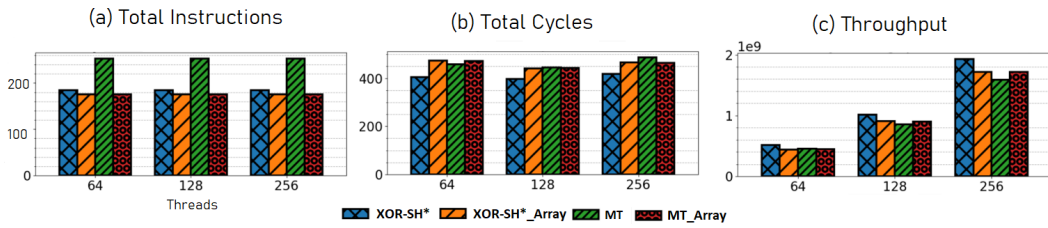
Prefilling a CSet prior to running the microbenchmark experiment is also an important design consideration. Although experiments in this section used insert-only prefilling, we recommend against this for CSet microbenchmark experiments. (5) Data structure prefilling should occur through (a) randomized insert and delete operations, and (b) using the same n threads that will be employed during the measured portion of experiments. This will generate a more realistic configuration of a concurrent data structure in steady state as opposed to a data structure prefilled using single-threaded insert-only operations. Single-threaded prefilling will result in memory allocation specific to one thread's NUMA node. This will result in memory access latency for threads on different NUMA nodes during the measured portion of experiments. Using n threads to perform prefilling will disperse memory allocation across additional NUMA nodes. N -threaded prefilling with randomized insert and delete operations is used in Setbench as mentioned previously. We discuss additional considerations in microbenchmark design and provide further recommendations in Section 7. In the next section, we experiment with memory reclamation in microbenchmarks and evaluate its impact on performance.



■ **Figure 5** Maximum Resident Memory and Throughput results for Ascylib and Setbench on the lock-free BST ((a), (b)) and BST-TK ((c), (d)). Ascylib implementation contains microbenchmark updates contained in Ascylib2 (Section 4.2).

5 Memory Reclamation

A key measure of memory usage for an executing program is the *maximum resident memory* occupied in RAM by the program during the duration of its execution. The lock-free BST as described by Natarajan et al. [37] does not provide a complete algorithm for memory reclamation during execution. The partial algorithm suggests removing an unbounded number of nodes that are nearby neighbours in the tree pending deletion. Any given thread may proceed to delete and free (unlink) n nodes that are in close proximity within the tree. However, the original implementation was leaking memory. Synchrobench does not implement any memory reclamation in its implementation, whereas Ascylib has an added option for garbage collection (GC). The authors of the lock-free BST suggest adding epoch based memory reclamation, but it was not so simple. The memory reclamation algorithm from the original work is updated in the Setbench implementation to correctly reclaim memory [3]. We first compare the memory reclamation implementations in Setbench and Ascylib by setting Ascylib’s GC setting to true, and Setbench epoch based reclamation is turned on. We show comparative analysis of results across each microbenchmark in Figure 5(a) and (b). The Ascylib microbenchmark has been updated to `Ascylib2` in order to disable thread pinning and equalize the PRNG utilized in both microbenchmarks. We have ensured all 256 cores are being utilized by Ascylib. Figure 5(a) illustrates differences in each microbenchmark’s ability to reclaim memory as the thread count increases and cores are oversubscribed. Ascylib’s memory usage surpasses that of Setbench by over one order of magnitude, particularly as the thread count increases. Throughput results (Figure 5(b)) are relatively equal, however, the high maximum resident memory values may render Ascylib experiments unfeasible in some settings. We further consider microbenchmark comparisons on the equalized lock-free BST implementation with memory reclamation turned on. We



■ **Figure 6** Array Based Pre-generated PRNG vs Non Array based PRNG on a key range of 20 000. (a) Total Instructions per operation (b) Total Cycles per operation (c) Total Throughput per second

discover similar performance discrepancies to those discussed in Section 4, although the data structure implementation and memory reclamation algorithms are identical across the three microbenchmarks. Performance results continue to show variance until microbenchmark idiosyncrasies are accounted for. Results for these additional experiments can be seen in Appendix A of the full paper [22].

5.1 Setbench/Ascylib BST Ticket

In Section 4 of this work, we examined performance factors for the lock-free BST on three concurrent synthetic microbenchmarks. We noted substantial impacts on performance as a result of microbenchmark implementation intricacies. In this section we investigate performance differences across the BST ticket (BST-TK) CSet data structure as implemented in the Setbench and Ascylib microbenchmarks. The ticket based binary search tree by Guerraoui et al. [11] appears in both Setbench and Ascylib microbenchmarks; however, it is not implemented in Synchrobench. The BST-TK is an external binary tree where leaf nodes contain the set of keys contained within the data structure. Internal nodes are used for routing and contain locks and a version number. This allows optimistic searches on the tree where concurrency can be verified by the correct version number. Both Ascylib and Setbench implement the BST-TK with memory reclamation. Ascylib has garbage collection (GC) turned on, Setbench performs epoch based reclamation. We observe in Figure 5(d) that throughput results from both microbenchmarks are similar on the BST-TK data structure. In Figure 5(c), we see again the Ascylib microbenchmark has higher memory usage, a greater than one order of magnitude increase over Setbench. This may render Ascylib experiments impractical in some settings and indicates memory is leaking at higher thread counts.

We have seen microbenchmarks can vary greatly in performance and memory usage across two different concurrent data structure implementations. We recommend microbenchmark users investigate overall memory usage in parallel with throughput results in order to get a clear understanding of the role of memory reclamation on the performance of a CSet. Memory may not be leaking necessarily; if the memory reclamation algorithm is simply slow or inefficient, there maybe a tangible impediment on performance.

6 Randomness in Concurrent Microbenchmark Experiments

As we have seen in previous sections, concurrent microbenchmarks rely heavily on PRNGs to generate randomized keys and randomized operations for high performance CSets that can perform potentially billions of operations per second. A fast PRNG is key. In this section we draw our attention to best practices of PRNG usage in concurrent microbenchmark experiments. We limit our attention to non-cryptographic PRNGs due to the speed requirement.

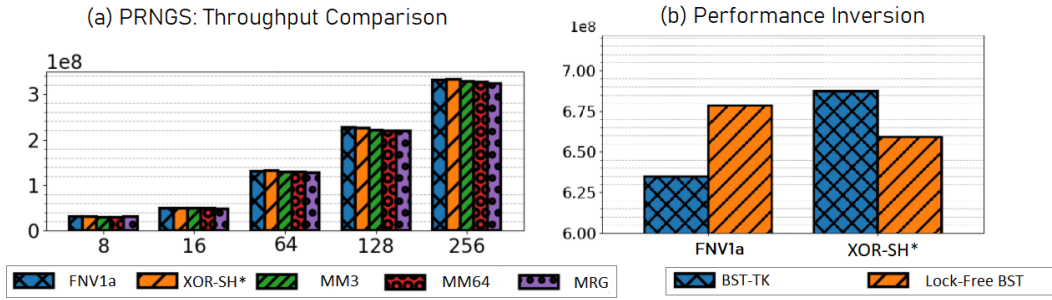
It is desirable to utilize a PRNG with low overhead costs to the running experiment. Some microbenchmarks may choose a custom built PRNG, while others may opt for a standard built-in PRNG such as `rand()` used in Synchrobench. Some will pregenerate an array of random numbers (RNs); this allows fast, direct access to a list of RNs and avoids in-place generation costs. If properties of high quality randomness are desired, one may use an architecture specific hardware RNG. We explore the practicality and benefit of these approaches in subsequent sections. The PRNGs tested in this work include commonly used software PRNGs: murmurhash2 (MM2) [45], murmurhash3 (MM3) [1], Mersenne Twister (MT) [31], MRG [36], and an implementation of the Marsaglia XOR-shift PRNG (XOR-SH*) [10, 30]. We describe custom hash functions and hardware RNGs in subsequent sections. Experiments in this section were run for durations of 3-5 seconds.

6.1 Pre-Generated Array of Random Numbers

One might be tempted to think that the best way to obtain fast, high quality randomness would be to pre-generate a large array of RNs (or one array per thread) before running an experiment. Then, one could employ hardware randomness, or a cryptographic hash function, and push the high cost of generating random numbers into the unmeasured setup phase of the experiment. We tested this method in Setbench with per thread arrays of pre-generated RNs using the XOR-SH* and MT PRNGs versus in-place RN generation with each algorithm. It is important to note that the pre-generated array approach eliminates the cost of in-place random number generation; during an experiment it is simply a matter of requesting an index into an array to generate the next random. A limitation of an array-based approach is, of course, the array size. It is undesirable to have frequent repetition of RNs during experimentation. We use array sizes of 10 million to generate a large set of RNs. Results in Figure 6(c) indicate the XOR-SH*_Array employed during experiments was notably slower than using the XOR-SH* algorithm in-place. This is due to the fact that accessing a large array of 10 million will lead to additional clock cycles generated by cache misses. An algorithm that is relatively fast, such as the XOR-SH* PRNG, will not benefit from taking a pre-generated array-based approach. However, a slightly more complex algorithm such as MT, which requires more instructions (Figure 6(a)), can benefit from an array-based approach. The MT_Array generates slightly higher throughput results than using MT alone as indicated in Figure 6(c). However, the benefit is not as striking as one may expect with an array-based PRNG approach. There may be use cases for an array-based PRNG such as requiring a more complicated (exotic) distribution of RNs. In this case, pre-generating RNs in an array may be an effective approach to limiting the overhead of a complex algorithm.

6.2 PRNG Associated Experimental Anomalies

In the search for high-speed generation of RNs, researchers may choose to implement their own PRNGs or use a custom hash function that may not have been well tested for properties consistent with high-quality PRNGs. Prior to this work, the PRNG used in Setbench was FNV1a [26], a fast, non-cryptographic 64-bit hash function. FNV1a is recommended by Lessley et al. as a hash function with “*consistently good performance over all configurations*” [26]. Setbench employed an FNV1a based PRNG that was used to generate both random operations and random keys. However, upon testing single threaded experiments, we noticed that the data structure prefilling step was failing to converge (i.e., it was non-terminating). Upon further investigation, we found that the FNV1a based PRNG was generating RNs that followed a strict odd-even pattern. That is, after generating an even number, the next number would always be odd, and vice versa. (The initial seed



■ **Figure 7** (a) Throughput results comparing FNV1a to other PRNGs; FNV1a algorithm does not indicate any detectable performance anomalies in throughput. Figure (b) on a smaller key range (4096), illustrates the subtle effects of a PRNG. There exists a performance inversion when using the FNV1a PRNG vs XOR-SH*.

determined whether the first number was even or odd.) During prefilling, a thread uses the first RN to determine the key and the second RN to determine the operation. In this case, the set of keys were always either all odd or all even, leading to an infinite loop when attempting to prefill the data structure to half full. Recall that Setbench employs both insertions and deletions to prefill the CSet to steady state (half-full in this case). This odd-even pattern can easily be missed in overall data structure performance results. Figure 7(a) illustrates throughput results comparing various PRNGs tested in Setbench. There is no notable indication of threads generating all even or all odd keys from the FNV1a algorithm. Some threads are generating all even keys, while others are generating all odd keys. Setbench prefilling occurs with n threads, so as soon as the thread count increases from 1, the probability of convergence increases. One could imagine this kind of error remaining undetected and having a subtle effect on performance; limiting the set of keys per thread will affect which other threads it could contend with. In addition, it is not sound experimental methodology for a microbenchmark to generate keys based on this pattern. Second, this undesirable behaviour found in FNV1a can lead to performance inversions when evaluating CSets in a microbenchmark. The impact of the FNV1a based PRNG is more clearly displayed in the results of Figure 7(b) where, given a high insert workload, FNV1a can lead to a performance inversion of experimental results. The experiment illustrates that the lock-free BST (Natarajan et al.) [37] throughput results are 1.12 times higher than that of the BST-TK (Guerraoui et al.) [11] when Setbench is using FNV1a as its PRNG. However, using another PRNG, such as XOR-SH*, we see the results indicate the lock-free BST *underperforms* by a factor of 0.96. This is an approximately 16% performance error leading to an inversion of results that could possibly remain undetected when one concurrent microbenchmark employs a problematic PRNG algorithm such as FNV1a. Incidentally, FNV1a also illustrated periodic behaviour in higher order bits. We implement a tool, the N^{th} - bit summation result, to assess bitwise randomness in RNs generated by a PRNG (Appendix B of full paper [22]).

6.3 Hardware RNG

A search for a high-quality 64-bit PRNG with its own source of entropy led us to an Intel Secure Key instruction, RDRAND, available on Ivy Bridge processors [19]. The RDRAND instruction returns an RN from Intel on-chip RNG hardware. We compare RDRAND with the software based PRNGs to evaluate the suitability of a hardware based PRNG for use in synthetic microbenchmarks. We have illustrated throughput results in experiments with

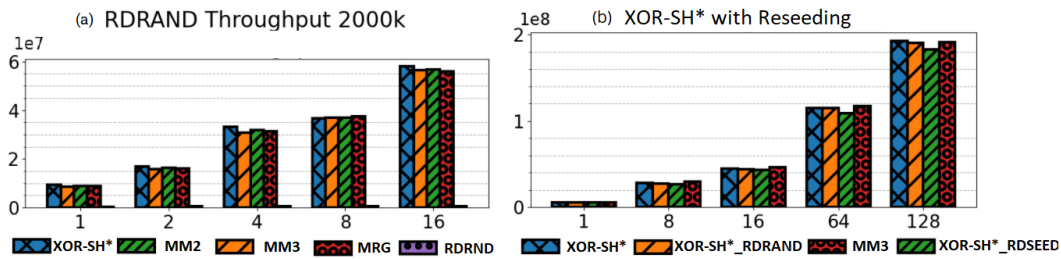


Figure 8 Hardware vs Software PRNGs: Figure (a) RDRAND throughput compared to software PRNGs. RDRAND has the lowest throughput values. In Figure (b) Reseeding XOR-SH* with RDSEED (or RDRAND) every 1 million random numbers indicates no strong penalty for reseeding a software PRNG with a hardware RNG.

various PRNGs in Figure 8(a). We can see that the experimental throughput of RDRAND is significantly less compared to software PRNGs. A smaller key range of 2000 keys was necessary to visually illustrate the low throughput results generated when RDRAND is employed in Setbench. The overhead costs of hardware entropy greatly impede the overall performance of an experiment which aims to maximize throughput results. Algorithms such as XOR-SH* and MM3 that are computationally fast in nature have much higher throughput results. For concurrent microbenchmark experiments, it is not recommended to use a hardware PRNG alone. During our experimentation, we found that because RDRAND is a significant bottleneck in the benchmark experimental loop, results can appear equal for two CSet data structures that otherwise behave very differently. Although RDRAND is slow, it can be useful as a source of entropy for faster software PRNGs. The idea of periodically reseeding to introduce additional randomness into a PRNG is discussed by Manssen et al. [28] and Dammertz [20]. RDSEED is an Intel Secure Key instruction that complements RDRAND and is used to generate high quality random seeds for seeding PRNGs [44]. RDSEED is slower than RDRAND but is recommended to use for reseeding PRNGs. We tested a hybrid PRNG solution on the XOR-SH* algorithm where RDSEED is used for reseeding at intervals of every 1 million RNs (*XOR-SH*_RDSEED*). The results in Figure 8(b) indicate comparable throughput results to purely software based PRNGs without reseeding. We compared XOR-SH* reseeding with RDSEED to XOR-SH* reseeding with RDRAND (*XOR-SH*_RDRAND*), and there is a small drop in performance with RDSEED.

6.4 PRNG Recommendations

Massively parallel, high throughput experiments require billions of random numbers to be generated per second, which pushes the limits on PRNGs of our time. Some important points to consider for PRNG usage in microbenchmarks: (1) Hardware RNGs provide an external source of entropy, however, they are impractical for use in high speed concurrent microbenchmark experiments as the performance results are greatly impeded by RN generation time. (2) A pre-generated array of RNs is also counterproductive due to penalties associated with cache access. A pre-generated array of RNs may be useful if the PRNG algorithm is complex and in-place RN generation is too expensive. (3) For synthetic microbenchmark experiments we recommend two PRNG instances per thread; one for generating random values during the experiment and one for injecting new entropy into the first PRNG (periodic reseeding). If periodic reseeding is used every 1 million keys, there is a low, intangible impact on performance. If RDSEED or RDRAND are not available, we recommend using a

high quality cryptographic PRNG for the 2nd PRNG. (4) Experiments that rely on bitwise randomness in bits should first examine the set of generated random numbers for periodic behaviour in bits. (5) Last, in an era where data structures are performing billions of operations per second, we also think it's important to use PRNGs with at least 64 bits of state to avoid repeating the same sequence of generated keys on a time scale of seconds.

7 Towards Better Microbenchmarks

In Section 4.8, 5.1 and 6.4 we gave some recommendations for good benchmark design that were informed by our study of Ascylib, Setbench and Synchrobench. Setbench was designed with many of those recommendations in mind, and underwent relatively few changes as a result of our study. In this section, we give some additional recommendations and highlight features of Setbench that promote high quality experiments.

7.1 Additional Recommendations

More expressive ADTs

Today, many CSet data structures support range query operations and other interesting operations such as clone and size, and benchmarks should consider including support for them. It is not necessary for every operation to be implemented by every data structure, but providing a framework for additional operations to be included in experiments may encourage research in this direction.

Similarly, we encourage support for maps (also called dictionaries), which associate a value with each key, and support for large and/or variable-sized keys and values. This could encourage evaluations that span data structures published in distributed computing venues and those published in database and data management venues (e.g., [4, 25, 27, 29]) – evaluations that are desperately needed in our opinion.

Starvation-aware experiments

Note, however, that some care is needed in experimenting with range queries, and any other types of long-running operations that are prone to starvation. Consider a workload where threads perform, say, 49% insert, 49% delete and 2% range queries spanning the entire range of keys contained in the data structure. One would expect such range queries to be starved by updates, but in practice we find they are not! The trick is that *each* thread will perform only so many updates before performing a range query. So, if all range queries are perpetually starved, while updates succeed, eventually *all* threads will be executing range queries, and they will all succeed in a batch. This behaviour makes starvation seem like less of a problem than it might be in the real world, where there might *never* be a time when there are no updates in progress. Experiments involving starvation prone operations should *expose* the effects of starvation, possibly by allowing groups of threads to be dedicated to starvation-prone and non-starvation-prone operations respectively (see, e.g., [2]).

Pinning threads

In Section 4.6.1, we disabled thread pinning in all microbenchmarks in order to have consistency across all experiments. We recommend pinning threads to improve consistency of experiments, so for example, when you run 48 threads on a system with four 48-thread sockets, your threads run on a predictable set of cores, rather than, e.g., being clustered on one socket in one execution, and spread across three sockets in another. Additionally, thread

pinning should be used to clearly expose the performance impact of hyper threading and the effects on non-uniform memory architectures in performance graphs. Thread pinning in benchmarks has been discussed in more detail by Gramoli et al. [14, 15] and Brown [6].

Non-uniform key distributions

Benchmarks should also consider incorporating various distribution generators for keys and values, rather than limiting experiments to uniform randomness. Researchers should consider using Zipfian, binomial, exponential or other skewed distributions in their experiments [35]. Distribution generators should be implemented efficiently, and sanity checks should be performed to ensure that the rate of key/value generation is not a bottleneck.

Uniform memory reclamation

Research in safe memory reclamation for CSets has consistently demonstrated that CSet performance can depend heavily on the algorithm for reclaiming memory (see, e.g., [9, 39, 43, 46, 52]). For this reason, memory should be reclaimed similarly across all data structures evaluated. In some cases, ad-hoc memory reclamation is tightly integrated in a CSet, but benchmarks should offer a fast, easy-to-use memory reclamation algorithm to promote uniformity wherever possible.

Performance tools

Benchmarks should make a best effort attempt to *automatically* gather lightweight systems-level performance data, such as cache misses per operation, total cycles per operation, and peak memory usage. We suggest incorporating a library for performance monitoring such as the Performance Application Programming Interface (PAPI) [49]. We think it is crucial that these measurements are not only automatically gathered, but automatically *visualized*. Ideally, graphs for CSet throughput results and for systems level performance monitoring would be produced by default, at the same time, and would be visible in the same place. “Easy to check” is good. “Difficult to ignore” is better.

7.2 Benchmarking Advances in Setbench

Setbench was specifically designed to address all of the recommendations above, featuring a 256-bit PRNG, range query support (with support for independent range query threads and update threads), the ability to specify thread pinning policies at the command line, fast Zipfian and Uniform key distributions, uniform epoch based memory reclamation, and integration with a rudimentary implementation [53] of TPC-C and YCSB application benchmarks. It also includes a large collection of powerful tools for debugging, running experiments and analyzing performance, as well as automatic containerization for artifact evaluation.

Collecting user defined statistics

Debugging and performance analysis are extremely time consuming, and often researchers are limited in how much investigation they can do by the time it takes to modify their code to record specific events in their data structure. These events can be quite varied.

For example, one might want to answer a simple question like: in a lock-free algorithm, *how often do threads help complete other threads’ operations?* Or, in an algorithm that uses epoch based memory reclamation, where objects are reclaimed in batches, one might want to

7:20 Performance Anomalies in Concurrent Data Structure Microbenchmarks

answer a much more complex question – *how to produce a logarithmic histogram showing the distribution of the sizes of the first 10,000 batches reclaimed by each thread in an execution*. Setbench’s global stats library (**gstats**) makes it fast and easy to explore such questions. To emphasize how easy gstats makes this, to implement the latter, one would first create a gstats statistic that is accessible globally (throughout all files in the entire benchmark), by adding the following code to a file in Setbench called `define_global_statistics.h`:

```
gstats_handle_stat(LONG_LONG, epoch_batch_size, 10000, \
    { gstats_output_item(PRINT_HISTOGRAM_LOG, NONE, FULL_DATA) }) \
```

In essence, this efficiently allocates global per-thread arrays of 10,000 elements, and specifies that their contents should be used to build a logarithmic histogram. Whenever a thread *T* reclaims a batch of size *n*, it can append the batch size *n* to its array by invoking:

```
GSTATS_APPEND(T, epoch_batch_size, n);
```

These simple modifications result in the following new output when the benchmark is run:

```
log_histogram_of_none_epoch_batch_size_full_data=[...]
[2^00, 2^01]: 71905
(2^01, 2^02): 206257
(2^02, 2^03): 307829
(2^03, 2^04): 469972
[...] // output truncated to save space
```

Furthermore, scripts are included to plot bar graphs and line graphs from any data collected with gstats. In this case, assuming the output above is in `data.txt`, one would simply run: `trial_to_plot.sh data.txt epoch_batch_size`, which would create a PNG file.

Running Experiments and Plotting Results

Setbench also offers a powerful suite of Python scripts for running experiments and automatically plotting their results. Example run scripts that are suitable for CSet research are included. They produce Matplotlib graphs of throughput *and many systems level performance metrics*, such as L3 cache misses per CSet operation, cycles per operation, and peak memory usage. Scripts are also available for several papers published by our group. The development of these scripts focused on conciseness, expressiveness and flexibility, and the scripts could be adapted to drive completely different benchmarks in different domains.

At a high level, to use these scripts, one defines a sequence of experimental *parameters*, and for each parameter, one specifies a list of values the parameter should take on. One then specifies a *run command* for the benchmark, and specifies how the parameters should be supplied to the run command. The command is run for each combination of parameters, and the output of each run is stored in an individual file. The scripts then process each file, and extract lines of the form “NAME=DATA” to produce columns in a **sqlite database**. As part of this process, data is *validated* according to user specified rules such as (`‘total_throughput’, is_positive`) or (`‘validate_result’, is_equal(‘success’)`). Failed validation causes (colourful!) warnings to be emitted, and warnings can also be queried later from the sqlite database.

The scripts expose functions for easily producing plots (bars, lines, histograms and heatmaps) from the sqlite database simply by specifying which columns of data should be used for the x-axis and y-axis. Additional columns can be specified and graphs will be produced for every combination of values in these columns. Filters can also be specified to add to the SQL WHERE clauses in the queries that underpin plot generation.

In short, a single command `run_experiment.py [your_experiment.py]`, depending on its arguments, can compile (`-c`), run (`-r`), create the sqlite database (`-d`), produce graphs (`-g`) and create an HTML website (`-w`) organizing them into sections for convenient viewing. Clicking a graph on the website drills down to the rows of data the graph was built from, and clicking a row shows the raw text output for that run. A generated example website can be viewed at: https://cs.uwaterloo.ca/~t35brown/setbench_example_www. Results in the sqlite database can also be queried conveniently from the command line using SQL (e.g., `run_experiment.py your_experiment.py -q "select * from data"`). A wide range of additional capabilities are documented in extensive Jupyter notebook tutorials.

8 Conclusions

We hope this work encourages further research into how best to design benchmarks for concurrent data structures. Setbench was carefully designed to mitigate many of the problems we are aware of, but there are surely more benchmarking pitfalls yet to be discovered in this area. We also encourage researchers to try using Setbench for their own experiments, because its features make it much easier to drill down to the root causes of performance anomalies. After designing an algorithm, proving correctness, and implementing it, there is often little time left to do systems level performance analysis. Our hope is that by improving tools and automating the collection and graphing of key performance metrics, we can improve the quality of experiments without unduly burdening researchers in this area.

References

- 1 Austin Appleby. Murmurhash3, 2012. URL: <https://github.com/aappleby/smhasher/wiki/MurmurHash3>.
- 2 Maya Arbel-Raviv and Trevor Brown. Harnessing epoch-based reclamation for efficient range queries. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '18*, pages 14–27, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3178487.3178489.
- 3 Maya Arbel-Raviv, Trevor Brown, and Adam Morrison. Getting to the root of concurrent binary search tree performance. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 295–306, 2018.
- 4 Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. Bztree: A high-performance latch-free range index for non-volatile memory. *Proceedings of the VLDB Endowment*, 11(5):553–565, 2018.
- 5 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.
- 6 Trevor Brown. Good data structure experiments are r.a.r.e, 2017. URL: <https://www.youtube.com/watch?v=x6HaBcRJHFY>.
- 7 Trevor Brown. Powerful tools for data structure experiments in c++, 2021. URL: <https://gitlab.com/trbot86/setbench>.
- 8 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.
- 9 Trevor Alexander Brown. Reclaiming memory for lock-free data structures: There has to be a better way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 261–270, 2015.
- 10 Wikipedia contributors. Xorshift. 2022. URL: <https://en.wikipedia.org/wiki/Xorshift>.

- 11 Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15*, pages 631–644, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2694344.2694359.
- 12 Tudor Alexandru David, Rachid Guerraoui, Tong Che, and Vasileios Trigonakis. Designing ascy-compliant concurrent search data structures. Technical report, EPFL Infoscience, 2014.
- 13 J. Evans. Scalable memory allocation using jemalloc, 2011. URL: <https://www.facebook.com/notes/10158791475077200/>.
- 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 Vincent Gramoli. The information needed for reproducing shared memory experiments. In *European conference on parallel processing*, pages 596–608. Springer, 2016.
- 16 Rachid Guerraoui and Vasileios Trigonakis. Optimistic concurrency with optik. *ACM SIGPLAN Notices*, 51(8):1–12, 2016.
- 17 Tim Harris. Benchmarking concurrent data structures, 2016. URL: <https://timharris.uk/misc/2017-tpc.pdf>.
- 18 Tim Harris. Do not believe everything you read in the papers, 2016. URL: <https://timharris.uk/misc/2016-nicta.pdf>.
- 19 Gael Hofemeier and Robert Chesebrough. Introduction to intel aes-ni and intel secure key instructions. *Intel, White Paper*, 62, 2012.
- 20 John E. Hopcroft, Wolfgang J. Paul, and Leslie G. Valiant. Random number generators for massively parallel simulations on gpu, 1975. doi:10.1109/SFCS.1975.23.
- 21 Nikolaos D Kallimanis. Synch: A framework for concurrent data-structures and benchmarks. *arXiv preprint arXiv:2103.16182*, 2021.
- 22 Rosina Kharal and Trevor Brown. Performance anomalies in concurrent data structure microbenchmarks, 2022. doi:10.48550/ARXIV.2208.08469.
- 23 Onur Kocberber, Babak Falsafi, and Boris Grot. Asynchronous memory access chaining. *Proceedings of the VLDB Endowment*, 9(4):252–263, 2015.
- 24 Petr Kuznetsov and LTCI INFRES. Refining concurrency for performance, 2017.
- 25 Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 38–49. IEEE, 2013.
- 26 Brenton Lessley, Kenneth Moreland, Matthew Larsen, and Hank Childs. Techniques for data-parallel searching for duplicate elements. In *2017 IEEE 7th Symposium on Large Data Analysis and Visualization (LDAV)*, pages 1–5, 2017. doi:10.1109/LDAV.2017.8231845.
- 27 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.
- 28 M. Manssen, M. Weigel, and A. K. Hartmann. Random number generators for massively parallel simulations on gpu. *The European Physical Journal Special Topics*, 210(1):53–71, August 2012. doi:10.1140/epjst/e2012-01637-8.
- 29 Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 183–196, 2012.
- 30 George Marsaglia. Xorshift rngs. *Journal of Statistical Software*, 8:1–6, 2003.
- 31 Makoto Matsumoto and Takuji Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, January 1998. doi:10.1145/272991.272995.

- 32 Frank McSherry, Michael Isard, and Derek G Murray. Scalability! but at what {COST}? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.
- 33 Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. Stamp: Stanford transactional applications for multi-processing. In *2008 IEEE International Symposium on Workload Characterization*, pages 35–46. IEEE, 2008.
- 34 Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F Sweeney. Producing wrong data without doing anything obviously wrong! *ACM Sigplan Notices*, 44(3):265–276, 2009.
- 35 N. Unnikrishnan Nair, P.G. Sankaran, and N. Balakrishnan. Chapter 3 - discrete lifetime models. In N. Unnikrishnan Nair, P.G. Sankaran, and N. Balakrishnan, editors, *Reliability Modelling and Analysis in Discrete Time*, pages 107–173. Academic Press, Boston, 2018. doi:10.1016/B978-0-12-801913-9.00003-8.
- 36 Morita Naoyuki. Pseudo random number generator with mrg (multiple recursive generator), 2020. URL: https://www.schneier.com/blog/archives/2008/05/random_number_b.html.
- 37 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, PPOPP '14*, pages 317–328, 2014. doi:10.1145/2555243.2555256.
- 38 Donald Nguyen and Keshav Pingali. What scalable programs need from transactional memory. *SIGPLAN Not.*, 52(4):105–118, April 2017. doi:10.1145/3093336.3037750.
- 39 Ruslan Nikolaev and Binoy Ravindran. Brief Announcement: Crystalline: Fast and Memory Efficient Wait-Free Reclamation. In *35th International Symposium on Distributed Computing (DISC 2021)*, volume 209 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 60:1–60:4. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.DISC.2021.60.
- 40 Javier Picorel, Djordje Jevdjic, and Babak Falsafi. Near-memory address translation. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 303–317. Ieee, 2017.
- 41 Wenjia Ruan, Yujie Liu, and Michael Spear. Stamp need not be considered harmful. In *Ninth ACM SIGPLAN Workshop on Transactional Computing*, 2014.
- 42 Tomer Shanny and Adam Morrison. Occualizer: Optimistic concurrent search trees from sequential code. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 321–337, 2022.
- 43 Gali Sheffi, Maurice Herlihy, and Erez Petrank. Vbr: Version based reclamation. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 443–445, 2021.
- 44 Thomas Shrimpton and R Seth Terashima. A provable-security analysis of intel’s secure key rng. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 77–100. Springer, 2015.
- 45 Hardy-Francis Simon. Murmurhash2, 2010, 2010. URL: <https://simonhf.wordpress.com/2010/09/25/murmurhash160/>.
- 46 Ajay Singh, Trevor Brown, and Ali Mashtizadeh. Nbr: neutralization based reclamation. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 175–190, 2021.
- 47 Ajay Singh and Sathya Peri. *Efficient means of Achieving Composability using Transactional Memory*. PhD thesis, Indian Institute of Technology Hyderabad, 2017.
- 48 Ajay Singh, Sathya Peri, G Monika, and Anila Kumari. Performance comparison of various stm concurrency control protocols using synchrobench. In *2017 National Conference on Parallel Computing Technologies (PARCOMPTECH)*, pages 1–7. IEEE, 2017.
- 49 Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting performance data with papi-c. In Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2009*, pages 157–173, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

- 50 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.
- 51 Haosen Wen, Joseph Izraelevitz, Wentao Cai, H Alan Beadle, and Michael L Scott. Interval-based memory reclamation. *ACM SIGPLAN Notices*, 53(1):1–13, 2018.
- 52 Haosen Wen, Joseph Izraelevitz, Wentao Cai, H Alan Beadle, and Michael L Scott. Interval-based memory reclamation. *ACM SIGPLAN Notices*, 53(1):1–13, 2018.
- 53 Xiangyao Yu. *An evaluation of concurrency control with one thousand cores*. PhD thesis, Massachusetts Institute of Technology, 2015.
- 54 Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. Efficient lock-free durable sets. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–26, 2019.

Robust and Fast Blockchain State Synchronization

Enrique Fynn ✉

Università della Svizzera italiana (USI), Lugano, Switzerland

Ethan Buchman ✉

Informal Systems, Guelph, Canada

Zarko Milosevic ✉

Informal Systems, Guelph, Canada

Robert Soulé ✉

Yale University, New Haven, CT, USA

Fernando Pedone ✉

Università della Svizzera italiana (USI), Lugano, Switzerland

Abstract

State synchronization, the process by which a new or recovering peer catches up with the state of other operational peers, is critical to the operation of blockchain-based systems. Existing approaches to state synchronization typically involve downloading snapshots of system state. Such approaches introduce an attack vector from malicious peers that can significantly degrade performance. Moreover, the process of creating snapshots leads to performance hiccups. This paper presents a technique for peers to catch up with operational peers without trusting any particular peer and gracefully recover from misbehavior during the process. We have integrated our design into a production blockchain middleware. Our evaluation shows that during operation, the transaction throughput is consistently higher without pauses for snapshot construction. Moreover, the time it takes for a new peer to join the blockchain is halved, while at the same time tolerating Byzantine peers.

2012 ACM Subject Classification Computer systems organization → Reliability; Computer systems organization → Availability; Computer systems organization → Redundancy; Computing methodologies → Distributed algorithms

Keywords and phrases state synchronization, replication, blockchain

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2022.8

Funding This work was supported in part by the Swiss National Science Foundation (grant 175717) and NSF FMITF (grant 2019285).

1 Introduction

Intuitively, a blockchain provides an append-only log of transactions implemented by geographically distributed peers. The execution of these transactions determines the system state. Each peer stores the system state in a Merkle tree [23], or similar data structures (e.g., Merkle-Patricia-tree [26]). Executing a transaction involves performing operations on the tree. By executing the log of transactions in the same order, each peer transitions through the same state changes (i.e., state machine replication model [22]). The fact that the state is stored in a Merkle tree allows a peer to validate the consistency of the state by computing a hash on the reconstructed tree. A Merkle-tree is a tree in which every leaf node stores a cryptographic hash of its value, and every non-leaf node stores the hash of its children. Every block header in the blockchain stores the hash of the tree root constructed by the transactions in a previous block (e.g., block n contains the hash of the tree root computed with transactions in block $n - 1$).



© Enrique Fynn, Ethan Buchman, Zarko Milosevic, Robert Soulé, and Fernando Pedone; licensed under Creative Commons License CC-BY 4.0

26th International Conference on Principles of Distributed Systems (OPODIS 2022).

Editors: Eshcar Hillel, Roberto Palmieri, and Etienne Rivière; Article No. 8; pp. 8:1–8:22



Leibniz International Proceedings in Informatics

LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

8:2 Robust and Fast Blockchain State Synchronization

In theory, a new peer joining the network (or a recovering peer) could download the transactions it misses, possibly the entire blockchain, and reconstruct the state by re-executing every missing transaction. However, this is impractical, since the number of transactions grows too large over time. Therefore, existing blockchain-based systems (e.g., [2, 26]) rely on periodic snapshots of the system state. The snapshot is a serialized representation of the tree data structure. When a new peer joins the blockchain (or a failed peer recovers), it downloads the blockchain blocks with transactions (or block headers, with a summary of the block) and the snapshot. The peer then installs the snapshot and replays all transactions since the snapshot was taken, to reconstruct the current system state. To validate a snapshot, a peer simply computes the hash on its tree and compares the computed value with the value stored in the trusted block header. To further decrease the time it takes for a new peer to join the blockchain, a snapshot can be divided into *chunks*, each one containing multiple nodes of the tree. This allows new peers to download chunks from many peers concurrently. A natural strategy to assign tree nodes to chunks is to traverse the tree (e.g., using depth-first search) and build fixed-sized chunks [3].

Unfortunately, this approach to state synchronization suffers from two subtle, but important problems. The first problem is due to the relationship between chunks and validation. If nodes of the state tree are assigned to chunks in a naïve way, as described above, then the chunks cannot be validated independently. Instead, a peer must download the entire tree before it can compute the hash. This introduces an effective attack vector for misbehaving peers. If a single malicious peer shares an invalid chunk, the new peer cannot identify the problem until it has downloaded all of the chunks. This wastes network and disk resources of both the sender and the receiver, and can significantly prolong the time it takes for a new peer to join the blockchain. The second problem is with performance. When a peer computes a snapshot, it needs to search the tree, serialize all nodes, build chunks, and save them to local storage. We show in the paper that existing blockchain-based systems based on this approach experience periodic hiccups, dropping transaction throughput.

In this paper, we provide data structures and algorithms for robust and fast blockchain state synchronization. By robust, we mean that our solution can tolerate Byzantine failures. By fast, we mean that (i) validation and state reconstruction can be performed quickly, and (ii) peers do not have to pause operation in order to compute a snapshot. The key component of our solution is the design of a novel data structure targeted specifically to the state synchronization use case. This data structure, which we call an AVL* tree, is a Merklized AVL tree in which the tree leaves are organized into chunks. However, the assignment of nodes to chunks ensures that each chunk always contains a sub-tree of the system state. This enables batches of leaves to be securely and efficiently downloaded concurrently, while also permitting chunks to be verified for integrity using a compact proof. Finally, the structure of the tree is such that after a transaction is executed, a peer needs only to recompute the hash of the affected chunk, and propagate the hash up the tree to the root. It does not need to recompute the entire snapshot. Thus, during normal operation, the peer never has to pause transaction processing.

Incorporating leaf batching into a Merkle-ized AVL tree might seem straightforward. In reality, the problem introduces several challenges. First among these is identifying the proper invariants that must be maintained so that the integrity of the chunks can be checked independently. Second is designing the non-trivial changes to the AVL tree's operation methods to preserve the invariants. Third, during state transfer, peers will download different chunks in parallel, and can start reconstructing the tree. In general, inserting the data into a tree can result in different trees. To ensure correctness, we need to guarantee that the tree construction algorithm deterministically builds the same tree on different peers.

We have implemented the AVL* tree data structure and integrated it into the Tendermint blockchain middleware. We show that the time it takes for a new peer to join the blockchain using AVL* is halved, while at the same time tolerating Byzantine peers. Moreover, the improvements in the performance of state synchronization do not degrade the performance of steady execution. In fact, AVL* slightly increases throughput and reduces latency. Finally, we show that these improvements increase with the system size.

The rest of this paper is organized as follows. We first provide the necessary background (§2). We then describe the basic structure and operations of our AVL* tree (§3), followed by algorithms that enable fast state synchronization (§4). We then provide a thorough evaluation of AVL* trees, comparing to an AVL tree (§5). Finally, we present related work (§6) and conclude (§7). The Appendix contains a correctness argument for tree operations and state synchronization, and the detailed algorithms proposed in the paper.

2 Background

2.1 Blockchain

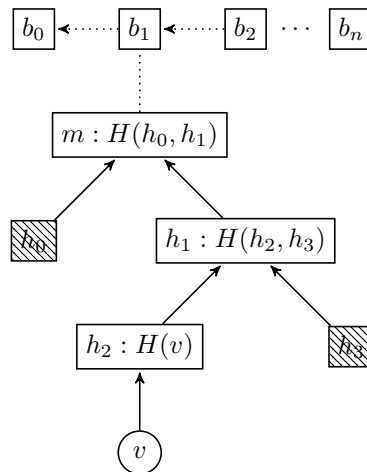
A blockchain is a distributed ledger, an append-only log of transactions implemented by geographically distributed peers. Clients submit transactions to the blockchain, which are appended to the log and executed by peers. Clients and peers may be *honest*, if they follow the protocol specification, or *malicious* (Byzantine), if nothing can be assumed about their behavior. The blockchain behaves correctly if a fraction of the peers, typically more than two-thirds, is honest [20].

The append-only log is structured as a linked-list of blocks, each block divided into a header and a body. The header contains, among other information, a cryptographic link to the previous block and a hash of the state. The body contains a list of transactions, each transaction cryptographically signed by the client that submitted it. Some blockchain systems (e.g., [24, 26]) allow the chain of blocks to momentarily fork, a situation in which multiple blocks are linked to a previous block. An alternative design is to ensure a total order on linked blocks (e.g., [15]). In this case, peers must agree on the next block to be appended by means of a byzantine fault-tolerant consensus protocol (e.g., [16, 21]).

2.2 Merkle trees

Generally, the state of a blockchain is stored locally by each peer in a key-value store structured as a Merkle-tree [25], or similar data structure (e.g., Merkle-Patricia-tree [26]). In a Merkle-tree, each leaf holds the value and its hash, and each inner node holds the hash of its children. The hash of the tree's root (Merkle-root) is stored in the blockchain header, which ensures its integrity. The key function of Merkle-trees is to provide succinct proofs of data integrity of any node of the tree. One can prove in logarithmic time and space that a leaf v belongs to the tree by providing the hash of the siblings of the tree nodes in the path from v to the Merkle-root m . In Figure 1, these hashes are h_0 and h_3 . One can verify that v belongs to the tree by recalculating h_2 , h_1 and m , and then verifying that the recalculated m is equal to the known trusted Merkle-root hash, stored in the blockchain.

The Merkle-tree of a blockchain is multi-version: a new tree is created for every new blockchain block, typically implemented with copy-on-write for performance reasons. When a new version of the tree is instantiated, the Merkle-root from the current version is copied to a new Merkle-root and made the Merkle-root of the new version. Modified nodes are duplicated and saved under the new tree. Although it is not common for peers to navigate on older



■ **Figure 1** Blockchain b_0, \dots, b_n and Merkle-proof $h_0;h_3$ of v . Hashes m , h_1 and h_2 are computed from v , h_0 , and h_3 , and hash function $H()$; v is valid if m matches the value stored in block b_1 .

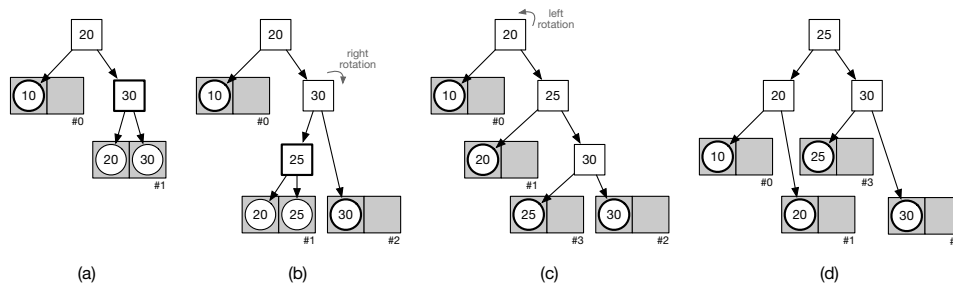
versions of the tree, it is required for certain inter-blockchain communication protocols [6], where peers serve as relayers between blockchains, allowing one blockchain to verify state proofs from another at a recent block.

2.3 State synchronization problem

A new (or recovering) blockchain peer needs to retrieve and execute transactions from all the blocks it has missed in order to catch up with operational peers. Merkle-trees enable fast catch up techniques based on traditional state machine replication [16]: Instead of re-executing all missing transactions, the peer retrieves a recent version of the Merkle-tree (i.e., a snapshot) from other peers and applies only the transactions in blocks that succeed the retrieved Merkle-tree. This technique has been implemented in popular blockchain clients (e.g., Geth [7], Ethereum’s main client).

The obvious question, though, is how should a peer download the snapshot? On the one hand, since a snapshot can be large, the download process can be accelerated by partitioning the tree into chunks, and downloading chunks from several peers in parallel. On the other hand, in blockchain systems, leaf sizes tend to be small (e.g., a hundred bytes), and the number of leaves quite large, creating significant I/O overhead if one were to serve each leaf independently. It makes sense to batch leaves together to reduce the I/O overhead.

Existing systems choose a fixed-size chunk and assign nodes to the chunk until it is full [2]. A natural implementation strategy is to perform a traversal of the tree (e.g., in depth-first order) and assign nodes to chunks. However, this approach fails to capitalize on the key property of Merkle-trees: that subtrees can be validated independently. Consequently, the snapshot cannot be validated until a peer has downloaded the complete state. Put another way, the peer must download all chunks before it can check if they are valid. Consequently, a single misbehaving peer serving an invalid chunk can initiate a type of “denial of service” attack, significantly prolonging the time it takes for a new peer to join the blockchain.



■ **Figure 2** Insertion and rotations in AVL* (white square: inner node; white circle: leaf; gray rectangle: chunk with chunk id; bold square/leaf: chunk root). (a) a balanced AVL tree, (b) an imbalanced AVL tree after the insertion of 25, (c) an imbalanced AVL tree after right rotation of previous tree at inner node 30, (d) a balanced AVL tree after left rotation of previous tree at pivot (inner node 20).

3 The AVL* chunked tree

The AVL* tree is a Merkle-ized, balanced binary search tree. As an AVL+ tree (§3.1), it preserves the same invariant: the height difference between its left and right children is at most 1. The key difference between the two is that the leaf nodes in an AVL* tree are organized into *chunks*, or batches (§3.2), and each AVL* chunk can be validated individually. The AVL* tree requires new data structures (§3.3), and algorithms for searching (§3.4), inserting (§3.5), and deleting (§3.6) nodes. Moreover, organizing the tree in chunks has implications on re-balancing (§3.7), multi-versioning (§3.8), and the correctness of the tree (discussed in the Appendix). A chunked AVL* tree can be downloaded in parallel during state synchronization, and independently checked for integrity.

3.1 AVL+ trees

While the techniques described in the paper could be used with any Merkle-ized trees, we focus the discussion on AVL trees [12, 5]. An AVL+ tree is a self-balanced ordered binary tree that implements a key-value storage API, where all values are stored in leaves and inner nodes store keys, used to keep the tree ordered. Leaves store the hash of values, and inner nodes the hash of their children and keys. As an immutable data structure, all updates are performed using copy-on-write. To add a key-value pair, the tree is traversed from its root until a leaf is found. Then, a new inner node is created, from which the found leaf and the new leaf (with the key-value to be included) will descend.

When the height difference between the left and right subtrees of an inner node is greater than one, the tree is said to be imbalanced. To re-balance the tree, one or two rotations are needed involving the lowest imbalanced subtree, whose root is called pivot node. Figures 2 (b) and (c) show imbalanced trees with inner node 20 as pivot. A left rotation at the pivot is enough if the subtree on the right of the pivot is higher than the subtree on the left of the pivot (Figure 2 (c)). But a right rotation at the right child of the pivot must happen first, if the left side of the pivot's right child is higher than the right side of the pivot's right child. This is what happens in Figure 2 (b) since the left side of inner node 30, on the right of pivot 20, is higher than its right side. The cases for right and left-right rotations are symmetric.

3.2 Chunks

A chunk is a set of nodes that are grouped together, and serves as the unit of access for the persistent store, communication, and integrity check. Grouping the nodes of the tree into batches facilitates parallel downloads. But, the assignment of nodes to chunks is important. If done incorrectly, it would not allow individual chunks to be checked for integrity. Each chunk has a *root* which is defined as follows:

► **Definition 1.** *The root of a chunk C , $root(C)$, is the lowest (i.e. deepest) node that has all the nodes in C as descendants.*

Given this definition, the important invariant preserved by AVL* when assigning nodes to chunks is as follows:

► **Property 1.** *For any chunks C_a and C_b , neither $root(C_a)$ is a descendant of $root(C_b)$ nor $root(C_b)$ is a descendant of $root(C_a)$.*

Property 1 ensures an overall minimal proof size for checking the integrity of a chunk. A client peer can check the integrity of chunk C with C and the proof that $root(C)$ is a valid node. If $root(C)$ is valid (see §2.2) then all data it stores is valid, including the hash of its subtree. The client peer then computes the hash that should be stored in $root(C)$ from the leaves in C all the way up to $root(C)$ and then checks whether the computed hash matches the hash stored in $root(C)$.

Trivially, we see that this property would be satisfied if we were to assign every node to the same chunk, or assign each node to its own chunk, begging the question of how big a chunk should be? There is a trade-off: if a chunk is too large, then we lose the ability to download multiple chunks concurrently, but if a chunk is too small, we lose the benefits of batching. The chunk size is a constant set when the tree is instantiated. In our experiments, we empirically evaluate different chunk sizes.

The key challenge in designing the AVL* tree is updating the insertion and delete algorithms of the AVL+ tree to respect this invariant, despite tree rotations.

3.3 Data structures

In an AVL* tree, inner nodes are stored in volatile memory only (DRAM); leaves are embedded in chunks, which are stored in volatile memory and in a persistent store. Table 1 details the structure of inner nodes, leaf nodes, and chunks.

An inner node contains a *key*, used to search the tree; a pointer to a *chunk*, if the key is the root of the chunk; pointers to *left* and *right* nodes (inner or leaf); the *height* of the node; a *hash*, discussed below; and a boolean *is_leaf* that asserts that a pointer to the node references an inner node.¹

A leaf node contains a key-value pair; a pointer to a *chunk*, if the key is the root of the chunk; an *i_node* pointer to an inner node, if the leaf's key is also an inner node; the *height* of the node, which is zero if the key is not stored as an inner node or the height of the inner node that stores the same key as the leaf; a *hash*; and a constant boolean *is_leaf*.

A chunk contains a unique chunk identifier *cid*, a number in $0..(m - 1)$, where m is the number of chunks; the *version* of the chunk, the number of leaves stored in the chunk, *size*; a pointer to the chunk *root*, which can be an inner node or a leaf; and a set *leaf* with all the leaves stored in the chunk. A chunk can store up to C_p leaves, a parameter of the system.

¹ For simplicity, we assume that pointers can reference either inner nodes or leaves.

■ **Table 1** The data structures used in the AVL*.

Inner node	
key	unique item identifier
chunk	pointer to chunk, if chunk root
left	pointer to the left node, inner or leaf
right	pointer to the right node, inner or leaf
height	the height of the node
hash	needed by the Merkle-ized tree
is_leaf	false
Leaf node	
key	unique item identifier
value	arbitrary data held in the node
chunk	pointer to chunk, if chunk root
i_node	pointer to matching inner node, if any
height	0 or i_node height (when sending chunk)
hash	needed by the Merkle-ized tree
is_leaf	true
Chunk	
cid	unique chunk id, starting in 0
version	version number of the chunk
size	number of leaves in the chunk
root	pointer to the chunk root, inner or leaf
leaf[0..($C_p - 1$)]	set of leaf nodes in the chunk

Hashes are computed after all changes in the tree have been performed, that is, the blockchain block the tree corresponds to has been fully processed. The hash of a leaf takes as input the key, value, height of the tree, and the chunk id and version, if the leaf is the chunk root. The hash of an inner node includes the key, the hashes of its children, and the chunk id and version, if the leaf is the chunk root.

3.4 Search

Searching for a key in the AVL* tree is just like a search in a regular AVL tree. The only difference is that if the search traverses a part of the tree that is not stored in main memory, then the corresponding chunk is read from disk and its entire subtree is stored in main memory.

3.5 Insertion

The insertion starts by searching down the tree for a leaf with a key that is immediately smaller or bigger than the key to be inserted. This will determine the position of the new leaf, either left or right of the found leaf, with the key-value element to be inserted. When searching down the tree, the root of a chunk is eventually found. From Property 1 (see §3.2), only one root chunk is traversed when inserting an element in the tree. If the root chunk points to a full chunk, before searching further, the chunk is split. The split, detailed next, does not change the structure of the tree but ensures that there is an available position in the chunk to accommodate the new leaf. To insert the new element, one inner node is created, pointing to the found leaf and the new leaf. When unrolling the recursion that leads to the insertion of an element, the height of every visited tree node is recomputed. If the subtree rooted at the visited node becomes imbalanced, a rotation is executed.

For example, in Figure 2 we add a node with key 25 in the tree depicted in (a), which has two chunks. Chunk #0 has one leaf and chunk #1 is full with two leaves. The insertion starts from the tree root and traverses to the right child at node 30, the chunk root of chunk

#1, which is at maximum capacity, and splits it, creating chunk #2. The algorithm continues traversing the tree until leaf 20 is reached. Since the new key is bigger than the leaf's key, a new inner node is added copying the value of the new key and rearranging the leaves accordingly, resulting in the tree (b) before balancing is done.

To split a chunk, we create two new chunks and run a depth-first search (DFS) from the left and right children of the chunk root to be split. Each call to DFS builds a new chunk that is assigned to the left and right children of the old chunk root. At the end of the split the old chunk is deallocated and its chunk root set to nil.

3.6 Deletion

Deleting a node from the AVL* tree is similar to deleting a node from an AVL tree. We discuss next the basic procedure and then two extensions that account for chunks.

To delete key k , we start by searching the tree to find a pivot node p such that (i) p 's left node is a leaf, or (ii) p is an inner node with key k . Then, there are four cases to consider:

- Case (a): p 's left child is the leaf with key k . In this scenario, p 's left node is deleted, p 's right node takes the place of the pivot, and the original pivot is deleted. This case happens when we want to remove key 10 in Figure 2 (a), where the pivot is inner node 20.
- Case (b): p 's right child is the leaf with key k . In this case, p is necessarily the inner node with the key to be deleted. Both p and its right child are deleted and p 's left child takes the place of the pivot. This case happens when we want to remove key 30 in Figure 2 (a), where the pivot is inner node 30.
- Case (c): The left child of p 's right child is the leaf with key k . The leaf with k is deleted, p is replaced with p 's right child, and the original pivot is deleted. For this case, consider the deletion of key 20 in Figure 2 (a), where the pivot is inner node 20.
- Case (d): Otherwise, we find the inner node x with the lowest value at the sub-tree on the right of p , delete x 's left leaf, replace pivot p with x , and delete the inner node p . We illustrate this case with the deletion of key 20 in Figure 2 (b), where the pivot is inner node 20 and x is inner node 25.

Akin to the AVL-tree deletion, when unrolling from the recursion that found pivot p , the nodes involved in the deletion have their heights updated and possibly rotated. Differently from the insertion, a deletion may involve rotations at all nodes in the way up to the root.

Deletion of a node in an AVL* tree differ from deletion in an AVL tree in two aspects. First, in cases (c) and (d) above, when an inner node x replaces a deleted inner node (i.e., the pivot). If x is a chunk root, then it will no longer be root, and the root of the chunk it referred to will be a node that descends from x .

Second, when deleting a leaf, it may happen that a chunk ends up with no leaves. This situation requires attention since the validity of a chunk is attested by the chunk root (discussed in §4), and an empty chunk has no root. To handle this case, when a chunk becomes empty, we take the chunk with the current largest unique id, assign to this chunk the id of the empty chunk, and decrement by one the number m of existing chunks (see §3.3). This ensures that every chunk with id in $0..(m - 1)$ has at least one node, and therefore a chunk root.

3.7 Re-balancing

If, as a result of an insertion or a deletion, there is a height difference between two child subtrees, then the parent tree must be re-balanced. Similar to the AVL tree, a rotation in an AVL* tree happens if the height difference from the children of a node is greater than

one. There can be four types of rotations: left, right-left, right, and left-right. Left and right rotations happen with a single rotation to the left or right, respectively. When rotating to the left on a pivot, the right child of the pivot takes the place of the pivot (called new pivot), the right child of the old pivot becomes the left child of the new pivot, and the left child of the new pivot becomes the old pivot. For the right-left rotation, first the right child of the pivot is rotated to the right and finally the pivot is rotated to the left. The right and left-right rotations are symmetrical to the cases explained before. The tree can be balanced with at most two rotations after an insertion.

If the rotation involves a chunk root, then there are two cases to consider. First, if the root of the chunk is the pivot of the rotation, then the node that takes the position of the pivot becomes the new chunk root. Second, if the pivot of the rotation is not the root of a chunk, but the node that takes the position of the pivot is the root of a chunk, then we split the chunk before doing the rotation. This is done to ensure Property 1. As a consequence, there will be extra splits even when a chunk is not full; in our experimental evaluation these splits amount for around 4% of the total number of splits.

Continuing from the example in Figure 2, the tree (b) is imbalanced after the insertion. Since the height of the root's right child is greater than the height of the root's left child, it triggers a left or a right-left rotation. Since the root's right child has a higher height on its left child than the right one we do a right-left rotation. First we rotate inner node 30 to the right and then rotate the inner node 20 (root) to the left. When rotating the inner node 30 to the right, we observe that its left child is a chunk root and split the chunk before continuing with the rotation. After this step, the tree is depicted in (c); observe that the tree has an extra chunk #3 created by the split and still is imbalanced. After the final rotation to the left, the tree is finally balanced as shown in (d).

3.8 Multi-versioning

An AVL* tree is multi-versioned, and a new version of the tree is created for every new blockchain block. For performance, a new tree is created using copy-on-write. Differently from an AVL+ tree, in the AVL* tree the unit of allocation is a chunk. This means that when a node is modified in the new version of the tree, the complete chunk that contains the node is copied. This chunk-based allocation suggests a tradeoff: small chunks reduce the overhead of creating the new tree, but increase the number of chunks that need to be propagated upon state synchronization. We experimentally evaluate this tradeoff in §5.

3.9 Correctness of tree operations

We initially argue that, in the absence of rotations, the insertion and deletion algorithms preserve Property 1.

In the case of an insertion, the property could only be invalidated when creating new chunks. When splitting a chunk, two new disjoint chunks are created and assigned to the left and right subtrees as the original chunk is extinguished. These steps do not violate Property 1 since they do not create descendants below or above each chunk root.

To see why deletion preserves Property 1, assume node x replaces the pivot, and x is root of its chunk. From the protocol, a descendant y of x becomes chunk root in place of x . Since x was chunk root, none of the nodes in its subtree are chunk root, and thus, no descendant of y is a chunk root.

We now argue that a right rotation preserves Property 1; the same reasoning applies for a left rotation. When rotating a node x to the right, x 's left child is assigned as the right child of x 's original left child. The only case a rotation would lead to a violation of Property 1 is

when there are chunk roots in x 's siblings. To deal with this case, before the assignment, we split the left chunk root in case of a right rotation (and right chunk root in case of a left rotation). After the split, the assigned node is a chunk root and can be safely moved to the opposite part of the tree.

4 Robust state synchronization

In general, there may be many ways to build a balanced binary tree from the same data. To be correct, we must ensure that all peers build the same tree. This is ensured if a peer collects all chunks that make up a tree, these chunks are valid, and the re-construction of the tree is deterministic. More formally, the tree reconstruction algorithm ensures the following two properties:

► **Property 2.** *Let T be the AVL* tree built by a honest peer after executing the n -th block of the blockchain, and $C_T = \{C_0, \dots, C_{m-1}\}$ the chunks of T . Upon receiving chunk C_k from a peer, a client peer can check whether C_k is valid (i.e., $C_k \in C_T$) before receiving any other chunks in C_T .*

► **Property 3.** *Let T and T' be two AVL* trees with the same nodes. If we build T' by inserting node by node following their order of height in T , then T and T' are isomorphic.*

We now describe a procedure to reconstruct the tree that satisfies Properties 2 and 3. To check that T is valid, the client needs the trusted Merkle-root hash of T and the number m of chunks in this tree. Both the Merkle-root hash and the number of chunks in the tree must be included in the block headers of the blockchain to ensure that they are trusted. In a blockchain, this information is available in the headers of a block that succeeds the n -th block (e.g., in the $(n + 1)$ -th block).

A client peer requests a chunk by its identifier (i.e., a value in $0..(m - 1)$) and receives as a response the requested chunk with the proof of inclusion of the chunk's root. The peer then rebuilds the subtree rooted at the chunk's root with all the nodes in the chunk. The procedure that checks the validity of the chunk (Property 2) proceeds in two steps. In the first step, the peer verifies the proof's validity by reconstructing the path from the chunk's root until the Merkle-root based on the hashes provided in the proof. The proof is valid if and only if the reconstructed Merkle-root matches the trusted one. In the second step, the peer recomputes the hashes of the subtree from the leaves up to the chunk root. The chunk is valid if the computed hash of the chunk root matches the hash provided, known to be valid from the first step.

To ensure that the client peer builds a proper subtree with the leaves in a chunk (Property 3), the peer first sorts all leaves by height. Recall from §3.3 that the height stored in a leaf is either 0, if the leaf's key is not an inner node, or the inner node's height in the tree. Then, the peer builds the tree by adding one node at a time, in descending order of the node height (i.e., root first). Multiple subtrees can be built in parallel to speed up the process. When the last chunk is built, the peer links all the chunk subtrees: The peer sorts all subtrees in descending order of their root node height, at which point the tree's root lies necessarily in the first position of the array. Each subtree is then added sequentially in the tree. When the last element is added the tree is complete. The correctness of this procedure is shown below. After the tree is built, the peer recomputes all the hashes.

4.1 Correctness state synchronization

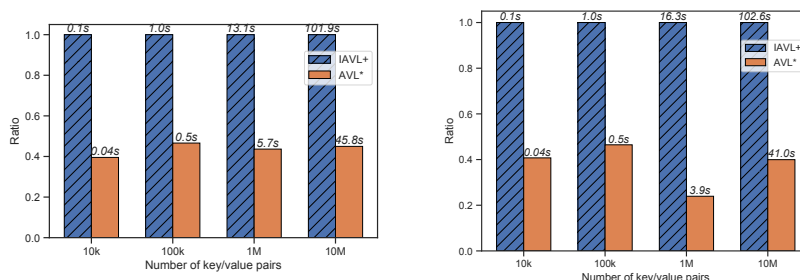
We initially consider Property 2. The first step of the protocol checks the validity of the chunk root using the proof of integrity of a single node of the tree. This follows immediately from the properties of Merkle trees (§2.2). A valid chunk root provides a trusted hash of its subtree. Then, in the second step, we compute the hashes from the leaves all the way up to the chunk root. If the trusted and the computed hash match, then the chunk is valid.

We now consider Property 3. Let h be the height of T , and $T(n)$ be a subtree of T that contains nodes from height h down to height n . The proof is by backwards induction on n .

Base step. Trivially true for $n = h$ since there is a single node with height h in subtrees $T(h)$ and $T'(h)$, the root.

Inductive step. We assume that $T(n+1)$ and $T'(n+1)$ are isomorphic and show that $T(n)$ and $T'(n)$ are isomorphic too, for $0 \leq n < h$. Let k be a node in $T(n)$ that is not in $T(n+1)$. Since T is a binary search tree, there is only one path in $T(n+1)$ that leads to k . From the inductive hypothesis, $T(n+1)$ and $T'(n+1)$ are isomorphic, thus, k will end up in the same location in $T'(n)$. Since $T(0) = T$ and $T'(0) = T'$, we conclude that T and T' are isomorphic.

5 Evaluation



(a) 10 peers.

(b) 80 peers.

■ **Figure 3** Time for a peer to recover from scratch, 100k chunks, varying number of key/value pairs.

5.1 Implementation and environment

To evaluate the behavior of our data structure and algorithms, we integrated them into Tendermint and conducted experiments under different conditions. Tendermint is a blockchain middleware that supports the replication of arbitrary applications. Tendermint provides applications with an AVL+ tree to manage state, called IAVL+. Peers can join the network by fetching a snapshot of the system state. To speed up the process, a peer can fetch a snapshot in chunks of fixed size.

All tests were conducted in a wide-area network (WAN) using Amazon’s Elastic Computing (EC2) platform. We evaluated the system with 10 peers (small setup) and 80 peers (large setup). Our large setup is a fair approximation of Cosmos/Tendermint’s current production system, with 125 peers. Peers were deployed in datacenters in seven Amazon regions: three datacenters in North America (Oregon, Ohio, Canada Central), two in Asia (Tokyo, Hong

8:12 Robust and Fast Blockchain State Synchronization

Kong), three in Europe (Paris, Frankfurt, and London), one in South America (Sao Paulo), one in the Middle East (Bahrain), and one in Africa (South Africa). We used t3.xlarge and r5.xlarge instances.

We used most of Tendermint’s default parameters, and set the mempool cache with 50k transactions, and block interval of 1 second for executions with 10 peers and 5 seconds for executions with 80 peers. These parameters led to the best results for throughput and latency for both the IAVL+ and the AVL* experiments. In the setup with 10 peers, each peer is connected to every other peer; in the setup with 80 peers, each peer is connected to 25 random peers.

We developed a key-value store application using Tendermint and benchmarked the application using the IAVL+ tree and the AVL* tree. Clients submit transactions that add new key-value pairs to the store. A key contains 20 bytes and a value contains 100 bytes, both generated randomly by clients. We evaluated the IAVL+ and the AVL* trees with chunks that can contain up to 10k and 100k key-value pairs, amounting to roughly 1MB and 10MB chunks, respectively. We considered executions in which clients include 10k, 100k, 1M and 10M key-value entries to the store.

5.2 State synchronization

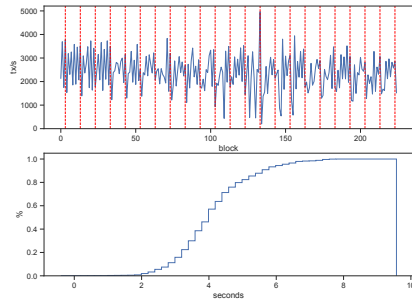
The first set of experiments evaluates the performance of the state synchronization operation. The main metric of concern is the time it takes to perform synchronization for a new peer joining the blockchain.

In these experiments, all peers but one are pre-initialized with a full tree. When the experiment begins, the new peer recovers the state by downloading chunks in parallel from the operational peers. We vary three parameters: (i) the size of the tree, (ii) the number of validators (10 and 80), and (iii) the size of the chunks (10k and 100k).

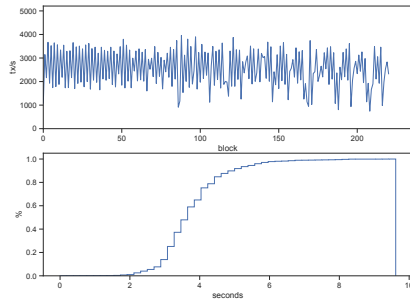
Figure 3 shows the state synchronization times for IAVL+ and AVL*. The results for 10k chunks and 100k chunks are similar; thus, we show results for 100k chunks only. We report the results as a ratio, with the absolute time printed at the top of each column. There are two important features to note. First, for both IAVL+ and AVL*, the synchronization time increases linearly with the size of the tree, and it does not depend on the size of the system. Second, the synchronization time for AVL* is roughly half the time required by IAVL+. This happens because in the AVL*, when a chunk is received, it can be validated individually and, if valid, the subtree it contains can be built before all chunks are received.

■ **Table 2** Throughput and latency for IAVL+ and AVL* in different configurations.

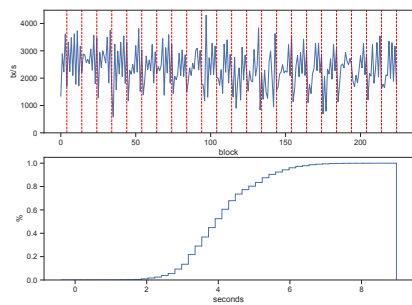
	10 peers							
	10k chunks				100k chunks			
	Throughput (tx/s)		Latency (s)		Throughput (tx/s)		Latency (s)	
	average	std.	average	std.	average	std.	average	std.
IAVL+	2373.86	764.03	4.18	1.1	2362.76	713.46	4.16	1.02
AVL*	2538.63	798.46	3.85	0.94	2491.69	802.94	3.96	0.99
variation	+7%		-8%		+5%		-5%	
	80 peers							
	10k chunks				100k chunks			
	Throughput (tx/s)		Latency (s)		Throughput (tx/s)		Latency (s)	
	average	std.	average	std.	average	std.	average	std.
IAVL+	893.28	218.33	8.63	2.27	892.82	235.59	8.71	2.6
AVL*	988.42	124.39	7.64	1.53	1001.25	131.8	7.54	1.55
variation	+11%		-11%		+12%		-13%	



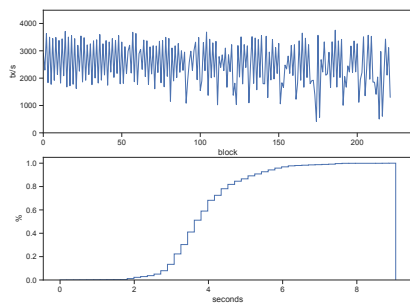
(a) IAVL+ with 10k chunks.



(b) AVL* with 10k chunks.



(c) IAVL+ with 100k chunks.



(d) AVL* with 100k chunks.

■ **Figure 4** Throughput and latency of transaction execution, 10 peers, 1M key/value pairs.

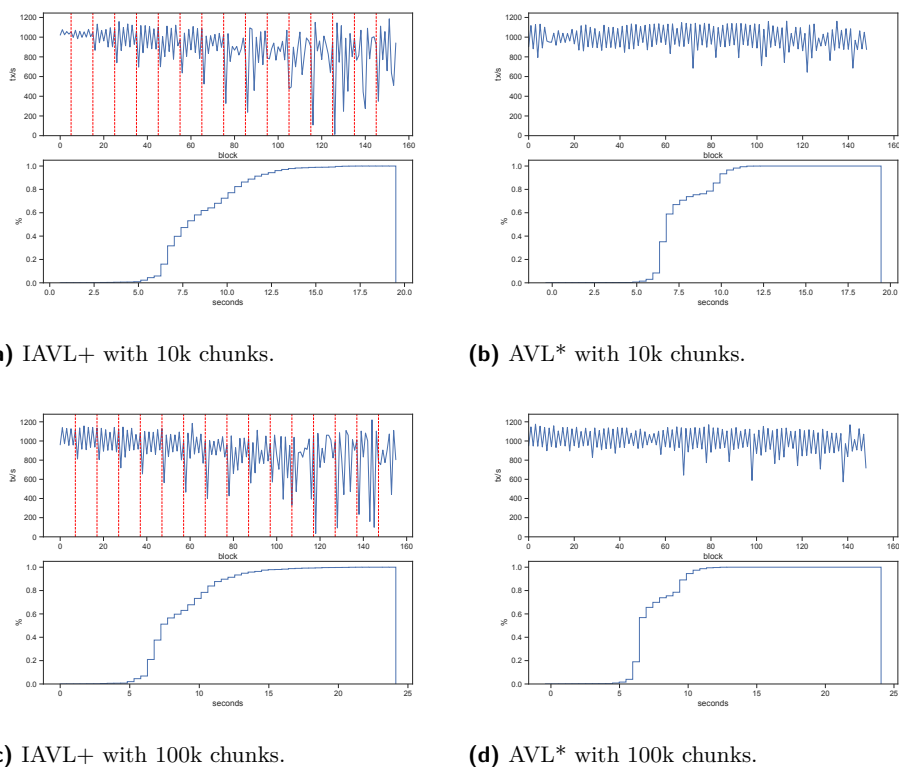
We do note one detail. Recall that the AVL* does not guarantee fixed-sized chunks. So, in these experiments, the AVL* chunks tend to be smaller than those made by snapshots in the IAVL+. Snapshots from the AVL* have around 16% more chunks than the IAVL+ tree. However, although there are more chunks in the application using the AVL* tree, the data stored in each chunk is increased, since it includes the proofs of inclusion for each chunk.

Overall, the state synchronization time when using the AVL* is on average 58% faster compared to the IAVL+ tree, despite having to download more chunks.

5.3 Steady-state operation

In the second set of experiments, we compare transaction throughput and latency of Tendermint using an AVL* and an IAVL+ tree. One distinctive aspect of AVL* is that peers do not need to periodically build state snapshots. To understand the overhead of IAVL+, we measure the time it takes to compute a snapshot. Finally, we assess AVL* space utilization.

In all the experiments, clients operate in a closed-loop, meaning a client only submits a new transaction after it receives the response for the previously submitted transaction. The client subscribes to the blockchain and delivers the blockchain blocks. Latency is computed as the time it takes for a transaction to be included in a block. Throughput is the number of transactions in a block divided by the time it took for the block to be ordered (i.e., interval between the current block and the previous one). In the experiments with the IAVL+ tree, snapshots are built every ten blocks.



■ **Figure 5** Throughput and latency of transaction execution, 80 peers, 1M key/value pairs.

5.3.1 Steady-state performance

We show the throughput as a time series and the latency CDF for Tendermint using both IAVL+ and AVL*. Figure 4 presents the results for a blockchain with 10 peers. Figure 4 (a) and (b) report the results for chunks of size 10k, and Figure 4 (c) and (d) report results for chunks of size 100k. Figure 5 shows the results for the same set of experiments when the number of peers is increased to 80.

The graphs show that using the AVL* tree results in more predictable performance. One of the reasons for the volatility of the IAVL+ tree is that there are periodic drops in performance that occur during a snapshot operation. In the graph, the dashed-red lines indicate the time that a snapshot occurs.

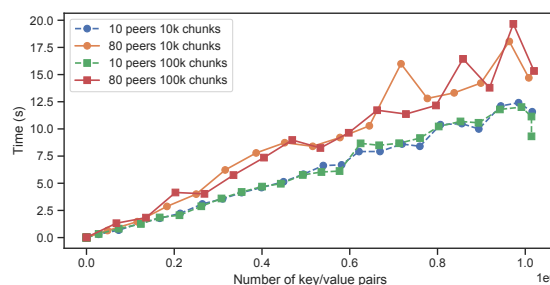
Although it is somewhat difficult to tell from the graphs, the performance of AVL* is not only more predictable, but it is also better, on average, than the IAVL+. Table 2 shows the mean values, the corresponding standard deviation, and the relative improvement of AVL* over IAVL+ for all these experiments.

In a blockchain with 10 validators, AVL* increases the throughput and reduces the latency by at least 5% when compared to the IAVL+ tree. These improvements increase with the size of the blockchain: with 80 validators, the improvements in throughput and latency are at least 11%.

5.3.2 Time for a snapshot

One of the major differences between IAVL+ and AVL* is that AVL* trees do not need to pause execution to compute a snapshot. This happens because in AVL* trees, chunks are an integral part of the tree data structure. In IAVL+, chunks are built from tree snapshots. Because snapshot computation has a significant impact on the IAVL+ performance, we wanted to quantify the overhead.

Figure 6 shows the time it takes to compute a snapshot with IAVL+ trees as the number of tree leaves (i.e., key-value pairs) is increased, varying from 0 to one million leaves. As the tree gets bigger, snapshots take more time to complete. That happens because each snapshot has to serialize the whole tree. Changing the chunk sizes does not have a significant impact on the experiment's performance.



■ **Figure 6** Time for an IAVL+ snapshot.

5.3.3 Space efficiency

The trade-off for using an AVL* is space efficiency, as the algorithm may fill chunks up to their capacity. We define space efficiency as the number of chunks in the AVL* divided by the ideal number of chunks (ceiling of elements divided by chunk size). For instance, if the space efficiency were 2, it would mean that the AVL* uses twice as much chunks as the ideal scenario. Note that when creating snapshot for the IAVL+, the space efficiency is always 1, because the serialization process always serializes the tree from scratch. We evaluated the space efficiency of the AVL* tree as we insert one million random keys into an empty AVL* tree. The space efficiency stabilizes at around 1.4 for random data, which we believe is an acceptable overhead.

5.4 State synchronization under attack

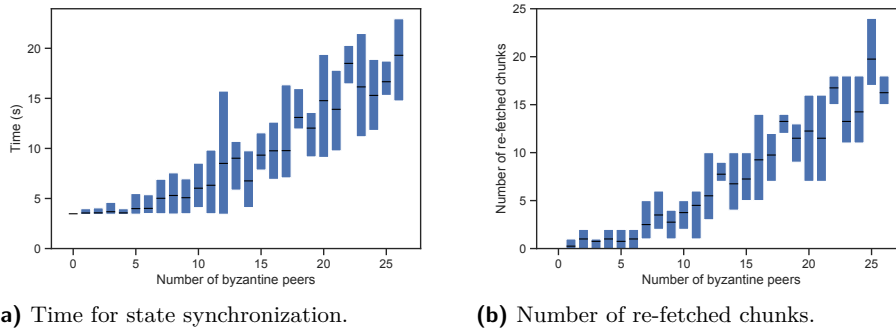
A key benefit of an AVL* tree over the IAVL+ tree is that it can gracefully recover from Byzantine or malicious behavior from peers. An IAVL+ cannot be checked for validity until the entire tree has been downloaded and reconstructed. If, after reconstructing the tree, the tree's root hash is different from the one in the trusted block header, then the client peer must refetch all chunks again. In contrast, the AVL* tree allows the client peer to detect invalid chunks easily, and remove misbehaving peers from their peer list.

To quantify the performance of AVL* in the presence of malicious peers, we again performed the state synchronization experiment, but introduced malicious peers that respond with invalid chunks. In these experiments, we used a fixed number of 80 peers and a tree with

one million entries. We varied the number of malicious peers from 1 to 26. We performed the experiment five times, and report mean synchronization time, as well as the maximums and minimums in bars and whiskers. Figure 7 shows the results.

In Figure 7 (a), we see that, as expected, the presence of malicious peers degrades the state synchronization time. The state synchronization time grows linearly with the number of malicious peers. Because a peer can respond to multiple requests for chunks in parallel before they are verified, the number of invalid chunks sent by peers can vary. In Figure 7 (b) we see that the number of chunks that need to be re-fetched is proportional to the state synchronization time.

From §5.2, we know that without any byzantine peers, it takes 16.3 seconds for a client peer using IAVL+ to complete state synchronization (with 80 peers, 100k chunks and 1M key-value pairs). With one byzantine peer, this time would double since the client peer would have to start from scratch. And even after retrying, there is no guarantee that the second attempt would succeed. Thus, a coordinated attack could substantially increase state sync time of IAVL+. AVL* outperforms IAVL+ even under attack by 20 byzantine peers.



■ **Figure 7** State sync with byzantine peers, 100k chunks, 1M key/value pairs.

6 Related work

Many systems for state machine replication with byzantine actors have addressed the problem of fast state synchronization without executing the entire transaction log. This is typically done by taking snapshots of the state, often called checkpoints. Such checkpoints can then be downloaded by new or recovering peers. While PBFT [17] proposed the use of a Merkle tree for its checkpoints, the Upright [18] and BFT-SMaRt [13] systems consider Merkle trees and copy-on-write semantics to be too invasive in general to the application developer. Upright outlines three simple approaches to state transfer [18], and BFT-SMaRt [13] describes a detailed Collaborative State Transfer protocol, where the full state is downloaded from a single peer and verified against hashes from other peers. In PBFT, a binary Merkle-tree is built at each checkpoint by partitioning the application state in 4KB pages. The pages are stored as leaves of the Merkle-tree using copy-on-write to only persist pages that have been modified since the previous checkpoint. This approach is not efficient to encode a key-value storage, since operations on the key space (e.g., searches) do not have logarithmic complexity.

Unlike SMR systems, which either consider Merkle-trees too expensive [18, 13], or construct them only for state synchronization at periodic checkpoints [17], many blockchain systems already use Merkle-ized data structures to store the state. The primary use case for such Merkle-trees is to facilitate light clients, who can efficiently query for particular leaves

of the tree and verify their integrity, without ever downloading the entire state or transaction history. The use of Merkle-trees for state synchronization has received much less attention, but as the state of blockchain systems grow, synchronizing it becomes more expensive.

In Geth [7], state synchronization is performed by requesting individual nodes of the tree. Peers do not have a way of knowing how long the state synchronization will last, because they do not know the total number of nodes [4]. Since the Merkle-tree is part of the consensus rules (i.e., Merkle-roots are stored in block headers), peers can verify that a received node from the tree is correct. However, given the small size of nodes (less than one KB), their randomized distribution in the underlying database, and the large size of the state (tens of GBs), requesting nodes individually leads to performance degradation for peers requesting and providing nodes. Batching nodes is a promising solution, however, it is challenging to batch nodes in a manner that can be securely verified by peers, and limits attacks on honest peers. For instance, in OpenEthereum [8], snapshots are taken periodically by serializing the entire state, and dividing it in large chunks. The hashes of each chunk are published in a manifest file. Since the manifest is not part of the consensus process, there is no way to verify that a chunk is correct before downloading all of them. Successfully completing the state synchronization in such a system thus depends on retrieving a correct manifest, which requires strong assumptions, for instance, that a particular peer can be trusted or that a majority of connected peers are correct. This is stronger than the usual assumption of a single (though unspecified) correct peer commonly used in blockchain systems.

Other blockchain systems have proposed to take snapshots of the Merkle-tree by periodically dividing it in chunks. In Codechain snapshots [9], chunks are built from nodes within a certain depth from a common root, and the hash of the snapshot is included in the block header so chunks can be verified incrementally by peers. Chunks may contain entire sub-trees, or may be limited to the upper nodes in a sub-tree. In Tendermint IAVL+ snapshots, tree nodes are serialized in order and assembled into chunks of a given size [1]. However, since Tendermint's block header does not currently support snapshot hashes, chunks cannot be incrementally verified by peers. Hence the need for a tree that incorporates chunking directly into its structure.

Motivated by their use in the blockchain context, numerous Merkle-tree designs have been proposed lately. TurboGeth [10] separates the key-value storage from the Merkle tree structure, and batches Merkle tree nodes in chunks to reduce the number of lookups during Merkle operations. This has the effect of greatly improving performance without changing the structure of the hash tree itself. Sparse Merkle-trees [19] have been adopted by other blockchain projects [11]. Recent advances in cryptography have even offered a glimpse into generalizations of Merkle trees called accumulators, which enable $O(1)$ proofs of set-membership and state-less blockchain clients [14]. While there has been a wide diversity of proposed and implemented tree designs, the AVL* is the only known tree to target both the light client and state synchronization use cases found in blockchain systems.

7 Conclusion

State synchronization is a significant bottleneck for blockchain-based systems. In this paper, we have presented a novel extension to Merkle-ized AVL+ trees that incorporates chunks. This extension allows peers to download portions of the state in parallel, and validate each chunk independently. We have also described an algorithm for deterministic tree reconstruction, ensuring that all peers have the same state. We have extensively tested our algorithms in a geo-distributed environment that show the benefits when reconstructing the state from snapshots and gracefully coping with byzantine failures.

References

- 1 Adr 053: State sync prototype. <https://github.com/tendermint/tendermint/blob/master/docs/architecture/adr-053-state-sync-prototype.md>.
- 2 Cosmos network. <https://cosmos.network/>.
- 3 Cosmos sdk. <https://github.com/cosmos/cosmos-sdk>.
- 4 Go ethereum faq. <https://geth.ethereum.org/docs/faq>.
- 5 Iavl+ implementation. <https://github.com/tendermint/iavl>.
- 6 The interblockchain communication protocol. <https://github.com/cosmos/ics/blob/master/spec.pdf>.
- 7 Official go implementation of the ethereum protocol. <https://github.com/ethereum/go-ethereum>.
- 8 Openethereum warpsync. <https://openethereum.github.io/wiki/Warp-Sync>.
- 9 Snapshot sync proposal. <https://research.codechain.io/t/snapshot-sync-proposal/21>.
- 10 Turbo-geth programmer's guide. https://github.com/ledgerwatch/turbo-geth/blob/master/docs/programmers_guide/guide.md.
- 11 Urkel tree implementation. <https://github.com/handshake-org/urkel>.
- 12 George M Adel'son-Vel'skii and Evgenii Mikhailovich Landis. An algorithm for organization of information. In *Doklady Akademii Nauk*, volume 146, pages 263–266. Russian Academy of Sciences, 1962.
- 13 Alysson Bessani, Marcel Santos, João Felix, Nuno Neves, and Miguel Correia. On the efficiency of durable state machine replication. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 169–180, 2013.
- 14 Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to iops and stateless blockchains. In *Annual International Cryptology Conference*, pages 561–586. Springer, 2019.
- 15 Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. *CoRR*, abs/1807.04938, 2018. [arXiv:1807.04938](https://arxiv.org/abs/1807.04938).
- 16 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, November 2002. doi:10.1145/571637.571640.
- 17 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.
- 18 Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright cluster services. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 277–290, 2009.
- 19 Rasmus Dahlberg, Tobias Pulls, and Roel Peeters. Efficient sparse merkle trees. In *Nordic Conference on Secure IT Systems*, pages 199–215. Springer, 2016.
- 20 Ittay Eyal and Emin Gün Sirer. Majority is not enough: bitcoin mining is vulnerable. *Commun. ACM*, 61(7):95–102, 2018.
- 21 Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzyva: Speculative byzantine fault tolerance. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, pages 45–58, 2007. doi:10.1145/1294261.1294267.
- 22 L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
- 23 Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378, 1987. doi:10.1007/3-540-48184-2_32.
- 24 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, bitcoin, 2008.
- 25 Michael Szydlo. Merkle tree traversal in log space and time. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 541–554. Springer, 2004.
- 26 Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

A Appendix: Algorithms

Algorithm 1 Insert.

```

1: next_cid := 0                                ▷ next chunk identifier
2: procedure INSERT(node, key, val)              ▷ node is a pointer to root
3:   return INSERT_AUX(node, key, val,  $\epsilon$ )    ▷ return ptr to new node
4: procedure INSERT_AUX(node, key, val, last_chunk)
5:   if node =  $\epsilon$  then                          ▷ if the tree is empty:
6:     chunk := NEW_CHUNK(next_cid)                ▷ first chunk
7:     node := INSERT_IN_CHUNK(chunk, key, val)
8:     node→chunk := chunk                          ▷ node becomes chunk root
9:     chunk→root := node                           ▷ ditto
10:    return node
11:   if node→chunk  $\neq \epsilon$  then                ▷ if node is the chunk root:
12:     c := node→chunk                             ▷ let c be this chunk
13:     if c→size =  $C_p$  then                       ▷ if c is a full chunk:
14:       SPLIT_CHUNK(node)                         ▷ split node (i.e., c's root)
15:   else
16:     c := last_chunk                             ▷ chunk root is a higher node
17:   if node→is_leaf then                          ▷ if node is a leaf:
18:     node_chunk := node→chunk                    ▷ keep its chunk, if any
19:     node→chunk :=  $\epsilon$                           ▷ node can't be chunk root
20:     if key < node→key then                       ▷ normal AVL left insert
21:       left := INSERT_IN_CHUNK(c, key, val)
22:       node := NEW_INNER(node→key, left, node, 1)
23:     else                                         ▷ normal AVL right insert
24:       right := INSERT_IN_CHUNK(c, key, val)
25:       node := NEW_INNER(key, node, right, 1)
26:     node→right→inner_node := node                ▷ leaf points to its inner
27:     node→chunk := node_chunk                    ▷ inner gets kept chunk
28:     if node→chunk  $\neq \epsilon$  then
29:       c→root := node
30:   else                                         ▷ if node is an inner node:
31:     if key < node→key then                       ▷ go down left or...
32:       node→left := INSERT_AUX(node→left, key, val, c)
33:     else                                         ▷ ...go down right...
34:       node→right := INSERT_AUX(node→right, key, val, c)
35:   UPDATE_HEIGHT(node)                           ▷ new height from children heights
36:   return BALANCE(node)                          ▷ if needed, rotate to keep balance
37: procedure UPDATE_HEIGHT(node)
38:   node→height :=                                ▷ height gets max children height plus one
39:   max(node→left→height, node→right→height) + 1
40: procedure BALANCE(node)
41:   h := node→left→height - node→right→height
42:   if h < -1 then                                ▷ if right branch is bigger than left branch:
43:     return ROTATE_RL(node)                       ▷ rotate [right] left
44:   if h > 1 then                                  ▷ if left branch is bigger than right branch:
45:     return ROTATE_LR(node)                       ▷ rotate [left] right
46:   return node

```

Algorithm 2 Auxiliary procedures.

```

1: procedure NEW_INNER(key, ln, rn, ht)                                ▷ create inner node
2:   new_innode := allocate new inner node
3:   new_innode→key := key
4:   new_innode→left := ln
5:   new_innode→right := rn
6:   new_innode→height := ht
7:   return new_innode

8: procedure NEW_CHUNK(cid)                                          ▷ create chunk
9:   new_c := allocate new chunk
10:  new_c→cid := cid
11:  new_c→size := 0
12:  new_c→root :=  $\epsilon$ 
13:  return new_c

14: procedure INSERT_IN_CHUNK(c, key, val)                            ▷ insert key,val
15:  c→leaf[c→size].key := key
16:  c→leaf[c→size].value := val
17:  node_addr := pointer to c→leaf[c→size]
18:  c→size := c→size + 1
19:  return node_addr

20: procedure DELETE_FROM_CHUNK(c, key)
21:   for node in c→leaf do
22:     if node.key = key then                                       ▷ found leaf
23:       SWAP(node, c→leaf[c→size])                                  ▷ swap with the last
24:       deallocate c→leaf[c→size]                                   ▷ free last leaf
25:       c→size := c→size - 1                                       ▷ decrement number of leaves
26:       if c→size = 0 then                                         ▷ chunk is empty
27:         next_cid := next_cid - 1                                   ▷ decrement number of chunks
28:         last_chunk := GET_CHUNK(next_cid)                         ▷ get chunk with highest id
29:         last_chunk→cid := c→cid                                   ▷ replace chunk's id
30:         deallocate c                                             ▷ free empty chunk

31: procedure SPLIT_CHUNK(node)                                       ▷ split chunk rooted at node
32:   new_c := NEW_CHUNK(node→chunk→cid)
33:   DFS(node→left, node, new_c)
34:   node→left→chunk := new_c
35:   new_c→root := node→left                                       ▷ assign left chunk
36:   next_cid := next_cid + 1
37:   new_c := NEW_CHUNK(next_cid)
38:   DFS(node→right, node, new_c)
39:   new_c→root := node→right                                       ▷ assign right chunk
40:   node→right→chunk := new_c
41:   deallocate node→chunk
42:   node→chunk :=  $\epsilon$                                            ▷ x is no longer chunk root
43:   return

44: procedure DFS(ptr, pnt, c)
45:   if ptr→is_leaf then
46:     c→leaf[c→size].key := ptr→key
47:     c→leaf[c→size].value := ptr→value
48:     c→leaf[c→size].i_node := ptr→i_node
49:     if ptr→key < pnt→key then                                     ▷ assign parent
50:       pnt→left := pointer to chunk→leaf[c→size]
51:     else
52:       pnt→right := pointer to chunk→leaf[c→size]
53:     c→size := c→size + 1                                         ▷ one more leaf in chunk
54:   else
55:     DFS(ptr→left, ptr, c)
56:     DFS(ptr→right, ptr, c)
57:   return

```

Algorithm 3 Delete.

```

1: procedure DELETE(node, key)
2:   if node.key = key and node.is_leaf then                                ▷ only one node
3:     DELETE_FROM_CHUNK(node.chunk, key)
4:     return  $\epsilon$ 
5:   return DELETE_AUX( $\epsilon$ , node, key,  $\epsilon$ )                                ▷ return ptr to new node
6: procedure DELETE_AUX(node, key, last_chunk)
7:   if node→chunk  $\neq \epsilon$  then                                           ▷ if node is the chunk root:
8:     c := node→chunk                                                       ▷ let c be this chunk
9:   else
10:    c := last_chunk                                                         ▷ chunk root is a higher node
11:  if node→left→key = key and node→left→is_leaf then
12:    if c =  $\epsilon$  then                                                       ▷ chunk is necessarily on the left
13:      c := node→left→chunk
14:      DELETE_FROM_CHUNK(c, key)
15:      promoted := node→right
16:      if node→chunk  $\neq \epsilon$  then
17:        promoted→chunk := node→chunk                                       ▷ node is a chunk root
18:        deallocate node                                                     ▷ no need for inner-node
19:      return promoted
20:  if node→key = key and node→right→is_leaf then                             ▷ chunk is necessarily on the right
21:    if c =  $\epsilon$  then
22:      c := node→right→chunk
23:      DELETE_FROM_CHUNK(c, key)
24:      promoted := node→left
25:      if node→chunk  $\neq \epsilon$  then
26:        promoted→chunk := node→chunk                                       ▷ node is a chunk root
27:        deallocate node                                                     ▷ no need for inner-node
28:      return promoted
29:  if node→key = key and node→right→left→is_leaf then
30:    if c =  $\epsilon$  then                                                       ▷ chunk is lower
31:      if node→right→chunk  $\neq \epsilon$  then                                       ▷ chunk on right
32:        c := node→right→chunk
33:        node→rightNode→rightNode→chunk = c
34:        node→chunk =  $\epsilon$ 
35:      else
36:        c := node→right→left→chunk                                         ▷ chunk on the left
37:        DELETE_FROM_CHUNK(c, key)
38:        aux := node→right
39:        node→key := aux→key
40:        node→right := aux→right
41:        deallocate aux
42:        return node
43:  if node→key = key then
44:    n, p := DELETE_LEAF(node→right, c)
45:    node→key = n→key
46:    node→right := p
47:    deallocate n
48:  else
49:    if key < node→key then
50:      node→left := DELETE_AUX(node→left, key, c)
51:    else
52:      node→right := DELETE_AUX(node→right, key, c)
53:  UPDATE_HEIGHT(node)                                                       ▷ new height from children heights
54:  return BALANCE(node)                                                     ▷ if needed, rotate to keep balance

```

Algorithm 4 Rotations.

```

1: procedure ROTATE_RL(node)                                ▷ normal AVL [right] left rotation
2:   rl_height := node→right→left→height
3:   rr_height := node→right→right→height
4:   if rl_height > rr_height then
5:     node→right := ROTATE_R(node→right)
6:     UPDATE_HEIGHT(node)
7:   return ROTATE_L(node)

8: procedure ROTATE_LR(node)                                ▷ normal AVL [left] right rotation
9:   ll_height := node→left→left→height
10:  lr_height := node→left→right→height
11:  if ll_height < lr_height then
12:    node→left := ROTATE_L(node→left)
13:    UPDATE_HEIGHT(node)
14:  return ROTATE_R(node)

15: procedure ROTATE_L(node)                                  ▷ node is the pivot
16:  if node→chunk ≠ ε then                                    ▷ if subtree rooted at node in a chunk:
17:    node→chunk→root := node→right
18:    node→right→chunk := node→chunk                            ▷ node's right child...
19:    node→chunk := ε                                           ▷ ...becomes new chunk root
20:  else                                                       ▷ else, rotation may involve two chunks
21:    if node→right→chunk ≠ ε then                               ▷ if r child is chunk root:
22:      SPLIT_CHUNK(node→right)
23:    new_pivot := node→right                                    ▷ rotation in three steps: one, ...
24:    node→right := new_pivot→left                               ▷ ...two, and...
25:    new_pivot→left := node                                     ▷ ...three!
26:    UPDATE_HEIGHT(node)                                       ▷ update moved node height
27:    UPDATE_HEIGHT(new_pivot)                                  ▷ update moved node height
28:  return new_pivot

29: procedure ROTATE_R(node)                                  ▷ node is the pivot
30:  if node→chunk ≠ ε then                                    ▷ if subtree rooted at node in a chunk:
31:    node→left→chunk := node→chunk                             ▷ node's left child...
32:    node→chunk := ε                                           ▷ ...becomes new chunk root
33:  else                                                       ▷ else, rotation may involve two chunks
34:    if node→left→chunk ≠ ε then                               ▷ if l child is chunk root:
35:      SPLIT_CHUNK(node→left)
36:    new_pivot := node→left                                    ▷ rotation in three steps: one, ...
37:    node→left := new_pivot→right                               ▷ ...two, and...
38:    new_pivot→right := node                                     ▷ ...three!
39:    UPDATE_HEIGHT(node)                                       ▷ update moved node height
40:    UPDATE_HEIGHT(new_pivot)                                  ▷ update moved node height
41:  return new_pivot

```

A Privacy-Preserving and Transparent Certification System for Digital Credentials

Rodrigo Q. Saramago ✉

University of Stavanger, Norway

Hein Meling ✉

University of Stavanger, Norway

Leander N. Jehl ✉

University of Stavanger, Norway

Abstract

A certification system is responsible for issuing digital credentials, which attest claims about a subject, e.g., an academic diploma. Such credentials are valuable for individuals and society, and widespread adoption requires a trusted certification system. Trust can be gained by being transparent when issuing and verifying digital credentials. However, there is a fundamental tradeoff between privacy and transparency. For instance, admitting a student to an academic program must preserve the student's privacy, i.e., the student's grades must not be revealed to unauthorized parties. At the same time, other applicants may demand transparency to ensure fairness in the admission process. Thus, building a certification system with the right balance between privacy and transparency is challenging.

This paper proposes a novel design for a certification system that provides sufficient transparency and preserves privacy through selective disclosure of claims such that authorized parties can verify them. Moreover, unauthorized parties can also verify the correctness of the certification process without compromising privacy. We achieve this using an incremental Merkle tree of cryptographic commitments to users' credentials. The commitments are added to the tree based on verifying zero-knowledge issuance proofs. Users store credentials off-chain and can prove the ownership and authenticity of credentials without revealing their commitments. Further, our approach enables users to prove statements about the credential's claims in zero-knowledge. Our design offers a cost-efficient solution, reducing the amount of linkable on-chain data by up to 79 % per credential compared to prior work, while maintaining transparency.

2012 ACM Subject Classification Security and privacy → Privacy-preserving protocols; Security and privacy → Pseudonymity, anonymity and untraceability; Information systems → Extraction, transformation and loading

Keywords and phrases verifiable credentials, privacy-preserving, zero-knowledge, blockchain

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2022.9

Supplementary Material *Software (Source Code)*: <https://github.com/r0qs/zkcertree>
archived at `swh:1:dir:491b29c170a47c1c5697bfe504075848d19a27bf`

Funding This work is partially funded by the BBChain and Credence projects under grants 274451 and 288126 from the Research Council of Norway.

1 Introduction

Smart contract-based issuing of digital credentials can increase transparency and thus help to detect fraud. This is especially important for academic credentials, which should be the result of a long learning and evaluation process. Fraud has been shown within educational institutions [13], but more importantly, fraudulent organizations, known as degree mills which give out credentials with no or dubious processes [2, 15]. Transparency standards of the evaluation and issuance process could significantly simplify the detection and blacklisting



© Rodrigo Q. Saramago, Hein Meling, and Leander N. Jehl;
licensed under Creative Commons License CC-BY 4.0

26th International Conference on Principles of Distributed Systems (OPODIS 2022).

Editors: Eshcar Hillel, Roberto Palmieri, and Etienne Rivière; Article No. 9; pp. 9:1–9:24

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

of such organizations. Further, unlike purely signature-based approaches, blockchain-based credentials allow long-term validity and revocation. However, in existing solutions [9, 29, 36], transparency of the issuance process is achieved only at the cost of users' privacy.

Saramago et al. [29] propose to issue credentials for individual certification steps, such as courses completed towards a degree. Their protocol allows authenticity checks of these individual credentials through metadata logged to a blockchain. Presenting individual certification steps provides transparency to the issuance process and enables verification of the issuance process, e.g., the time frame of the credentials' creation. Such properties guarantee that the issued credentials represent achievements acquired over a specific trusted time frame. Additionally, the metadata logged on-chain gives transparency of the issuers' overall processes, e.g., how many credentials are created during which periods. This transparency of the overall issuing process is useful for accreditation purposes and simplifies the detection of degree mills. They present an implementation of the above process using smart contracts running in an EVM-compatible blockchain. However, this implementation allows one to track all users' credentials and related activities based on the on-chain data. This is possible since the user's address (hash of its public key) is contained in the metadata for all its credentials. In the case of academic diplomas, this traceability may, for example, expose a user's failed or unfinished courses, even though irrelevant for their final certification.

This paper presents zkCert, a novel certification system design that allows transparency for an issuer's process while maintaining the users' privacy. zkCert enables users to commit to a set of credentials issued to them and prove the ownership and authenticity of such credentials in zero knowledge. Moreover, users can disclose a partial set of credentials to authorized parties for verification. Our approach reduces the amount of traceable information published on-chain while preserving the transparent and accountable certification process.

Besides the privacy and scalability concerns, a key challenge with implementing a certification system on an EVM-based platform is the high deployment, transaction, and storage costs. This demands that contracts are carefully designed to manage their resource use. Specifically, a contract must be deployed sparingly, it must utilize its storage efficiently, and transactions must be carefully managed.

We have implemented zkCert as a smart contract, called Notary, that acts on behalf of an issuer. Additionally, we implemented several applications based on zkSNARK circuits in the Circom [21] language to enable users to proof statements about their Notary issued credentials in zero-knowledge.

Our Notary contract allows educational institutions to automate a large portion of their issuing processes of digital credentials. National authorization agencies responsible for institutions can conduct audits without violating users' privacy. Users can apply to study programs or jobs by selectively disclosing only relevant credentials to the educational institution or employer, which would be designated as verifiers. Hence, a verifier could rank candidates based on their grades without actually knowing the grades. These are just some applications that zkCert can support via zkSNARK circuits and Notary issued credentials.

Our contributions include: (1) zkCert, a novel certification system design, combining an incremental Merkle tree for scalability with zkSNARKs for privacy, (2) a Notary smart contract implementation of zkCert, (3) a set of circuits demonstrating the capabilities of Notary issued credentials, and (4) our evaluation shows that zkCert is cost-efficient, scales well, and outperforms prior work despite stronger privacy guarantees.

2 Background

Credentials are certificates that attest a statement about a subject. The W3C data model defines a verifiable credential as a digital representation of a physical certificate in a cryptographically secure and machine-verifiable form [12]. For example, an academic diploma is a credential attesting the proficiency of a student and the authenticity of its digital representation can be verified through the use of digital signatures.

Compared to paper-based certificates, digital certificates offer a more secure and scalable alternative, being easy to create, revoke and verify, and further enabling a certification system that is less prone to human errors. The use of digital signatures, however, requires that a verifier can check if specific cryptographic keys belong to issuing authorities and individuals [10]. The security of the system relies on keeping the signing keys secret and compromised keys of issuing authorities may allow arbitrary creation of authentic credentials.

2.1 Blockchain Data Registries

Certificate authorities have a long history of security and privacy issues affecting trust in such systems, limiting mass adoption [18,22]. On the other hand, blockchain technologies provide a decentralized and transparent database infrastructure that can improve how certificates are issued and managed.

Data written on the blockchain is timestamped and tamper-resistant, forming a secure, consistent, and append-only log of transactions shared across peers in the blockchain network. Further, some blockchains can also execute code as part of a state transition, i.e., smart contracts. However, this level of transparency, where anyone can verify transactions, limits its use for applications that require a higher degree of privacy. For instance, popular blockchain technologies commonly offer pseudo-anonymity, where entities are represented by pseudonyms corresponding to public keys. Thus, naively using built-in transaction methods leaves a trail of all entities' activities. Consequently, the knowledge of identities with whom the pseudonyms are associated puts privacy at risk.

Hence, a fundamental tension exists between transparency and privacy in a blockchain. However, zero-knowledge proofs applied to blockchain state transitions [8,20] offer a promising solution to balance this tradeoff.

2.2 zkSNARKs

Zero-knowledge succinct, non-interactive arguments of knowledge (zkSNARK) [5] is a cryptographic proof construction where one can prove the knowledge of a secret while only revealing its validity and no other information. The proof satisfies an NP relation (nondeterministic polynomial time), and some zkSNARKs can prove any relation within a bounded-size arithmetic circuit [6,24]. Anyone can verify a proof, and the proof length and the verification time are sublinear in the circuit size. However, proof generation is expensive; typically, orders of magnitude slower than checking the relation directly [16,24].

To prove a program's computation using zkSNARKs, the program must be expressed as a set of quadratic constraints. Thus, in simple terms, proving a computation requires converting the program to a polynomial and evaluating it for a set of inputs. The first step is transforming a program from a domain-specific language to an arithmetic circuit [21].

An arithmetic circuit is a directed acyclic graph over a finite prime field such that the vertices are called gates, and the edges are called wires [6]. Wires can represent inputs or outputs. Input wires can be public or private, and outputs are always public. Private inputs

compose the witness, or the secret data, that must not be revealed to ensure zero knowledge. Gates take two inputs and perform multiplication or addition in the field [24]. Arithmetic circuits are closely related to circuits of logical gates but with arithmetic gates instead.

The circuit is then converted into a system of equations that express the arithmetic circuit satisfiability, i.e., a constraint system. The rank-1 constraint system (R1CS) is one such system and is used in many state-of-the-art zero-knowledge-proof systems [24]. However, since the number of constraints can be huge and, thus, inefficient for real programs, the R1CS representation is usually encoded as a single polynomial in the form of a Quadratic Arithmetic Program (QAP) [19]. QAP is a way to efficiently compute arithmetic programs over large finite fields using polynomials to represent the circuit constraints rather than numbers.

To be non-interactive, zero-knowledge protocols usually require a *trusted setup*. A trusted setup defines the parameters used to construct and verify proofs. These parameters are a set of random numbers that generate the proving and verifying keys in the system. Thus, if a single party knows all the parameters, that party could produce fraudulent proofs. One approach to avoid this problem is establishing a distributed ceremony with multiple unknown parties that derive their own private random data and collaborate through a Multi-Party Computation protocol to combine their shares for the final set of parameters [24]. As a result, the setup can be trusted, assuming at least one of the participating parties deletes their private data, as all parties would need to collude to put the construction at risk.

Some zkSNARK constructions require a trusted setup for each new program, i.e., the setup requires application-specific circuits [3]. However, recent works introduced the concept of *universal setup*. PLONK [16] is one such construction that allows multiple programs to reuse a single trusted setup within a bounded circuit size.

2.3 Incremental Merkle Tree

A Merkle tree is a (binary) tree data structure where leaf nodes are labeled with the hash of the data, and non-leaf nodes are the result of the hash of their children [23]. Merkle trees provide an efficient way to verify the membership of a leaf without requiring the verifier to know all the leaves by using *Merkle proofs*.

A Merkle proof consists of the leaf data to be proven, the tree's root hash, and a branch consisting of all siblings of the nodes in the path from the data to the root. For instance, suppose that a large database is stored as a Merkle tree and that the root is publicly known and trusted, e.g., digitally signed. A verifier can perform a lookup in such a database by verifying a Merkle proof of the data without requiring access to the whole database. Thus, Merkle trees can be used as tamper-resistant and authenticated data structures, allowing public audits.

However, naively updating a Merkle tree, e.g., adding a leaf, incur recomputing all intermediary nodes up to the new root. Since the size of Merkle trees can be huge, it is often impractical to reconstruct the whole tree every time a new leaf is added.

An *incremental Merkle tree* is an optimization that reduces time and space complexity by initializing the tree with empty nodes (filled with zeros) and keeping a partial Merkle tree for data updates [25]. A leaf node in such a tree is incrementally added (from left to right), updating the partial Merkle tree by replacing an empty node. Therefore, only the hashes of the nodes on the path from the new leaf to the root need to be recomputed.

2.4 Cryptographic Commitments

Cryptographic commitments allow one to commit to a value while keeping it secret from others and subsequently making the commitment public. Once committed, the scheme ensures that the value was not changed before disclosure [26].

Commitment schemes have two main properties, namely hiding and binding. Hiding ensures that no one can learn the value until it is revealed, even with access to the commitment, e.g., the hash of a preimage. When a commitment is opened, e.g., the preimage is revealed, the binding assures that it is infeasible to replace the value with something else.

Poseidon hash [17] is an elliptic curve hashing algorithm that compresses values into points on the curve, which can be produced and verified efficiently within arithmetic circuits, requiring fewer constraints per round. Poseidon is tailored to work on finite fields used in zero-knowledge proof systems and is currently one of the most efficient hashing functions for zero-knowledge applications. Poseidon can also be used for signature and commitment schemes where the knowledge of the committed value is proven in zero knowledge and is faster than previous methods like Pedersen commitments [17, 26].

2.5 Related Works

Semaphore [33] is a zero-knowledge protocol that allows users to vote by casting a signal, and only one, as a member of some group of entities without revealing their identity. Users in their system can participate in anonymous polls and Decentralized Autonomous Organizations (DAOs) and prove their participation on-chain. Their system provides a high level of privacy but was designed for a different use case, i.e., private voting, despite sharing similarities with the commitment-reveal scheme used in our work.

Tornado Cash [34] is another application that makes use of zero-knowledge proofs to provide anonymous payments through a mixing service. Users of tornado cash make deposits by committing to some amount of a cryptocurrency, e.g., ETH, and can later withdraw the deposited amount by providing a zkSNARK proof of the commitment within a Merkle tree and a nullifier. The nullifier is revealed in the withdrawal and used to prevent double-spending on a commitment. Tornado cash enables anonymity by grouping all deposits to the tornado contract in an *anonymity set*. The smaller the anonymity set, the worse the privacy of the system since it becomes easy to analyze the on-chain data and infer relations between deposits and withdraws. zkCert was also inspired by Tornado Cash adapting their solution to the verifiable credentials scenario.

Iden3 [31] is a Self-Sovereign Identity (SSI) platform that allows users to provide zero-knowledge proofs of membership and non-membership of claims on-chain. Their solution for verifiable credentials also allows on-chain revocations. In their system, identities sign the root of a claim tree of other identity creating a chain of signed trees that resembles web of trust. Each identity act as an issuer and claims are cryptographically proved and verified.

Certree [29] is a certification protocol that provides a transparent log of the issuer's actions by establishing an on-chain tree data structure of smart contracts and credentials. Each credential hash is stored on the blockchain as leaves of the tree-like structure and is identified by the subject's unique address. Their tree of smart contracts models the issuer's certification processes and defines authorization rules and scopes of each credential created by the issuer. However, all operations and identifiable metadata of credentials can be linked to the subject's address. This is a consequence of the proposed progressive construction of their credential tree. We propose zkCert, providing the same level of transparency as Certree, but significantly improve privacy due to our use of zero-knowledge proofs. zkCert

uses PLONK and provides a progressive certification process in the form of an incremental Merkle tree, enabling improved on-chain storage utilization and lower monetary costs to produce credentials.

3 System Model

We aim to design a certification system that delivers strong privacy guarantees to subjects while providing transparency of the entire process. As such, we assume a certification ecosystem consisting of four main parties defined by the W3C verifiable credentials data model [12]: issuers, subjects, verifiers, and a verifiable data registry. For convenience, we assume the subject to be the same as the holder defined by the W3C specification.

The purpose of a certification system is to issue credentials and provide sufficient information to third-party entities to verify the authenticity of credentials. A credential is often an official document that attests to a particular fact. In our model, a credential is a digital document that stores data as a claim, i.e., a statement about a subject.

Subjects, in a nutshell, are entities about which claims are made. Issuers are responsible for executing the certification process and asserting claims about subjects in the form of credentials. They are typically represented by a group of individuals that act on behalf of the issuer organization. A verifier is any entity that evaluates whether credential claims are authentic and meet specific criteria. Last but not least, a verifiable data registry facilitates the creation and validation of keys, identifiers, and other pertinent data [12].

3.1 System Properties

The goal of this work is to provide a certification system with the following properties.

Transparency. It should be able to demonstrate the process by which the issuer used to arrive at the conclusions stated in the claims [18].

Privacy. It should minimize exposure of linkable information to preserve users' privacy.

Integrity. It should prevent manipulation of a credential's content, once issued.

Trusted Timestamping. It should prevent manipulation of an issued credential's issuing time.

Authenticity. It should allow anyone to verify the authenticity of a presented credential.

Data and Service Availability. Subjects should have ownership of their credentials' data and should not depend on the issuer for verification. Credentials should remain valid and be verifiable even though the issuer becomes unavailable, e.g., goes out of business.

Revocability. Issuers and subjects should be able to revoke previously issued credentials, and anyone should be able to verify the change in a credential's status.

Agreement. Subjects can receive credentials from any issuer and must be able to accept or deny the issuance of a credential to him.

In Section 4.4 we briefly discuss how our system implements those properties.

3.2 System Assumptions

We assume that one or more cryptographic key pairs represent the entities in the system and that cryptographic primitives cannot be easily circumvented. System entities should be able to exchange sensitive data over a secure communication channel.

We assume that issuers can confirm the identity of subjects and that verifiers can identify issuers using mechanisms like decentralized identifiers [11]. Thus, we consider the existence of a registry for verifying the system entities' public keys, e.g., DPKI [1].

Moreover, we assume the existence of a blockchain-based verifiable data registry, with support for smart contracts, that gives write access to issuers and subjects and read-only access to verifiers. Additionally, the entities in the system are not able to trivially bypass the blockchain's security, timestamping, safety, and append-only properties.

3.3 Threat Model

Our system contains issuers, subjects, and verifiers, each of which may attack the system in different ways. A corrupt issuer may try to create or gain control over credentials with a correct timeframe. Creating credentials with a correct timeframe may take several years. Corrupted issuers may therefore attempt to alter or gain control over previously issued credentials. A corrupt subject may try to receive (from the issuer) or present (to a verifier) false credentials. The subject may also try to prevent revocation or falsely present revoked credentials as valid. Finally, a corrupt verifier may attempt to extract additional information about a subject's credentials from public information.

4 Privacy-Preserving Credentials

In this section, we describe our protocol, zkCert, for a privacy-preserving certification system. zkCert builds on a blockchain that supports smart contracts. The blockchain acts as a verifiable data registry for certification processes with strong timestamping properties. The certification processes are managed by a set of smart contracts, collectively named *Notary*, that encode the certification logic.

The contracts keep track of all issued credentials using an efficient Merkle tree data structure that can be augmented incrementally. The Merkle tree stores cryptographic commitments to subjects' credentials. Subjects can prove ownership of commitments in the tree without revealing which commitments they own. We cover this mechanism in detail in Section 4.2. First, however, we introduce the credential document format we use to prove, in zero knowledge, the existence of individual fields of a credential. This format allows individual fields to be selectively disclosed. Lastly, in Section 4.3 we describe how credentials can be verified and give some example use cases.

4.1 Credential Document File Format

An issuer creates a credential by grouping claims about a subject in a document. In zkCert, this document is a JSON file following an issuer-defined schema. The schema may be aligned with the W3C verifiable credentials data model [12], which specifies mechanisms to express secure and machine-verifiable digital credentials.

According to the W3C specification, a verifiable credential is composed of three core elements:

- (W3C1) Claims;
- (W3C2) Credential's metadata;
- (W3C3) Proof mechanisms.

Claims are a sequence of assertions made by an issuer about a subject. A credential's metadata describe properties that can be used for verification, as well as expiration dates and revocation mechanisms. Lastly, the proof mechanisms are used to verify a credential's authorship, prove its claims, and detect tampering.

The specification does not dictate a particular proof mechanism or file format. To facilitate a transparent and privacy-preserving certification process, we decouple the three elements of a verifiable credential into on-chain and off-chain data. We explain the rationale for this decoupling in the rest of the paper.

4.1.1 Merkle Tree of Claims

In our protocol, the claims (W3C1) are represented as fields of a JSON document. When issuing a credential, the document is augmented with a Merkle tree called *credential tree*. This tree is constructed deterministically from individual fields of the original document. The credential tree enables subjects to selectively disclose parts of the document and to prove that these parts belong to the original document using Merkle proofs. For instance, a student could reveal the field corresponding to his grades for a set of credentials without revealing other information about the credential besides the grades.

Our certification system requires that issuers use a predefined schema to create credentials. Otherwise, claims about specific fields may be deceiving. For instance, a credential could have fields for approval and revocation dates. In this scenario, a Merkle proof for the approval date alone would be insufficient to determine if the approval has been revoked.

The credential documents' fields form the leaves of the credential tree, constructed using Equation (1), where *key* is the SHA256 hash of the field's canonical property name modulo the SNARK prime field. The leaf *F* is the Poseidon hash of the concatenation of the *key*, the field's *value*, and a random *salt*.

$$F = \text{Poseidon}(\text{key} \parallel \text{value} \parallel \text{salt}) \quad (1)$$

As each field expresses a claim, the combination of fields can be used to compose an information graph about the subject. This, in turn, can be used to prove statements about the subject's credentials in zero knowledge. For example, the subject's grade for a course is greater than a given value. Figure 1 shows an example document and its credential tree.

The credential tree encoding enables more sophisticated computations over a credential's fields without revealing them. For example, we can prove that a set of credentials were issued in a specific period, and we can compute the weighted sum of a set of fields. These are relevant use cases for proving the duration of a bachelor's degree and a student's GPA. Appendix A provide details of how these examples are implemented in zkCert.

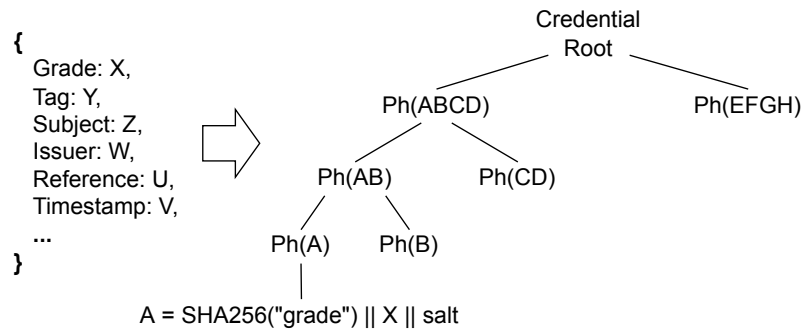


Figure 1 The encoding of a credential into a credential tree. *Ph* is the Poseidon hash function. Each field of the credential document is encoded as a leaf of the credential tree using Equation (1) over the field's key, value, and salt. In the example, *A* represents the encoding for the property "grade" and value "X".

4.2 Certification Tree

In this section we explain the certification tree, which is used to store credential commitments for all users. The certification tree is dynamically extended and keeps track of the issuer’s actions for transparency.

In our protocol, credentials hold claims data (W3C1) in the form of a credential tree, and is stored off-chain by the subject. Proofs (W3C3) are generated on-demand by the subject, which can be verified by on-chain and off-chain mechanisms during certification. Finally, the credentials’ metadata (W3C2) is stored on-chain in the Notary’s state.

The metadata is mainly used to verify and manage the credentials created by the Notary in the data registry. For instance, the metadata of a credential contains its current status, i.e., issued, revoked, or expired. Thus, the status can be changed or verified when the issuer revokes a credential without relying on the issuer’s servers.

Storing metadata on-chain increases the availability of verification services and can help to prevent censorship. However, it also imposes a challenge since it requires using a unique public identifier per credential to verify its metadata. Moreover, such identifiers could potentially be used to track credential owners.

To overcome this problem, we have designed a mechanism to prove the credential’s ownership in zero-knowledge. Our mechanism effectively breaks the link between subjects and their credentials, while maintaining the credential’s on-chain state for public verifiability and uses techniques that are already well-known in privacy preserving solutions using blockchains [20, 31, 33, 34].

The process of issuing credentials comprises two phases: Registration and Approval. A credential is considered valid if, and only if, registered by the issuer, approved by the subject, and it is not revoked or expired. Both phases use the PLONK [16] SNARK construction to generate the zkSNARK proofs in zkCert. Due to its upgradable and universal setup process, PLONK allows the issuer to create custom circuits up to a bounded size defined during trusted setup, enabling verification of a variety of credential’s information when used in conjunction with the credential and certification tree. We describe these phases in Sections 4.2.2 and 4.2.3. But first we explain the basic structure of the certification tree.

4.2.1 On-chain Credential Registry

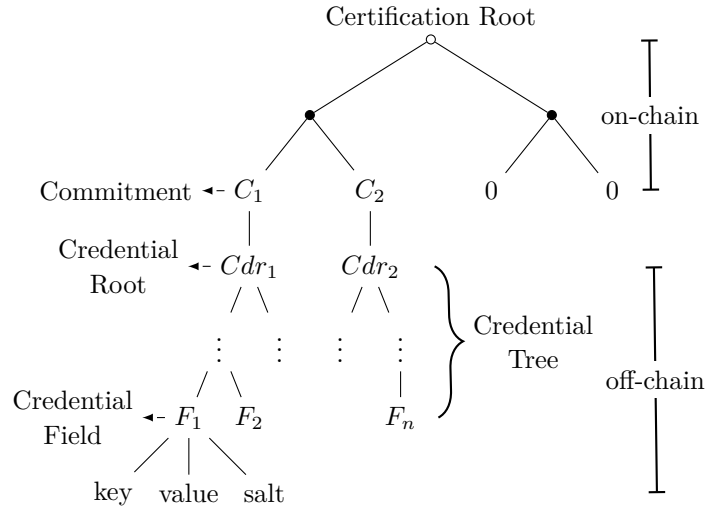
When creating a credential, the issuer records a cryptographic commitment to the credential in the Notary contract. Our protocol never reveals the preimage of the commitment. This is achieved by proving the commitment binding within the zero-knowledge circuit.

Commitments are created using the Poseidon hash function, which is efficient in zero-knowledge circuits [17]. Thus, the commitment C in Equation (2) is simply a point on the $bn128$ elliptic curve. The $bn128$ curve is natively supported on the EVM, enabling contracts to perform efficient zkSNARKs verification as well [28]. The *secret* is a random number, e.g., the subject’s EdDSA private key, and *sub* is the hash of the subject’s EdDSA public key.

$$C = \text{Poseidon}(cdr, secret, sub) \quad (2)$$

The credential root cdr is obtained by formatting the credential document as a Merkle tree, as described in Section 4.1, and retrieving the tree’s root. Subjects only disclose the hash of their credential root H_{cdr} when claiming ownership of credentials, i.e., approving the credential issuance, as described in Section 4.2.3. Such hash represents a unique ID for every issued credential.

The commitments are stored in the leaves of an incremental Merkle tree in the issuer’s Notary contract. We call this a *certification tree*, and its construction is shown in Figure 2. The tree is initialized with a fixed size in the Notary deployment, and all leaves are initially filled with zeros. Consequently, considering a perfect binary Merkle tree, the maximum number of credentials a Notary can register is 2^h , where h is the tree’s height. This tree construction results in a Merkle tree whose hash values are in the *bn128* elliptic curve, allowing efficient computation in the circuit [34].



■ **Figure 2** Incremental certification tree and its credentials’ trees.

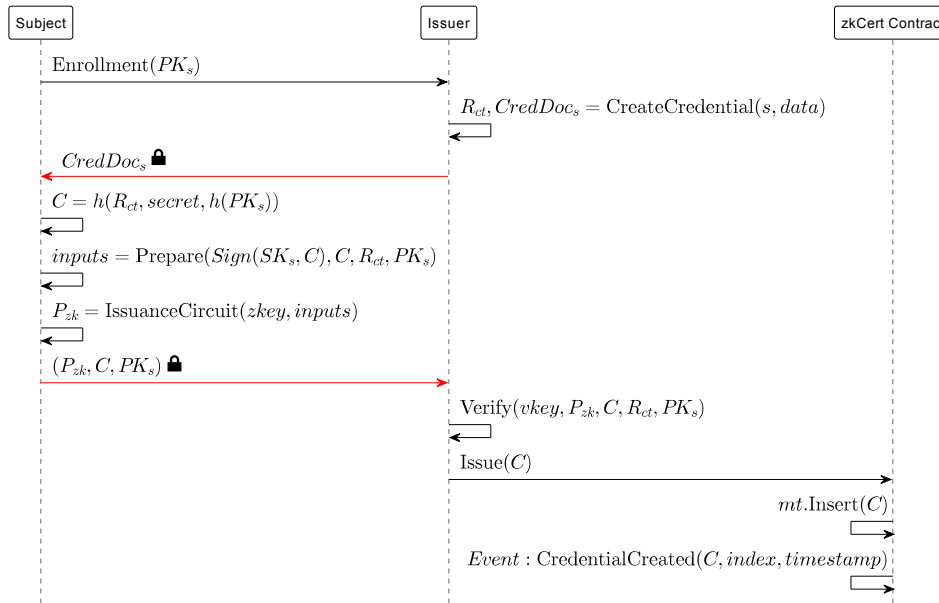
The incremental certification tree is responsible for keeping track of all credentials issued, preserving a progressive issuance process and a cryptographically immutable log of issuers’ actions while increasing privacy and optimizing the on-chain storage and execution cost.

For instance, a naive Merkle tree algorithm would require $O(2^h)$ time and space complexity to add new leaves. The incremental Merkle Tree, on the other hand, reduces the computational cost when adding a new leaf, i.e., replacing a zero leaf, requiring only $O(h)$ time and space complexity to update the Merkle tree’s root [25]. Leaves are added in the tree from the leftmost to the rightmost leaf, replacing the zero data with the credential commitment and updating the Merkle path to the new root.

Before creating any credential in zkCert, a trusted setup must be performed for the PLONK zkSNARK construction. The parties involved in the trusted setup in zkCert are arbitrary and defined by the issuer and trusted third parties, e.g., universities and national accreditation bodies. The subjects and verifiers do not need to participate in such a setup. Protocols like the scalable two-phase multi-party computation protocol MMORPG proposed by [7] can be used to generate the setup parameters. These parameters create the proving and verification keys used by zkCert to generate and verify proofs.

4.2.2 Registration Phase

To create a credential, an issuer first formats the claims about a subject in the form of a *credential tree*, as defined in Section 4.1. The issuer then sends the credential document to the subject over an encrypted channel. This phase is visualized in Figure 3. After verifying the credential’s claims, the subject computes the credential root and generates an *issuance*



■ **Figure 3** Registration phase: An issuer creates a credential to a subject and appends the commitment C to such credential in the Notary certification tree.

proof in zero knowledge using Circuit 1. The issuance proof attests to the issuer that the commitment is well-formed, i.e., used the expected credential root, and was created by the correct subject, i.e., the owner of the enrolled public key representing the subject’s identity. This proof is used to justify the inclusion of the commitment into the Notary certification tree by the issuer and can be used for internal audits.

■ **Circuit 1** Issuance proof.

```

1  IssuanceCircuit():
2  Public Inputs:
3      comm                                ▷ The credential commitment
4      cdr                                  ▷ The credential root
5      pk                                  ▷ The subject’s EdDSA public key
6  Private Inputs::
7      secret                              ▷ The subject’s secret
8      sig                                  ▷ The commitment signature
9  sub ← Subject(pk)
10 Assert: comm ≠ 0
11 c ← Commitment(cdr, sub, secret)
12 Verify: c = comm
13 v ← EdDSAPoseidonVerifier(c, pk, sig)
14 Verify: v = true
  
```

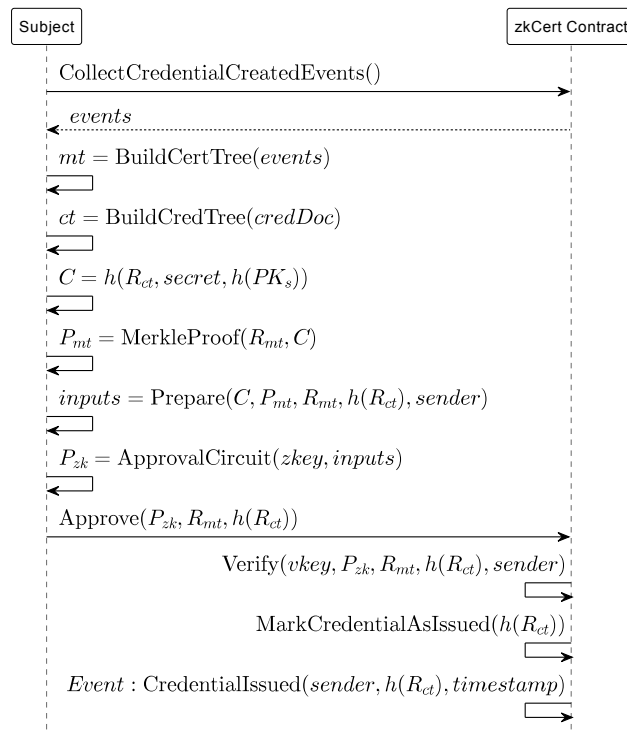
The subject then sends, through an off-chain channel, the issuance proof and the commitment to the issuer. Upon receiving the subject’s message, the issuer verifies the proof and, if valid, adds the commitment to the certification tree. We assume that the issuer can verify the subject’s identity using its public key, e.g., through a PKI.

It is important to note that the issuer is the one who creates a credential and consequently shares management rights, e.g., revocation. As a result, the issuer should already know the subject’s identity, the content of the credential, and its commitment. On the other hand, the subject creates the commitment using Equation (2) and is the credential owner. Consequently, nobody other than the subject can demonstrate knowledge of the preimage to such commitment without knowing the subject’s secret inputs. Nevertheless, as explained in Section 4.3, the commitments are not revealed during presentations to third parties, and the subject can selectively disclose the credential’s data as needed.

4.2.3 Approval Phase

The approval phase ensures that whoever was the subject of a registered credential agreed with its claims and could provide proof claiming its ownership. The phase represents a public consent of a subject within the credential without exposing the subject’s identity and enables verification by third parties of the authenticity of future credential presentations.

This phase is visualized in Figure 4. After the issuer registers a commitment, the correspondent subject can, at any later point in time, call the Notary contract to approve the credential by revealing the H_{cdr} .



■ **Figure 4** Approval phase: The subject reconstructs the certification tree locally based on the Notary events and generates a zkSNARK proof to approve a credential. The proof is then submitted to the contract by the sender, i.e., the contract’s caller, and upon verification, updates the credential’s state.

To generate an authenticity proof, a subject first retrieves the last certification root from the Notary contract and queries the necessary *CredentialCreated* events emitted by the contract in order to construct the Merkle proof of his commitments within the tree. This

process is needed since the certification tree is constantly being updated, and to be able to produce proofs for the most recent root, new data need to be retrieved from the blockchain. Of course, the data can be cached, and the issuer or third parties can provide services to speed up the data retrieval.

The subject generates an approval proof by executing Circuit 2, verified on-chain by the Notary contract. It attests that the subject not only has knowledge of a preimage to a commitment within the tree, but also knows the Merkle path to such commitment without revealing it. This attestation breaks the link between a credential's commitment from its presentation to an external verifier, which only needs to know the H_{cdr} to verify the status of the credential on-chain.

■ Circuit 2 Approval proof.

```

1 ApprovalCircuit():
2   Public Inputs:
3     ctr                                ▷ The certification tree root
4     Hcdr                               ▷ The hash of the credential root
5   Private Inputs::
6     cdr                                  ▷ The credential root
7     subject                              ▷ The subject ID
8     secret                               ▷ The subject's secret
9     MtProofcomm                         ▷ The commitment's merkle proof
10   $c \leftarrow \text{Commitment}(cdr, subject, secret)$ 
11  Verify:  $\text{Poseidon}(cdr) = H_{cdr}$ 
12   $r \leftarrow \text{VerifyMerkleProof}(c, MtProof_{comm})$ 
13  Verify:  $r = ctr$ 

```

The contract records the H_{cdr} hashes already approved, so a credential cannot be issued twice by the same Notary contract. The existence of the H_{cdr} on-chain also eases the management of issued credentials. Further, the proof contains a *sender* address of an entity authorized by the subject to call the contract in his behalf. This allows subjects to delegate the approval to third parties, which could also be the issuer. Note, however, that the issuer or any delegated third party *cannot* produce a valid proof on behalf of the subject, since only the subject knows the entire preimage data and his private key.

4.3 Verifying Credentials

The W3C verifiable credentials data model [12] defines a verifiable presentation as a mechanism to express data derived from credentials in which the authorship can be verifiable. In our protocol, a subject can prove the authorship of a credential without revealing all credentials they own from a specific issuer. Combined with the credential tree format described in Section 4.1, it is also possible to prove the existence of a field in the credential within the certification tree root and, consequently, the existence of a commitment to such credential on-chain. As writing data on-chain requires signing a transaction, such proofs enable authenticity checks of issued credentials while keeping the underlying data private.

As mentioned in Section 4.2, zkCert relies on the PLONK zkSNARK construction. Thus after the issuer has performed the setup, it can make available the proving and verification keys, and the supported circuits, to users and developers. We provide a set of circuits for the use cases covered in this paper, along with our companion source code¹.

Our design combines on-chain authenticity and validity checks, performed by Notary contracts, with off-chain claims verification based on zero-knowledge proofs provided during the presentation of credentials. A third-party verifier can, without any interaction with the issuer, check:

- (a) The authenticity of credentials and its current status (e.g. revocation, expiration date);
- (b) If the values in the credential's fields met certain conditional criteria (e.g. all grades are greater than B);
- (c) The result of an arithmetic computation over a set of credentials (e.g. the GPA of a student);
- (d) The time-frame in which the presented credentials were issued (e.g. if all credentials of a subject were issued in a three years period);

Note that the items listed above do not constitute an exhaustive list, and all can be verified in zero-knowledge, thus without exposing any underlying private values.

Item (a) is verified by requesting zero-knowledge proof for the credentials being presented and cross-checking the hashes H_{cdr} of these credentials in the Notary regarding their current status. It is important to note that such verification reveals the H_{cdr} to the verifier since it is required to check the credential status, e.g., revocation, but an external observer to the blockchain cannot track such activity. Nevertheless, the correspondent commitment and credential's data remain secret to the verifier, and it is infeasible to recover such information from a given H_{cdr} . However, it may be possible to de-anonymize users with statistical analysis of on-chain transaction patterns. The complete flow of this process is included in Appendix A.

Further, due to the way that our credentials are built (see Figure 2), Item (a) can also be extended to verify the authenticity of specific fields of credentials or perform simple arithmetic computation over it. In Appendix A we give an example of a circuit that computes the weighted sum of a set of authenticated integers using our approach.

Subjects can also select specific fields of a set of credential trees they own to present to a verifier in zero-knowledge, this enables checks as described in Item (b). However, a naive implementation would require that the subject create one Merkle proof for each field to be proven.

To reduce the number of Merkle proofs required to proof Items (b)-(d), we added support for Merkle multi-proofs [27], which reduces the number of Merkle proofs required to prove the inclusion of multiple leaves to a single proof. This improvement allows the creation of Merkle multi-proofs for multiple credentials and multiple fields of credentials. We also provide a circuit implementation in Circom [21] of the Merkle multi-proof algorithm, enabling membership proofs in zero knowledge.

4.4 Design Properties Analysis

As mentioned in Section 4, zkCert uses the blockchain to guarantee a trusted timestamping of issued credentials. Timestamp information is added to the credential JSON document and consequently to its credential tree representation. This information can be verified off-chain

¹ <https://github.com/r0qs/zkcertree>

at the circuit level, as exemplified in Appendix A.4, and cross-checked against its on-chain registration and approval metadata in the certification tree, considering a time threshold. This characteristic can help to detect fraudulent patterns in the issuance process.

The blockchain also ensures integrity, authenticity, and service availability while guaranteeing that the metadata stored on-chain are valid and always accessible for verification. Although we do not address the off-chain storage of credentials in this work but assume its existence, we acknowledge that it is an essential part of a certification system. The use of content-addressing storage like IPFS [4] or Swarm [35] can improve censorship resistance and availability of proving data as well, i.e., credential documents.

The certification tree provides transparency in the certification process through its incremental tamper-resistant event log, and combined with identity mechanisms like DIDs [11] and DPKI [1], can ensure the authenticity of an issuer's actions and issued credentials. It can further provide non-repudiation through subject approvals, helping auditing processes.

The approval phase in Section 4.2.3 may not be suitable for all use cases, e.g., issuance of criminal records. However, it is a feature of our protocol to register a public agreement of all signing parties of a certificate, i.e., the issuer and the subject. The approval phase can safely be omitted when the registration phase is sufficient to meet the use case's requirements.

The certification tree also enables the revocation of credentials without relying on a central server. The revocation permissions can be configured through an access control list of authorized issuer's personnel.

Finally, using zkSNARK to prove statements about credentials adds a privacy layer to our certification system, preventing unauthorized entities monitoring blockchain events from tracking user activities, while preserving the transparency properties of the protocol.

5 Evaluation

In this section, we present our evaluation of zkCert. We first evaluate the monetary cost of issuing diplomas on various blockchains. Then we assess the performance of proving and verification time of the zkSNARK construction.

For the monetary cost analysis, we compare the cost of the operations performed by zkCert with Certree's costs [29]. The cost is expressed in dollars instead of gas units [14] since it reflects the storage and computational cost of smart contracts operations in the analyzed blockchains. We also analyzed the on-chain storage usage of both approaches. Our results are shown in Table 1 and include updated costs for Certree based on recent exchange rates for three mainstream blockchains: Ethereum [14], Avalanche [30] and Polygon [32].

An *Exam cert* in Table 1 represents one issued credential and does not consider the cost or storage space of the contract deployed. Other lines show complete cost for a certificate composed of multiple credentials. For instance, a course certificate (*Course cert*) in our example is composed of five exam certificates issued to the same subject. As in [29], we assume that courses correspond to 10 ECTS and a semester to 30 ECTS.

The table includes both costs of issuing and approving credentials. In Certree [29], the approval cost must be borne by the subject, the credential's owner. In zkCert, however, this cost can be offloaded to a third party, e.g., the issuer. This is possible since approval is not linked to the subject's on-chain identity; instead, the subject's approval is linked via a zero-knowledge proof. Further, in Certree, a contract must be deployed for each course, adding progressive deployment costs, while in zkCert, the Notary contract is only deployed once. Thus, zkCert adds a fixed initial storage cost of 8.39 Kb for initializing the incremental certification tree as part of the Notary contract. As described in Section 4.2, issuers can update the Notary contract incrementally as new credentials are issued. It is

■ **Table 1** Monetary cost and on-chain storage utilization comparison of zkCert and Certree for a hypothetical scenario issuing Bachelor’s and Master’s credentials for one student.

Protocol	ECTS	Cred. Issued	Contracts Deployed	On-chain Storage	Cost ETH/USD*	Cost AVAX/USD [‡]	Cost MATIC/USD [†]	
Certree	Exam cert		1	0	321 b	20.01	0.22	0.01
	Course cert	10	5	1	1.60 Kb	297.82	3.27	0.15
	Semester cert	30	16	3	5.33 Kb	1106.23	12.15	0.56
	Bachelor’s diploma	180	97	18	32.67 Kb	6881.09	75.59	3.49
	Master’s diploma	120	65	12	21.89 Kb	4658.32	51.17	2.36
zkCert	Exam cert		1	0	66 b	39.18	0.43	0.02
	Course cert	10	5	1	8.72 Kb	195.92	2.15	0.10
	Semester cert	30	16	1	9.38 Kb	587.77	6.46	0.30
	Bachelor’s diploma	180	97	1	22.72 Kb	3618.83	39.75	1.84
	Master’s diploma	120	65	1	20.74 Kb	2443.30	26.84	1.24

* Gas price = 33 Gwei and ETH 1 = 1634.53 USD at 2022-08-20.

[‡] Gas price = 26 nAVAX and AVAX = 22.79 USD at 2022-08-20.

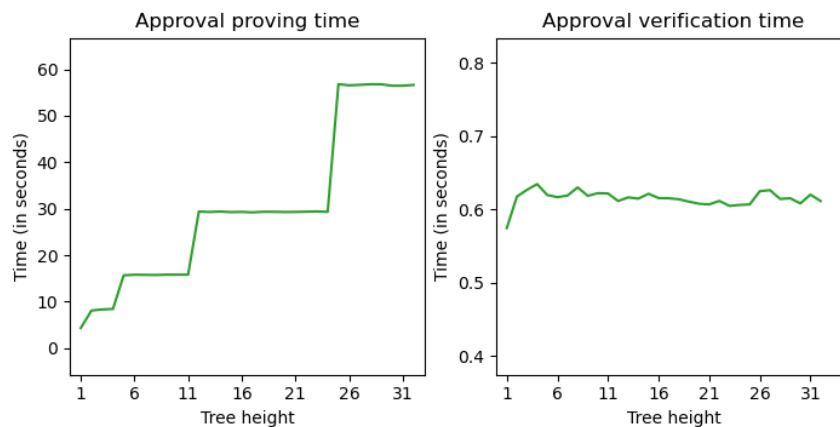
[†] Gas price = 34.4 Gwei and MATIC = 0.796 USD at 2022-08-20.

also worth noting that the cost of zkCert considers a partial binary Merkle tree of height up to 31. Table 1 shows the combined cost of deploying contracts and creating and approving credentials.

Table 1 shows that the cost of issuing a single credential increased by almost 49 % with zkCert compared to Certree. This increase is due to adding a new leaf in the Notary’s tree during registration and on-chain verification of a zkSNARK proof during approval. However, this initial cost is amortized as more credentials are created since no new contracts must be deployed. In Certree, however, each new leaf or intermediary node in their tree adds a fixed contract deployment cost. For instance, to issue a Bachelor’s diploma over three years, there is an aggregate cost reduction of 47 % compared to Certree. Further, using zero-knowledge proofs, we reduced the on-chain space requirement per credential by more than 79 %.

We next evaluate the computation time to generate an approval proof and the corresponding time to verify an approval. The approval circuit has 3379 constraints. The experiment run in a machine with Archlinux installed with 16 GB RAM, processor Intel Core i7-8550U 1.80 GHz and 500 GB SSD.

As shown in Figure 5, the time to generate an approval proof follows a step-function over the tree’s height. The tree’s maximum size, decided during deployment, dictates the time it takes to generate an approval. We observe, however, that the generation and verification time is unaffected by the number of credentials already issued. Thus, the proving time using a tree of height 21, which can hold 2^{21} elements, will be around 30 seconds, irrespective of the number of elements in the tree. This is because generating a Merkle proof for an incremental Merkle tree has linear time complexity [25]. The verification time of the approval proof is polylogarithmic in the circuit size [16].



■ **Figure 5** Time to generate and verify approval proofs per the certification tree's height.

6 Conclusion

We have presented zkCert, a certification system for digital credentials that ensures transparency of the issuer's processes and preserves privacy for its users. zkCert is implemented using smart contracts, where an incremental Merkle tree enables scalability, and zkSNARKs provide privacy to users.

Our evaluation shows that zkCert provide significant improvement over prior art, both in on-chain storage utilization and lower monetary costs to produce credentials over time. For deployments that can support up to 2^{31} credentials, it takes less than a minute to generate a credential while maintaining transparency.

Although we focused our work on a specific use case of academic credentials, our design can be used in various use cases that can benefit from a transparent and progressive history of issuance activities while ensuring a high level of privacy for their users, e.g., driver's licenses, supply chain, and fair trade.

References

- 1 Christopher Allen, Arthur Brock, Vitalik Buterin, Jon Callas, Duke Dorje, Christian Lundkvist, Pavel Kravchenko, Jude Nelson, Drummond Reed, Markus Sabadello, Greg Slepak, Noah Thorp, and Harlan T. Wood. Decentralized public key infrastructure, 2015. URL: <https://www.weboftrust.info/downloads/dpki.pdf>.
- 2 Philip G. Altbach, Liz Reisberg, and Laura E. Rumbley. Trends in global higher education: Tracking an academic revolution. Technical report, United Nations Educational, Scientific and Cultural Organization (UNESCO), January 2009. URL: https://www.cep.edu.rs/public/Altbach,_Reisberg,_Rumbley_Tracking_an_Academic_Revolution,_UNESCO_2009.pdf.
- 3 Stephanie Bayer and Jens Groth. Efficient zero-knowledge argument for correctness of a shuffle. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, pages 263–280, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
- 4 Juan Benet. IPFS - content addressed, versioned, P2P file system. *CoRR*, abs/1407.3561, 2014. [arXiv:1407.3561](https://arxiv.org/abs/1407.3561).
- 5 Nir Bitansky, Ran Canetti, Alessandro Chiesa, Shafi Goldwasser, Huijia Lin, Aviad Rubinfeld, and Eran Tromer. The hunting of the snark. *Cryptology ePrint Archive*, Paper 2014/580, 2014. URL: <https://eprint.iacr.org/2014/580>.
- 6 Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Jens Groth, and Christophe Petit. Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, pages 327–357, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

- 7 Sean Bowe, Ariel Gabizon, and Ian Miers. Scalable multi-party computation for zk-snark parameters in the random beacon model. Cryptology ePrint Archive, Paper 2017/1050, 2017. URL: <https://eprint.iacr.org/2017/1050>.
- 8 Vitalik Buterin. An approximate introduction to how zk-snarks are possible. Accessed: September 2022. URL: <https://vitalik.ca/general/2021/01/26/snarks.html>.
- 9 Guang Chen, Bing Xu, Manli Lu, and Nian-Shing Chen. Exploring blockchain technology and its potential applications for education. *Smart Learning Environments*, 5, December 2018. doi:10.1186/s40561-017-0050-x.
- 10 Santosh Chokhani, Warwick Ford, Randy V. Sabett, Charles (Chas) R. Merrill, and Stephen S. Wu. Internet x.509 public key infrastructure certificate policy and certification practices framework. RFC 3647, RFC Editor, November 2003. URL: <https://www.rfc-editor.org/rfc/rfc3647>.
- 11 World Wide Web Consortium. Decentralized identifiers (dids), 2017. Accessed: August 2022. URL: <https://w3c.github.io/did-core/>.
- 12 World Wide Web Consortium. Proposal specification of verifiable credentials, 2019. Accessed: August 2022. URL: <https://www.w3.org/TR/vc-data-model>.
- 13 Wikipedia contributors. Varsity Blues scandal, March 2019. Accessed: September 2022. URL: https://en.wikipedia.org/wiki/Varsity_Blues_scandal.
- 14 Vitalik Buterin et al. A next-generation smart contract and decentralized application platform, 2014. Accessed: August 2022. URL: <https://ethereum.org/en/whitepaper>.
- 15 Council for Higher Education Accreditation, Scientific United Nations Educational, and Cultural Organization. Toward effective practice: Discouraging degree mills in higher education, 2009. URL: <https://unesdoc.unesco.org/ark:/48223/pf0000183247>.
- 16 Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Paper 2019/953, 2019. URL: <https://eprint.iacr.org/2019/953>.
- 17 Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for zero-knowledge proof systems. Cryptology ePrint Archive, Paper 2019/458, 2019. URL: <https://eprint.iacr.org/2019/458>.
- 18 Alex Grech and Anthony F. Camilleri. Blockchain in education, 2017. EUR 28778 EN. doi:10.2760/60649.
- 19 Jens Groth and Mary Maller. Snarky signatures: Minimal signatures of knowledge from simulation-extractable snarks. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017*, pages 581–612, Cham, 2017. Springer International Publishing.
- 20 Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash protocol specification, version 2022.3.6, 2022. Accessed: September 2022.
- 21 Iden3. Circom: Circuit compiler. Accessed: September 2022. URL: <https://docs.circom.io/>.
- 22 Ben Laurie and Adam Langley. Certificate transparency, 2013. Accessed: August 2022. URL: <https://www.certificate-transparency.org/>.
- 23 Ralph C. Merkle. Protocols for public key cryptosystems. *1980 IEEE Symposium on Security and Privacy*, pages 122–122, 1980.
- 24 Alex Ozdemir and Dan Boneh. Experimenting with collaborative zk-snarks: Zero-knowledge proofs for distributed secrets. Cryptology ePrint Archive, Paper 2021/1530, 2021. URL: <https://eprint.iacr.org/2021/1530>.
- 25 Daejun Park, Yi Zhang, and Grigore Rosu. End-to-end formal verification of ethereum 2.0 deposit smart contract. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification*, pages 151–164, Cham, 2020. Springer International Publishing.
- 26 Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *Advances in Cryptology — CRYPTO '91*, pages 129–140, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.
- 27 Lum Ramabaja and Arber Avdullahu. Compact merkle multiproofs, 2020. doi:10.48550/arXiv.2002.07648.

- 28 Christian Reitwiessner. Precompiled contracts for addition and scalar multiplication on the elliptic curve alt-bn128. Accessed: September 2022. URL: <https://github.com/ethereum/EIPs/blob/master/EIPS/eip-196.md>.
- 29 Rodrigo Q. Saramago, Leander Jehl, Hein Meling, and Vero Estrada-Galiñanes. A tree-based construction for verifiable diplomas with issuer transparency. In *2021 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*, pages 101–110, 2021. doi:10.1109/DAPPS52256.2021.00017.
- 30 Avalanche Team. Avalanche. Accessed: September 2022. URL: <https://www.avax.network/>.
- 31 Iden3 Team. Identity protocol. Accessed: September 2022. URL: <https://iden3.io/>.
- 32 Polygon Team. Polygon. Accessed: September 2022. URL: <https://polygon.technology/>.
- 33 Semaphore Team. Signal anonymously. Accessed: September 2022. URL: <https://semaphore.appliedzkp.org/>.
- 34 Tornado Cash Team. Tornado cash documentation. Accessed: September 2022. URL: <https://web.archive.org/web/20220624094748/https://docs.tornado.cash/general/readme>.
- 35 Viktor Trón. The book of swarm - storage and communication infrastructure for self-sovereign digital society, 2020. URL: <https://swarm-gateways.net/bzz:/latest.bookofswarm.eth/>.
- 36 M. Turkanović, M. Hölbl, K. Košič, M. Heričko, and A. Kamišalić. Eductx: A blockchain-based higher education credit platform. *IEEE Access*, 6:5112–5127, 2018. doi:10.1109/ACCESS.2018.2789929.

A Verifiable Presentations

In this section, we present some potential use cases for our protocol. All verification flows presented in this section do not require that the subject share the entire credential document or reveal the values of the underline credential’s fields being proved.

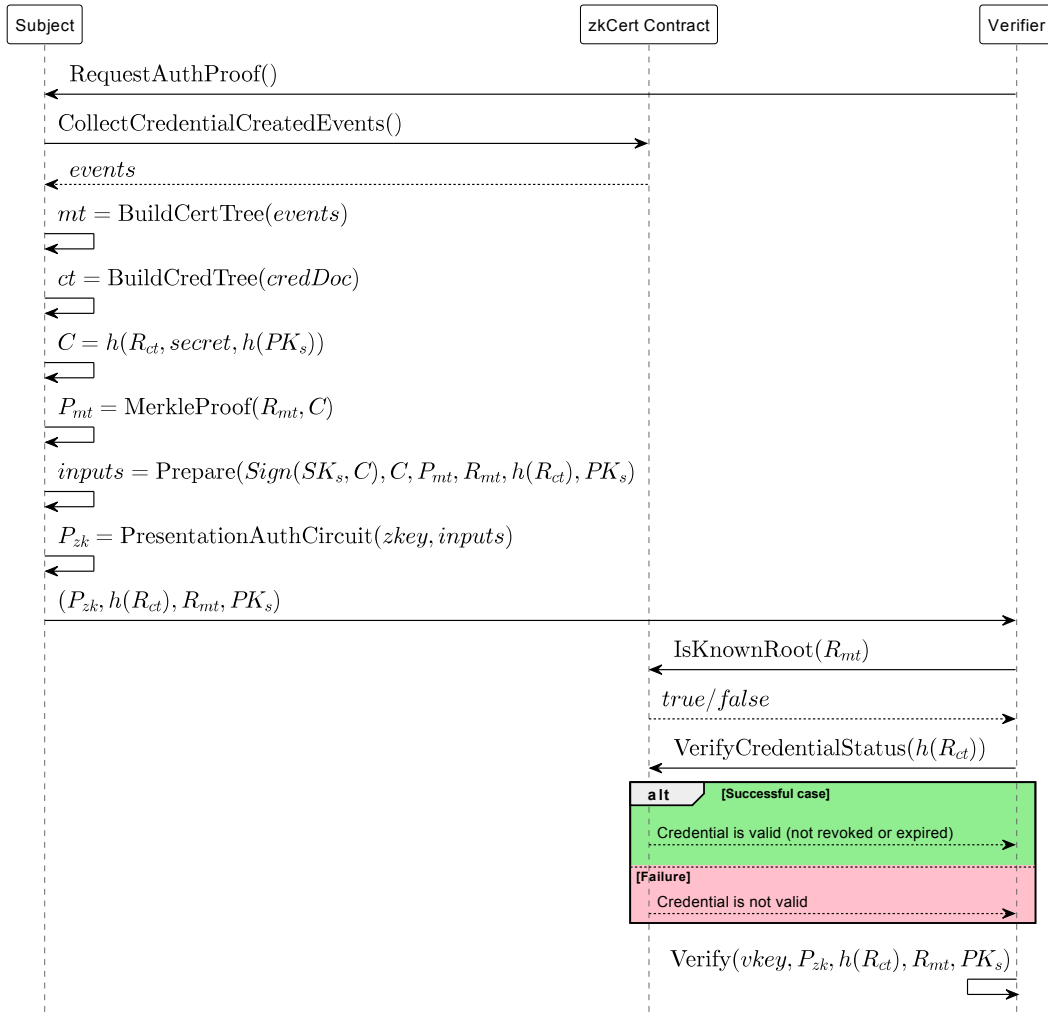
A.1 Authentication Proof

For the verification of Item (a) in Section 4.3, consider Figure 6, that shows how a subject can prove the authenticity of a credential by creating a presentation zkSNARK proof.

Like with the approval mechanism, the subject first reconstructs the latest state of the certification tree locally based on the Notary events on chain. After retrieving the necessary data, the subject generates the proof using the proving key and the private and public inputs, and send it to the verifier with the H_{cdr} , the latest certification root, and his EdDSA public key.

The verifier then checks whether the received certification root exists in the root history of the contract, and if so, verify the proof running the circuit with the received public inputs and the verification key. This verification attests that:

- The subject knows the preimage of a commitment for the presented credential;
- The commitment exists in the contract’s merkle tree for the given root;
- The given root is one of the latest roots in the contract;
- The credential was issued to the subject (i.e. he has knowledge of the EdDSA private key used to sign the commitment);
- The subject consent with the claims within the credential (i.e. the credential was approved by the subject);



■ Figure 6 Steps to verify the authenticity of a credential presentation.

A.2 Conditional Proof

For the verification of Item (b) in Section 4.3 consider the case where a student needs to prove that pass in an exam by achieving a grade higher than 60 on a scale of 0 to 100.

The student can choose the specific field of the document by its key, i.e., hash value, to be proven from his credential and create a zkSNARK proof for the field, criterion, and operator challenged by the verifier. The operator is a bitmap representing the conditional operator to be used, e.g., greater or equal. In our example, the criterion is the minimum grade required to pass the exam, i.e., 60. If the key is mismatched, the proof generation will fail. Proof generation also fails if the criterion is not satisfied.

A.3 Score Proof

For the verification of Item (c) in Section 4.3 consider the case where a student would like to apply to a study loan which requires a certain score measured based on the grades of the applicants.

Usually, such a scenario would require that each applicant share all their grades with the loan agency, e.g., by sharing their transcript of records. The loan agency would need to verify the authenticity of each application with different issuers, e.g., universities, and perform some score function to select the best candidates.

Ignoring any personal interview that may be required, the initial triage of the candidates could be automated. However, such automation may not be possible in many current systems. Further, the process may reveal more information than necessary for a triage phase, and the computation of the scores must be performed transparently to represent a fair selection of the candidates.

Thus, a naive solution to the problem may compromise the applicants' privacy. On the other hand, our solution provides an alternative to guarantee the correctness of the computation over authenticated grades while enabling automation of the verification process. Figure 7 illustrate such example in zkCert.

T_i represents a tag or ID for a course that the loan agency considered required to apply to the study program, and W_i is the correspondent weight of such a course. Let's consider that the score function is simply the weighted sum of all grades required. The loan agency would select the candidates with higher scores.

Each candidate can retrieve the information necessary to construct their proof, as described in Section 4.3, compute their score, and send the zkSNARK proof and the public inputs to the verifier.

Further, the verification of the results can be performed by anyone. For instance, a smart contract could check each applicant's zkSNARK proof and determine which of the candidates satisfies the minimum score to pass to the next phase of the admission process. It is important to note that verifying the score, sc , does not reveal the student's grades.

A.4 Time-frame Proof

For the verification of Item (d) in Section 4.3 consider the case where a bachelor's student would like to apply for a masters program. The main requirement of such program is that the student has a valid bachelor's degree, which is usually verified by requesting a diploma, i.e., the student's academic credential.

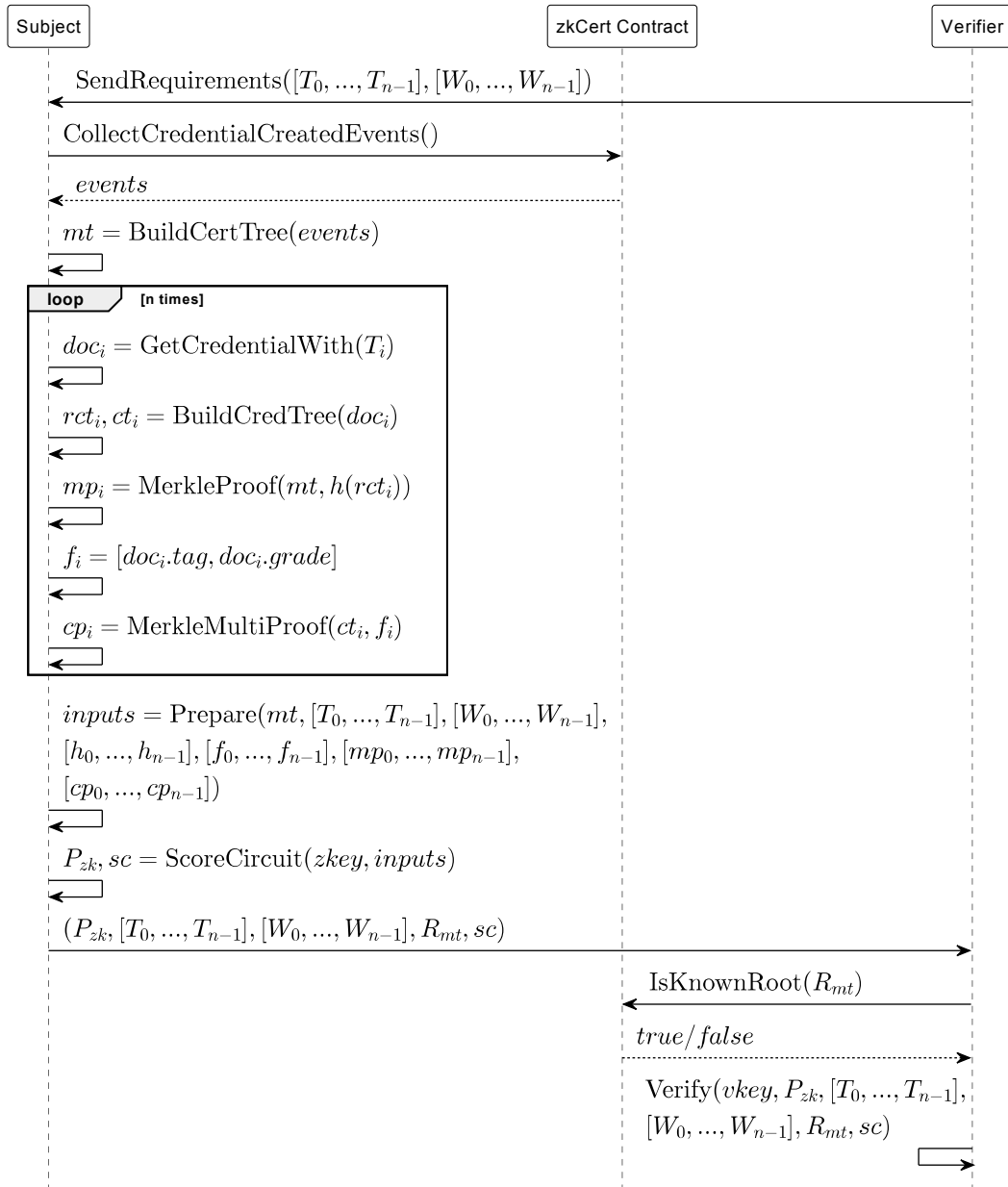
However, when verifying such credential, usually, only the authenticity of the signatures on the diploma is taken in consideration, which does not easily detect fake diplomas produced by Degree Mills [15].

zkCert can be used in such a scenario to prove that a diploma is composed of credentials that fit a specific trustful time range. As the issuer timestamps each credential on creation as well by the Notary contract on approval, neither the student nor the issuer can tamper with the period in which all the credentials that compose a diploma were issued. Figure 8 illustrate such scenario.

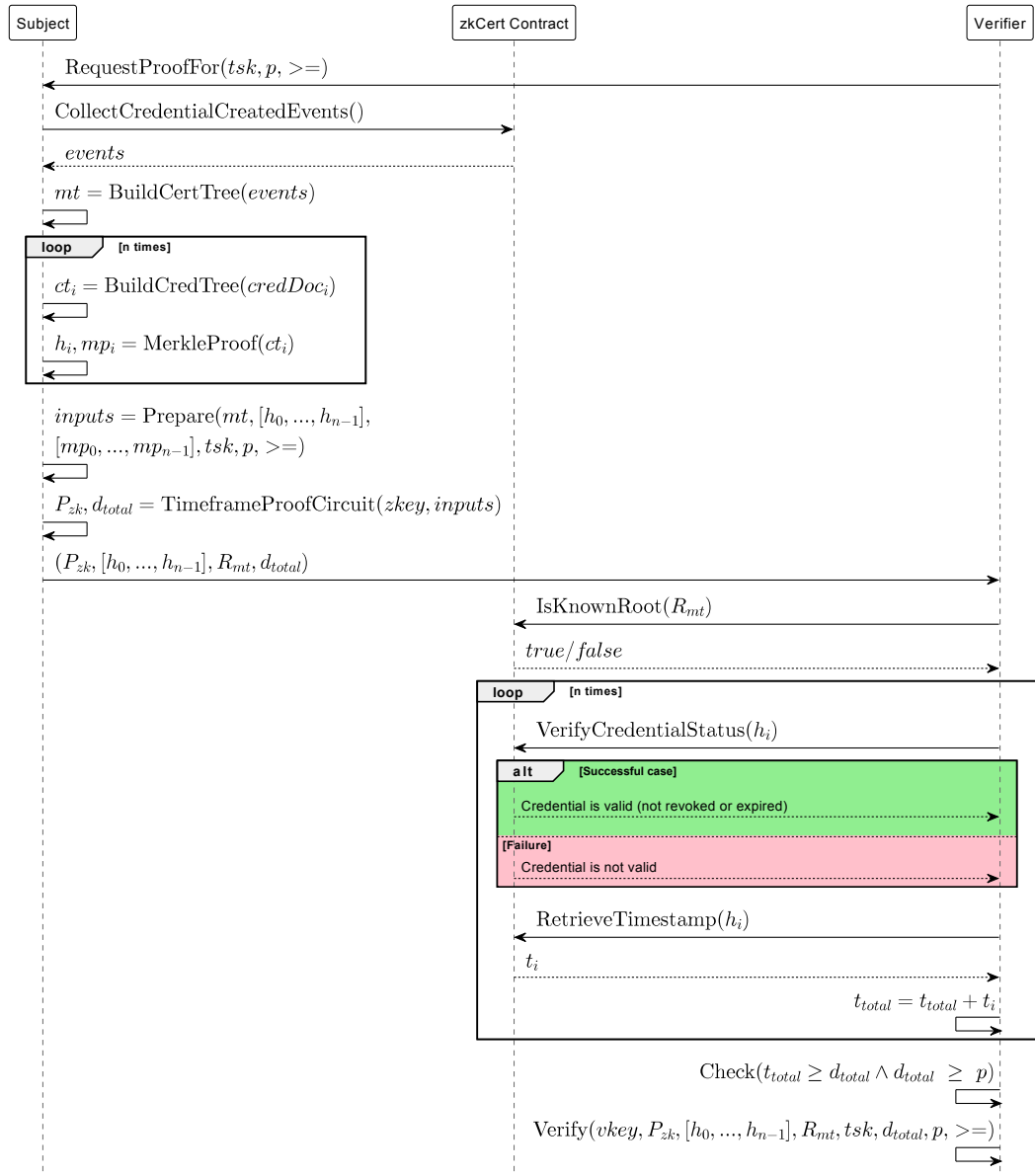
Thus, a verifier can require a time-frame proof from the student challenging him to prove that his diploma was issued over a period greater than three years. The proof can be generated and verified using the Circuit 3. The bitmap comparison operator OP is three bits bitmap that defines the currently supported operators in our implementation. For the example above, the operator would be 011, which translates to \geq . Our source code² contains more details of the currently supported operators.

The verifier can check if the issuance period was indeed greater than three years and the Circuit 3 ensures that a valid proof does not violate the incremental timestamp invariant for the provided credentials without requiring share the credentials documents with the verifier.

² <https://github.com/r0qs/zkcertree>



■ **Figure 7** Verify a score function over a set of student's credentials.



■ **Figure 8** Verify issuance period of a set of credentials.

■ **Circuit 3** Verify issuance period of a set of credentials.

```

1 TimeframeProofCircuit(n):
   ▷ n is the number of credentials in the certree
2 Public Inputs:
3   ctr                                     ▷ The latest certree root
4   NH[n]   ▷ List of n nullifier hashes sorted by their credential's timestamp
5   tsk                                           ▷ The timestamp field key
6   period                                       ▷ The duration to be checked
7   OP                                           ▷ The bitmap comparison operator
8 Private Inputs::
9   ts[n][3]                                   ▷ Timestamp field of each credential
10  MtProoffld[n]                             ▷ Field merkle proofs
11  nullifiers[n]                             ▷ The list of credential roots
12  subjects[n]                               ▷ The list of subjects
13  secrets[n]                               ▷ The list of secrets
14  MtProofcomm[n]                           ▷ The commitment's merkle proofs
15 for i ← 0 to n do
16   v ← VerifyCredentialFieldCircuit(tsk, ts[i], MtProoffld[i],
   nullifiers[i], subjects[i], secrets[i], MtProofcomm[i])
17   Verify: v == true
18 end
19 d ← ComputeTotalDuration()
20 Assert:  $\sum_{i=0}^{n-1} (ts[i+1] - ts[i] = d_i \wedge d_i > 0)$ 
21 Verify:  $d_{total} == ts[n-1] - ts[0]$ 
22 Verify: ( $d_{total} OP period$ ) == true
23 out ← dtotal

```

When Is Spring Coming? A Security Analysis of Avalanche Consensus

Ignacio Amores-Sesar ✉ 

University of Bern, Switzerland

Christian Cachin ✉ 

University of Bern, Switzerland

Enrico Tedeschi ✉ 

The Arctic University of Norway, Tromsø, Norway

Abstract

Avalanche is a blockchain consensus protocol with exceptionally low latency and high throughput. This has swiftly established the corresponding token as a top-tier cryptocurrency. Avalanche achieves such remarkable metrics by substituting proof of work with a random sampling mechanism. The protocol also differs from Bitcoin, Ethereum, and many others by forming a directed acyclic graph (DAG) instead of a chain. It does not totally order all transactions, establishes a partial order among them, and accepts transactions in the DAG that satisfy specific properties. Such parallelism is widely regarded as a technique that increases the efficiency of consensus.

Despite its success, Avalanche consensus lacks a complete abstract specification and a matching formal analysis. To address this drawback, this work provides first a detailed formulation of Avalanche through pseudocode. This includes features that are omitted from the original whitepaper or are only vaguely explained in the documentation. Second, the paper gives an analysis of the formal properties fulfilled by Avalanche in the sense of a generic broadcast protocol that only orders related transactions. Last but not least, the analysis reveals a vulnerability that affects the liveness of the protocol. A possible solution that addresses the problem is also proposed.

2012 ACM Subject Classification Theory of computation → Cryptographic protocols; Software and its engineering → Distributed systems organizing principles

Keywords and phrases Avalanche, security analysis, generic broadcast

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2022.10

Related Version *Full Version:* <https://arxiv.org/abs/2210.03423>

Funding This work has been funded by the Swiss National Science Foundation (SNSF) under grant agreement Nr. 200021_188443 (Advanced Consensus Protocols).

1 Introduction

The Avalanche blockchain with its fast and scalable consensus protocol is one of the most prominent alternatives to first-generation networks like Bitcoin and Ethereum that consume huge amounts of energy. Its AVAX token is ranked 14th according to market capitalization in August 2022 [9]. Avalanche offers a protocol with high throughput, low latency, excellent scalability, and a lightweight client. In contrast to many well-established distributed ledgers, Avalanche is not backed by proof of work. Instead, Avalanche bases its security on a deliberately metastable mechanism that operates by repeatedly sampling the network, guiding the honest parties to a common output. This allows Avalanche to reach a peak throughput of up to 20'000 transactions per second with a latency of less than half a second [29].

This novel mechanism imposes stricter security constraints on Avalanche compared to other networks. Traditional Byzantine fault-tolerant consensus tolerates up to a third of the parties to be corrupted [24] and proof-of-work protocols make similar assumptions in



© Ignacio Amores-Sesar, Christian Cachin, and Enrico Tedeschi;
licensed under Creative Commons License CC-BY 4.0

26th International Conference on Principles of Distributed Systems (OPODIS 2022).

Editors: Eshcar Hillel, Roberto Palmieri, and Etienne Rivière; Article No. 10; pp. 10:1–10:22

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

terms of mining power [13, 12]. Avalanche, however, can tolerate only up to $O(\sqrt{n})$ malicious parties. Furthermore, the transactions in the “exchange chain” of Avalanche (see below) are not totally ordered, in contrast to most other cryptocurrencies, which implement a form of atomic broadcast [6]. As the protocol is structured around a directed acyclic graph (DAG) instead of a chain, it permits some parallelism. Thus, the parties may output the same transactions in a different order, unless these transactions causally depend on each other. Only the latter must be ordered in the same way.

The consensus protocol of a blockchain is of crucial importance for its security and for the stability of the corresponding digital assets. Analyzing such protocols has become an important topic in current research. Although Bitcoin appeared first without formal arguments, its security has been widely understood and analyzed meanwhile. The importance of proving the properties of blockchain protocols has been recognized for a long time [8].

However, there are still protocols released today without the backing of formal security arguments. The Avalanche whitepaper [29] introduces a family of consensus protocols and offers rigorous security proofs for some of them. Yet the Avalanche protocol itself and the related Snowman protocol, which power the platform, are not analyzed. Besides, several key features of this protocol are either omitted or described only vaguely.

In this paper, we explain the Avalanche consensus protocol in detail. We describe it abstractly through pseudocode and highlight features that may be overlooked in the whitepaper (Sections 3–4). Furthermore, we use our insights to formally establish safety properties of Avalanche. Per contra, we also identify a weakness that affects its liveness. In particular, Avalanche suffers from a vulnerability in how it accepts transactions that allows an adversary to delay targeted transactions by several orders of magnitude (Section 5), which may render the protocol useless in practice. The problem results from dependencies that exist among the votes on different transactions issued by honest parties; the whitepaper does not address them. The attack may be mounted by a single malicious party with some insight into the network topology. Finally, we suggest a modification to the Avalanche protocol that would prevent our attacks from succeeding and reconstitute liveness of the protocol (Section 6). This version, which we call *Glacier*, restricts the sampling choices in order to break the dependencies, but also eliminates the parallelism featured by Avalanche.

The vulnerability has been acknowledged by the Avalanche developers. However, the deployed version of the protocol implements another measure that prevents the problem.

2 Related work

Despite Avalanche’s tremendous success, there is no independent research on its security. Recall that Avalanche introduces the “snow family” of consensus protocols based on sampling [29, 4]: Slush, Snowflake, and Snowball. Detailed proofs about liveness and safety for the snow-family of algorithms are given. The Avalanche protocol for asset exchange, however, lacks such a meticulous analysis. The dissertation of Yin [32] describes Avalanche as well, but does not analyze its security in more detail either.

Recall that Nakamoto introduced Bitcoin [23] without any formal analysis. This has been corrected by a long line of research, which established the conditions under which it is secure (e.g., by Garay, Kiayias, and Leonardos [13, 14] and by Eyal and Sirer [12]).

The consensus mechanisms that stand behind the best-known cryptocurrencies are meanwhile properly understood. Some of them, like the proof-of-stake protocols of Algorand [15] and the Ouroboros family that powers the Cardano blockchain [17, 10], did apply sound design principles by first introducing and analyzing the protocols and only later implementing them.

Many others, however, have still followed the heuristic approach: they released code first and were confronted with concerns about their security later. This includes Ripple [3, 1] and NEO [31], in which several vulnerabilities have been found, or Solana, which halted multiple times in 2021–2022. Stellar comes with a formal model [21], but it has also been criticized [18].

Protocols based on DAGs have potentially higher throughput than those based on chains. Notable examples include PHANTOM and GHOSTDAG [27], the Tangle of IOTA (www.iota.org), Conflux [20], and others [16]. However, they are also more complex to understand and susceptible to a wider range of attacks than those that use a chain. Relevant examples of this kind are the IOTA protocol [22], which has also failed repeatedly in practice [30] and PHANTOM [27], for which a vulnerability has been shown [19] in an early version of the protocol.

3 Model

3.1 Avalanche platform

We briefly review the architecture of the Avalanche platform [4]. It consists of three separate built-in blockchains, the *exchange* or *X-Chain*, the *platform* or *P-Chain*, and the *contract* or *C-Chain*. Additionally there are a number of subnets. In order to participate in the protocols and validate transactions, a party needs to stake at least 2'000 AVAX (about 50'000 USD in August 2022 [9]).

The *exchange chain* or *X-Chain* secures and stores *transactions* that trade digital assets, such as the native AVAX token. This chain implements a variant of the *Avalanche consensus protocol* that only partially orders the transactions and that is the focus of this work. All information given here refers to the original specification of Avalanche [29].

The *platform chain* or *P-Chain* secures *platform primitives*; it manages all other chains, allows parties to join the network, designates parties to become validators or removes them again from the validator list, and creates or deletes wallets. The P-Chain implements the *Snowman* consensus protocol: this is a special case of Avalanche consensus that always provides total order, like traditional blockchains. It is not explained in the whitepaper and we do not describe it further here.

The *C-Chain* hosts *smart contracts* and runs transactions on an Ethereum Virtual Machine (EVM). It also implements the Snowman consensus protocol of Avalanche and totally orders all transactions and blocks.

3.2 Communication and adversary

We now abstract the Avalanche consensus protocol and consider a static network of n parties $\mathcal{N} = \{p, q, \dots\}$ that communicate with each other by sending messages. An adversary may *corrupt* up to f of these parties and cause them to behave *maliciously* and diverge arbitrarily from the protocol. Non-corrupted parties are known as *honest*, messages and transactions sent by them are referred to as *honest*. Analogously, corrupted parties send *malicious* transactions and messages. The parties may access a low-level functionality for sending messages over authenticated point-to-point links between each pair of parties. In the protocol, this functionality is accessed by two events *send* and *receive*. Parties may also access a second low-level functionality for broadcasting messages through the network by gossiping, accessed by the two events *gossip* and *hear* in the protocol. Both primitives are subject to network and timing assumptions. We assume the same network model as in the original

Avalanche whitepaper [29]. Messages are delivered according to an exponential distribution, that is, the amount of time between the sending and the receiving of a message follows an exponential distribution with unknown parameter to the parties. However, messages from corrupted parties are not affected by this delay and will be delivered as fast as the adversary decides. This model differs from traditional assumptions like partial synchrony [11], because the adversary does not possess the ability to delay honest messages as it pleases.

3.3 Abstractions

The *payload transactions* of Avalanche are submitted by users and built according to the *unspent transaction output (UTXO)* model of Bitcoin [23]. A payload transaction tx contains a set of *inputs*, a set of *outputs*, and a number of digital signatures. Every input refers to a position in the output of a transaction executed earlier; this output is thereby *spent* (or *consumed*) and distributed among the outputs of tx . The balance of a user is given by the set of unspent outputs of all transactions (UTXOs) executed by the user (i.e., assigned to public keys controlled by that user). A payload transaction is valid if it is properly authenticated and none of the inputs that it consumes has been consumed yet (according to the view of the party executing the validation).

Blockchain protocols are generally formalized as *atomic broadcast*, since every party running the protocol outputs the same ordered list of transactions. However, the transaction sequences output by two different parties running Avalanche may not be exactly the same because Avalanche allows more flexibility and does not require a total order. Avalanche only orders transactions that causally depend on each other. Thus, we abstract Avalanche as a *generic broadcast* according to Pedone and Schiper [25], in which the total-order property holds only for *related* transactions as follows.

► **Definition 3.1.** *Two payloads tx and tx' are said to be related, denoted by $tx \sim tx'$, if tx consumes an output of tx' or vice versa.*

Our generic broadcast primitive is accessed through the two events $broadcast(tx)$ and $deliver(tx)$. Similar to other blockchain consensus protocols, it defines an “external” validity property and introduces a predicate V that determines whether a transaction is valid [7].

► **Definition 3.2.** *A payload tx satisfies the validity predicate of Avalanche if all the cryptographic requirements are fulfilled and there is no other delivered payload with any input in common with tx .*

For the remainder of this work, we fix the external validation predicate V to check the validity of payloads according to the logic of UTXO mentioned before.

Since Avalanche is a randomized protocol, the properties of our broadcast abstraction need to be fulfilled only with all but negligible probability.

► **Definition 3.3.** *A protocol solves validated generic broadcast with validity predicate V and relation \sim if it satisfies the following conditions, except with negligible probability:*

Validity. *If a honest party broadcasts a payload transaction tx , then it eventually delivers tx .*

Agreement. *If a honest party delivers a payload transaction tx , then all honest parties eventually deliver tx .*

Integrity. *For any payload transaction tx , every honest party delivers tx at most once, and only if tx was previously broadcast by some party.*

Partial order. *If honest parties p and q both deliver payload transactions tx and tx' such that $tx \sim tx'$, then p delivers tx before tx' if and only if q delivers tx before tx' .*

External validity. *If a honest party delivers a payload transaction tx , then $V(tx) = \text{TRUE}$.*

Note that different instantiations of the relation \sim transform the generic broadcast primitive into well-known primitives. For instance, when no pair of transactions are related, generic broadcast degenerates to reliable broadcast. Whereas when every two transactions are related, generic broadcast transforms into atomic broadcast. In our context, broadcasting corresponds to submitting a payload transaction to the network, whereas delivering corresponds to accepting a payload and appending it to the ledger.

The Avalanche protocol augments payload transactions to *protocol transactions*. A protocol transaction additionally contains a set of *references* to previously executed protocol transactions, together with further attributes regarding the execution. A protocol transaction in the implementation contains a batch of payload transactions, but this feature of Avalanche is ignored here, since it affects only efficiency. Throughout this paper, *transaction* refers to a protocol transaction, unless the opposite is indicated, and *payload* means simply a payload transaction.

A transaction references one or multiple previous transactions, unlike longest-chain protocols, in which each transaction has a unique parent [23]. An execution of the Avalanche protocol will therefore create a directed acyclic graph (DAG) that forms its ledger data structure.

Given a protocol transaction T , all transactions that it references are called the *parents* of T and denoted by $parents(T)$. The parents of T together with the parents of those, recursively, are called the *ancestors* of T , denoted by $ancestors(T)$. Analogously, the transactions that have T as parent are called the *children* of T and are denoted by $children(T)$. Finally, the children of T together with their recursive set of children are called the *descendants* of T , denoted by $descendants(T)$.

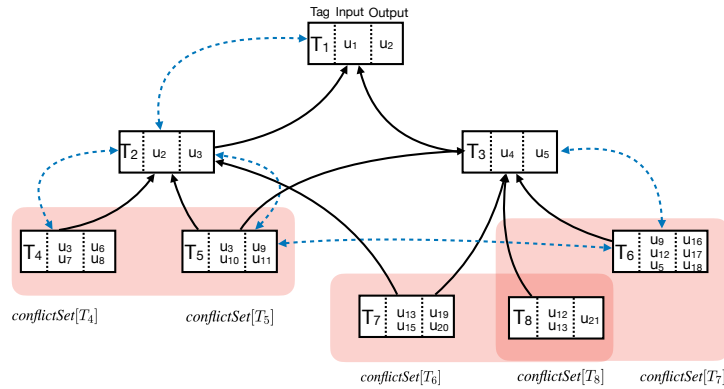
Note that two payload transactions tx_1 and tx_2 in Avalanche that consume the same input are not related, unless the condition of Definition 3.1 is fulfilled. However, two Avalanche payloads consuming the same output *conflict*. For each transaction T , Avalanche maintains a set $conflictSet[T]$ of transactions that conflict with T .

4 A description of the Avalanche protocol

Avalanche’s best-known quality is its efficiency. Permissionless consensus protocols, such as those of Bitcoin and Ethereum, are traditionally slow, suffer from low throughput and high latency, and consume large amounts of energy, due to their use of proof-of-work (PoW). Avalanche substitutes PoW with a random sampling mechanism that runs at network speed and that has every party adjust its preference to that of a (perceived) majority in the system. Avalanche also differs from more traditional blockchains by forming a DAG of transactions instead of a chain.

4.1 Overview

Avalanche is structured around its *polling* mechanism. In a nutshell, party u repeatedly selects a transaction T and sends a *query* about it to k randomly selected parties in the network. If a majority of them send a positive reply, the query is successful and the transaction contributes to the security of other transactions. Otherwise, the transaction is still processed but does not contribute to the security of any other transactions. Then the party selects a new transaction and repeats the procedure. A bounded number of such polls may execute concurrently. Throughout this work the terms “poll” and “query” are interchangeable.



■ **Figure 1** The UTXO model, conflicting transactions, and related transactions in Avalanche. The eight transactions are labeled T_1, \dots, T_8 . Each transaction is divided into three parts: the left part is a tag T_i to identify the transaction, the middle part is its set of inputs, and the right part is its set of outputs. The solid arrows indicate the references added by the protocol, showing the parents of each transaction. For instance, T_5 references T_2 and T_3 and has them as parents. The dashed double-arrows indicate related transactions. For example, T_5 and T_2 are related because u_3 is created by T_2 and consumed by T_5 . The conflict sets are denoted by the shaded (red) rectangles. As illustrated, conflict sets can be symmetric, as for T_4 and T_5 , where the conflict sets are identical ($\text{conflictSet}[T_4] = \text{conflictSet}[T_5]$) or asymmetric, as for T_6 , T_7 , and T_8 where $\text{conflictSet}[T_6] \cup \text{conflictSet}[T_7] = \text{conflictSet}[T_8]$.

In more detail, the protocol operates like this. Through the *gossip* functionality, every party is aware of the network membership \mathcal{N} . A party locally stores all those transactions processed by the network that it knows. The transactions form a DAG through their references as described in the previous section.

Whenever a user submits a payload transaction tx to the network, the user actually submits it through a party u . Then, u randomly selects a number of leaf nodes from a part of the DAG known as the *virtuous frontier*; these are the leaf nodes that are not part of any conflicting set. Party u then extends tx with references to the selected nodes and thereby creates a transaction T from the payload transaction tx . Next, u sends a QUERY message with T to k randomly, according to stake, chosen parties in the network and waits for their replies in the form of VOTE messages. When a party receives a query for T and if T and its ancestors are *preferred*, then the party replies with a positive vote. The answer to this query depends exclusively on the status of T and its *ancestors* according to the local view of the party that replies. Moreover, the definition of *preferred* is non-trivial and will be explained further below. If the polling party receives more than $\alpha > \frac{k}{2}$ positive votes, the poll is defined to be successful.

Every party u running the Avalanche protocol sorts transactions of its DAG into conflict sets.

► **Definition 4.1.** *The conflict set $\text{conflictSet}[T]$ of a given transaction T is the set of transactions that have an input in common with T (including T itself).*

Note that even if two transaction T and T' consume one common transaction output and thus conflict, their conflict sets $\text{conflictSet}[T]$ and $\text{conflictSet}[T']$ can differ, since T may consume outputs of further transactions. (In Figure 1, for example, T_8 conflicts with T_6 and T_7 , although T_7 conflicts with T_8 but not with T_6 .)

Decisions on accepting transactions are made as follows. For each of its conflict sets, a party selects one transaction and designates it as *preferred*. This designation is parametrized by a *confidence value* $d[T]$ of T , which is updated after each transaction query. If the confidence value of some conflicting transaction T^* surpasses $d[T]$, then T^* becomes the preferred transaction in the conflict set.

It has been shown [28, 29] that regardless of the initial distribution of such confidence values and preferences of transactions, this mechanism converges. For the transactions of one conflict set considered in isolation, this implies that all honest parties eventually prefer the same transaction from their local conflict sets. (The actual protocol has to respect also dependencies among the transactions; we return to this later.)

To illustrate this phenomenon, assume that there exist only two transactions T and T' and that half of the parties prefer T , whereas the other half prefers T' . This is the worst-case scenario. Randomness in sampling breaks the tie. Without loss of generality, assume that parties with preferred transaction T are queried more often. Hence, more parties consider T as preferred as a consequence. Furthermore, the next time when a party samples again, the probability of hitting a party that prefers T is higher than hitting one that prefers T' . This is the “snowball” effect that leads to ever more parties preferring T until every party prefers T .

This preferred transaction is the candidate for acceptance and incorporation into the ledger. The procedure is parametrized by a confidence counter for each conflict set, which reflects the probability that T is the preferred transaction in the local view of the party. The party increments the confidence counter whenever it receives a positive vote to a query on a descendant of T ; the counter is reset to zero whenever such a query obtains a negative vote. When this counter overcomes a given threshold, T is accepted and its payload is added to the ledger. We now present a detailed description of the protocol and refer to the pseudocode in Algorithm 1–4.

4.2 Data structures

The information presented here has been taken from the whitepaper [29], the source code [5], or the official documentation [4].

Notation. We introduce the notation used in the remaining sections including the pseudocode. For a variable a and a set \mathcal{S} , the notation $a \stackrel{R}{\leftarrow} \mathcal{S}$ denotes sampling a uniformly at random from \mathcal{S} . We frequently use hashmap data structures: A hashmap associates keys in a set \mathcal{K} with values in \mathcal{V} and is denoted by $HashMap[\mathcal{K} \rightarrow \mathcal{V}]$. For a hashmap \mathcal{F} , the notation $\mathcal{F}[K]$ returns the entry stored under key $K \in \mathcal{K}$; referencing an unassigned key gives a special value \perp .

We make use of timers throughout the protocol description. Timers are created in a stopped state. When a timer has been started, it produces a *timeout* event once after a given duration has expired and then stops. A timer can be (re)started arbitrarily many times. Stopping a timer is idempotent.

Global parameters. We recall that we model Avalanche as run by an immutable set of parties \mathcal{N} of size n . There are more three global parameters: the number k of parties queried in every poll, the majority threshold $\alpha > \frac{k}{2}$ for each poll, the acceptance parameters β_1 and β_2 , and the maximum number *maxPoll* of concurrent polls.

Local variables. Queried transactions are stored in a set \mathcal{Q} , the subset $\mathcal{R} \subset \mathcal{Q}$ is defined to be the set of *repollable* transactions, a feature that is not explained in the original paper [29]. The number of active polls is tracked in a variable *conPoll*. The parents of a transaction are

selected from the *virtuous frontier*, \mathcal{VF} , defined as the set of all *non-conflicting* transactions that have no known descendant and whose ancestors are preferred in their respective conflict sets. A transaction is non-conflicting if there is no transaction in the local DAG spending any of its inputs. For completeness, we recall that conflicting transactions are sorted in $\text{conflictSet}[T]$ formed by transactions that conflict with T , i.e., transactions which have some input in common with T .

Transactions bear several attributes related to queries and transaction preference. A *confidence value* $d[T]$ is defined to be the number of positive queries of T and its descendants. Given a conflict set $\text{conflictSet}[T]$, the variable $\text{pref}[\text{conflictSet}[T]]$, called *preferred transaction*, stores the transaction with the highest confidence value in $\text{conflictSet}[T]$. The variable $\text{last}[\text{conflictSet}[T]]$ denotes which transaction was the preferred one in $\text{conflictSet}[T]$ after the most recent update of the preferences. The preferred transaction is the candidate for acceptance in each conflict set, the acceptance is modeled by a counter $\text{cnt}[\text{conflictSet}[T]]$. Once accepted, a transaction remains the preferred one in its conflict set forever.

4.3 Detailed description

Each transaction goes through three phases during the consensus protocol: query of transactions, reply to queries, and update of preferences. All of the previous phases call the same set of functions.

Functions. The function $\text{updateDAG}(T)$ sorts the transactions in the corresponding conflict sets. The function $\text{preferred}(T)$ (L 98) outputs TRUE if T is the preferred transaction in its conflict set and FALSE otherwise. The function $\text{stronglyPreferred}(T)$ (L 100) outputs TRUE if and only if T , and everyone of its ancestors is the preferred transaction in its respective conflict set.

The function $\text{acceptable}(T)$ (L 102) determines whether T can be accepted and its payload added to the ledger or not. Transaction T is considered accepted when one of the two following conditions is fulfilled:

- T is the unique transaction in its conflict set, all the transactions referenced by \mathcal{T} are considered accepted, and $\text{cnt}[\text{conflictSet}[T]]$ is greater or equal than β_1 .
- $\text{cnt}[\text{conflictSet}[T]]$ is greater or equal than β_2 .

Finally, the function $\text{updateRepollable}()$ (L 106) updates the set of repollable transactions. A transaction T is repollable if T has already been accepted; or all its ancestors are preferred, a transaction in its conflict set has not already been accepted, and no parent has been rejected

Transaction query. A party in Avalanche progresses only by querying transactions. In each of these queries, party u selects a random transaction T (L 38), from the set of transactions that u has not previously queried by u . Then, it samples a random subset $\mathcal{S}[T] \subset \mathcal{N}$ of k parties from the set of parties running the Avalanche protocol and sends each a [QUERY, T] message. In the implementation of the protocol, party u performs up to maxPoll simultaneous queries. The repoll functionality (L 33–48) consists of performing several simultaneous transactions. When u does not know of any transaction that has not been queried, u queries a transaction that has not been accepted yet. The main idea behind this functionality is to utilize the network when this is not saturated. The repoll functionality (L 33–48) constitutes one of the most notable changes from Avalanche’s whitepaper [29].

Query reply. Whenever u receives a query message with transaction T , it replies with a message $[\text{VOTE}, u, T, \text{stronglyPreferred}(T)]$ containing the output of the binary function $\text{stronglyPreferred}(T)$ according to its local view (L 100).

Update of preferences. Party u collects the replies $[\text{VOTE}, v, T, \text{stronglyPreferred}(T)]$, and counts the number of positive votes. On the one hand, if the number of positive votes overcomes the threshold α (L 53), the query is considered successful. In this case party u loops over T and all its ancestors T' , increasing the confidence level $d[T']$ by one. If T' is the preferred transaction in its conflict set, then party u increases the counter for transaction $\text{cnt}[\text{conflictSet}[T']]$ by one. Subsequently, u checks whether T' has also previously been the preferred transaction in its conflict set. And when T' is not the preferred transaction according to the most recent query, party u will set the counter to one (L 53–67), in order to ensure that $\text{cnt}[\text{conflictSet}[T']]$ correctly reflects the number of consecutive successful queries of descendants of T' .

On the other hand, if u receives more than $k - \alpha$ negative votes, party u loops also over T and its ancestors, and sets their counters $\text{cnt}[\text{conflictSet}[T']]$ to zero as if to indicate that T' and the other transactions should not be accepted yet. (L 68–73). Party u only waits until α positive votes or $k - \alpha$ votes in total are received, since u can then determine the outcome of the query.

Acceptance of transactions. Party u accepts T when its counter $\text{cnt}[\text{conflictSet}[T]]$ reaches a certain threshold β_1 or β_2 . If T is the only transaction in its conflicting set and all its parents have already been accepted, then u accepts T if $\text{cnt}[\text{conflictSet}[T]] \geq \beta_1$, otherwise u waits until the counter overcomes a higher value β_2 .

No-op transactions. The local DAG is modified whenever a poll is finalized. In particular, only the queried transaction and its ancestors are modified. Avalanche makes use of *no-op transactions* to modify all the transactions in the DAG. After finalizing a poll, party u queries the network with all the transactions in the virtuous frontier whose state has not been modified, in a sequential manner.

4.4 Life of a transaction

We follow an honest transaction T through the protocol. The user submits the payload transaction tx to some party u , then u adds references refs to the payload transaction, creating a transaction $T = (tx, \text{refs})$. These references point to transactions in the virtuous frontier \mathcal{VF} . Transaction T is then *gossiped* through the network and added to the set of known transactions \mathcal{T} (L 22–28). Party u may also *hear* about new transactions through this gossip functionality. Whenever this is the case, u add the transaction to its set of known transactions \mathcal{T} (L 29–32).

Party u eventually selects T to be processed. When this happens, u *samples* k random parties from the network and stores them in $\mathcal{S}[T]$. Party u *queries* parties in $\mathcal{S}[T]$ with T and starts a timer $\text{timeout}[T]$. T is added to \mathcal{Q} (L 33–48).

Parties queried with T reply with the value of the function $\text{stronglyPreferred}(T)$ (L 100). This function answers positively (TRUE) if T is *strongly preferred*, i.e., if T and all of its ancestors are the preferred transaction inside each respective conflict set. A negative answer (FALSE) is returned if either T or any of its ancestors fail to satisfy these conditions.

Party u then stores the answer from party v to the query in the variable $\text{votes}[T][v]$ and proceeds according to them.

■ **Algorithm 1** Avalanche (party u), state.

Global parameters and state	
1: \mathcal{N}	// set of parties
2: $maxPoll \in \mathbb{N}$	// maximum number of concurrent polls, default value 4
3: $k \in \mathbb{N}$	// number of parties queried in each poll, default value 20
4: $\alpha \in \{\lceil \frac{k+1}{2} \rceil, \dots, k\}$	// majority threshold for queries, default value 15
5: $\beta_1 \in \mathbb{N}$	// threshold for early acceptance, default value 15
6: $\beta_2 \in \mathbb{N}$	// threshold for acceptance, default value 150
7: $\mathcal{T} \leftarrow \emptyset$	// set of known transactions
8: $\mathcal{Q} \subset \mathcal{T} \leftarrow \emptyset$	// set of queried transactions
9: $\mathcal{R} \subset \mathcal{Q} \leftarrow \emptyset$	// set of repollable transactions
10: $\mathcal{D} \subset \mathcal{T} \leftarrow \emptyset$	// set of no-op transactions to be queried
11: $\mathcal{VF} \subset \mathcal{Q} \leftarrow \emptyset$	// set of transactions in the virtuous frontier
12: $conPoll \in \mathbb{N} \leftarrow 0$	// number of concurrent polls performed
13: $conflictSet : \text{HashMap}[\mathcal{T} \rightarrow 2^{\mathcal{T}}]$	// conflict set
14: $\mathcal{S} : \text{HashMap}[\mathcal{T} \rightarrow \mathcal{N}]$	// set of sampled parties to be queried with a transaction
15: $votes : \text{HashMap}[\mathcal{T} \times \mathcal{N} \rightarrow \{\text{FALSE}, \text{TRUE}\}]$	// variable to store the replies of queries
16: $d : \text{HashMap}[\mathcal{T} \rightarrow \mathbb{N}]$	// confidence value of a transaction
17: $pref : \text{HashMap}[2^{\mathcal{T}} \rightarrow \mathcal{T}]$	// preferred transaction in the conflict set
18: $last : \text{HashMap}[2^{\mathcal{T}} \rightarrow \mathcal{T}]$	// preferred transaction in the last query
19: $cnt : \text{HashMap}[2^{\mathcal{T}} \rightarrow \mathbb{N}]$	// counter for acceptance of the conflict set
20: $accepted : \text{HashMap}[\mathcal{T} \rightarrow \{\text{FALSE}, \text{TRUE}\}]$	// indicator that a transaction is accepted
21: $timer : \text{HashMap}[\mathcal{T} \rightarrow \{\text{timers}\}]$	// timer for the query of transactions

- If u receives more than α positive votes, u runs over all the ancestors of T . If the ancestor T' was the most recent (or “last”) preferred transaction in its conflict set, its counter is increased by one. Otherwise, T' becomes the most recent preferred transaction and its counter is reset to one (L 53–67).
- If u receives at least $k - \alpha$ FALSE votes, u resets the counter for acceptance of all its ancestors $cnt[T'] \leftarrow 0$ (L 68–73).
- If timer $timeout[T]$ is triggered before the query is completed, the query is aborted instead. The votes are reset and every party is removed from the set $\mathcal{S}[T]$, so no later reply can be considered (L 80–83).

In parallel to the previous procedure, party u may perform up to $conPoll$ concurrent queries of different transactions.

Once T has been queried, it awaits in the local view of party u to be accepted. Since by assumption T is honest, $conflictSet[T] = \{T\}$. Hence T is accepted when $cnt[conflictSet[T]]$ reaches β_1 , if its ancestors are already accepted, or β_2 otherwise (L 102–104). We recall that $cnt[conflictSet[T]]$ is incremented whenever a query involving a descendant of T is successful. However, when a non-descendant of T is queried, it may trigger a no-op transaction (L 35) that is a descendant of T .

If there is no new transaction waiting to be queried, i.e., $\mathcal{T} \setminus \mathcal{Q}$ is empty, the party proceeds with a *repollable* transaction (L 40–42). A *repollable* transaction is one that has not been previously accepted but it is a candidate to be accepted (L 106–110).

Algorithm 2 Avalanche (party u), part 1.

```

22: upon broadcast( $tx$ ) do
23:   if  $V(tx)$  then
24:      $T \leftarrow (tx, \mathcal{VF})$  // up to a maximum number of parents
25:      $\mathcal{T} \leftarrow \mathcal{T} \cup \{T\}$ 
26:      $accepted[T] \leftarrow \text{FALSE}$ 
27:      $updateDAG(T)$ 
28:     gossip message [BROADCAST,  $T$ ]

29: upon hearing message [BROADCAST,  $T$ ] do
30:   if  $T \notin \mathcal{T}$  do
31:      $\mathcal{T} \leftarrow \mathcal{T} \cup \{T\}$ 
32:      $accepted[T] \leftarrow \text{FALSE}$ 

33: upon  $conPoll < maxPoll$  do
34:    $conPoll \leftarrow conPoll + 1$ 
35:   if  $\mathcal{D} \neq \emptyset$  then // prefer no-op transactions
36:      $T \leftarrow$  least recent transaction in  $\mathcal{D}$ 
37:   else if  $\mathcal{T} \setminus \mathcal{Q} \neq \emptyset$  then // take any not yet queried transaction
38:      $T \stackrel{R}{\leftarrow} \mathcal{T} \setminus \mathcal{Q}$ 
39:      $d[T] \leftarrow 0$ 
40:   else // all transaction queried already, take one of them
41:      $updateRepollable()$ 
42:      $T \stackrel{R}{\leftarrow} \mathcal{R}$ 
43:      $\mathcal{S}[T] \leftarrow sample(\mathcal{N} \setminus \{u\}, k)$  // sample  $k$  parties randomly according to stake
44:     send message [QUERY,  $T$ ] to all parties  $v \in \mathcal{S}[T]$ 
45:      $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\perp, \mathcal{VF} \setminus \{T\})\}$  // create a no-op transaction
46:     start timer[ $T$ ] // duration  $\Delta_{query}$ 
47:      $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{T\}$ 
48:      $updateDAG(T)$ 

49: upon receiving message [QUERY,  $T$ ] from party  $v$  do
50:   send message [VOTE,  $u, T, stronglyPreferred(T)$ ] to party  $v$ 

51: upon receiving message [VOTE,  $v, T, w$ ] such that  $v \in \mathcal{S}[T]$  do //  $w$  is the vote
52:    $votes[T, v] \leftarrow w$  //  $w \in \{\text{FALSE}, \text{TRUE}\}$ 

```

5 Security analysis

Avalanche deviates from the established PoW protocols and uses a different structure. Its security guarantees must be assessed differently. The bedrock of security for Avalanche is random sampling.

5.1 From Snowball to Avalanche

The Avalanche protocol family includes Slush, Snowflake, and Snowball [29] that implement single-decision Byzantine consensus. Every party *proposes* a value and every party must eventually *decide* the same value for an instance. The Avalanche protocol itself provides a “payment system” [29, Sec. V]; we model it here as generic broadcast.

10:12 When Is Spring Coming? A Security Analysis of Avalanche Consensus

■ **Algorithm 3** Avalanche (party u), part 2.

```

53: upon  $\exists T \in \mathcal{T}$  such that  $|\{v \in \mathcal{S}[T] \mid \text{votes}[T, v] = \text{TRUE}\}| \geq \alpha$  do // query successful
54:     stop  $\text{timer}[T]$ 
55:      $\text{votes}[T, *] \leftarrow \perp$  // remove all entries in  $\text{votes}$  for  $T$ 
56:      $\mathcal{S}[T] \leftarrow []$  // reset  $\mathcal{S}$  for  $T$ 
57:      $d[T] \leftarrow d[T] + 1$ 
58:     for  $T' \in \text{ancestors}(T)$  do // all ancestors of  $T$ 
59:          $d[T'] \leftarrow d[T'] + 1$ 
60:         if  $d[T'] > d[\text{pref}[\text{conflictSet}[T']]]$  then
61:              $\text{pref}[\text{conflictSet}[T']] \leftarrow T'$ 
62:         if  $T' \neq \text{last}[\text{conflictSet}[T']]$  then
63:              $\text{last}[\text{conflictSet}[T']] \leftarrow T'$ 
64:              $\text{cnt}[\text{conflictSet}[T']] \leftarrow 1$ 
65:         else
66:              $\text{cnt}[\text{conflictSet}[T']] \leftarrow \text{cnt}[\text{conflictSet}[T']] + 1$ 
67:          $\text{conPoll} \leftarrow \text{conPoll} - 1$ 

68: upon  $\exists T \in \mathcal{T}$  such that  $|\{v \in \mathcal{S}[T] \mid \text{votes}[T, v] = \text{FALSE}\}| > k - \alpha$  do // query failed
69:     stop  $\text{timer}[T]$ 
70:      $\text{votes}[T, *] \leftarrow \perp$  // remove all entries in  $\text{votes}$  for  $T$ 
71:      $\mathcal{S}[T] \leftarrow []$  // reset  $\mathcal{S}$  for  $T$ 
72:     for  $T' \in \text{ancestors}(T)$  do // all ancestors of  $T$ 
73:          $\text{cnt}[\text{conflictSet}[T']] \leftarrow 0$ 

74: upon  $\exists T \in \mathcal{T}$  such that  $\text{acceptable}(T) \wedge \neg \text{accepted}[T]$  do //  $T$  can be accepted
75:      $(tx, \text{parents}) \leftarrow T$ 
76:     if  $V(tx)$  then
77:          $\text{accepted}[T] \leftarrow \text{TRUE}$ 
78:         deliver  $tx$ 

80: upon timeout from  $\text{timer}[T]$  do // not enough votes on  $T$  received
81:      $\mathcal{Q} \leftarrow \mathcal{Q} \setminus \{T\}$ 
82:      $\text{votes}[T, *] \leftarrow \perp$  // remove all entries in  $\text{votes}$  for  $T$ 
83:      $\mathcal{S}[T] \leftarrow []$  // do not consider more votes from this query

```

The whitepaper [29] meticulously analyzes the three consensus protocols. It shows that as long as $f = O(\sqrt{n})$, the consensus protocols are live and safe [29] based on the analysis of random sampling [26]. On the other hand, an adversary controlling more than $\Theta(\sqrt{n})$ parties may have the ability to keep the network in a bivalent state. For the remainder of this section we assume $f = O(\sqrt{n})$.

However, the Avalanche protocol itself is introduced without a rigorous analysis. The most precise statement about its is that “*it is easy to see that, at worst, Avalanche will degenerate into separate instances of Snowball, and thus provide the same liveness guarantee for virtuous transactions*” [29, p. 9]. In fact, it is easy to see that this is *wrong* because every vote on a transaction in Avalanche is linked to the vote on its ancestors. The vote on a descendant T' of T depends on the state of T .

Algorithm 4 Avalanche, auxiliary functions.

```

84: function updateDAG( $T$ )
85:    $\mathcal{VF} \leftarrow$  set of non-conflicting leaves in the DAG
86:    $\text{conflictSet}[T] \leftarrow \emptyset$ 
87:   for  $T' \in \mathcal{T}$  such that  $T' \neq T$  and  $T'$  has a common input with  $T$  do
88:      $\text{conflictSet}[T] \leftarrow \text{conflictSet}[T] \cup \{T'\}$ 
89:      $\text{conflictSet}[T'] \leftarrow \text{conflictSet}[T'] \cup \{T\}$ 
90:   if  $\text{conflictSet}[T] = \emptyset$  then //  $T$  is non-conflicting
91:      $\text{pref}[\text{conflictSet}[T]] \leftarrow T$ 
92:      $\text{last}[\text{conflictSet}[T]] \leftarrow T$ 
93:      $\text{cnt}[\text{conflictSet}[T]] \leftarrow 0$ 
94:      $\text{conflictSet}[T] \leftarrow \text{conflictSet}[T] \cup \{T\}$ 

95: function getParents( $T$ )
96:    $(tx, \text{parents}) \leftarrow T$ 
97:   return  $\text{parents}$  // set of parents stored in  $T$ 

98: function preferred( $T$ )
99:   return  $T \stackrel{?}{=} \text{pref}[\text{conflictSet}[T]]$ 

100: function stronglyPreferred( $T$ )
101:   return  $\bigwedge_{T' \in \text{ancestors}(T)} \text{preferred}(T')$ 

102: function acceptable( $T$ )
103:   return  $(|\text{conflictSet}[T]| = 1 \wedge \text{cnt}[\text{conflictSet}[T]] \geq \beta_1) \wedge \bigwedge_{T' \in \text{parents}(T)} \text{acceptable}(T')$ 
104:    $\vee \text{cnt}[\text{conflictSet}[T]] \geq \beta_2$ 

105: function isRejected( $T$ )
106:   return  $\exists T' \in \mathcal{T}$  such that  $\forall T' \in \text{conflictSet}[T] \setminus \{T\} : \text{acceptable}(T')$ 

106: function updateRepollable()
107:    $\mathcal{R} \leftarrow \emptyset$ 
108:   for  $T \in \mathcal{T}$  do
109:     if  $\text{acceptable}(T) \vee \bigwedge_{T' \in \text{parents}(T)} \text{stronglyPreferred}(T') \wedge \neg \text{isRejected}(T')$  then
110:        $\mathcal{R} \leftarrow \mathcal{R} \cup \{T\}$ 

```

However, we can isolate single executions of Snowball that occur inside Avalanche. For an execution of Avalanche and a transaction T , we define an *equivalent* execution of Snowball consensus as the execution in which a party u proposes 1 if it locally prefers T in the Avalanche execution, proposes 0 if u prefers some other transaction, and does not propose otherwise. Every party also selects the same parties in each round of snowball and for a query with T , for a query with a transaction that conflicts with T , or for any query with a descendant of these two. A formal description of Snowball is provided in the full version [2].

► **Lemma 5.1.** *If party u delivers an honest transaction in Avalanche, then u decided 1 in the equivalent execution of Snowball with threshold β_1 . Furthermore, u delivers a conflicting transaction in Avalanche, then u decides 1 in Snowball with threshold β_2 .*

Proof. By construction of the Avalanche and Snowball protocols [29], the counter for acceptance of value 1 in Snowball is always greater or equal than the counter for acceptance in Avalanche. Since a successful query in Avalanche implies a successful query in Snowball,

10:14 When Is Spring Coming? A Security Analysis of Avalanche Consensus

if an honest transaction in Avalanche is delivered, the counter in the equivalent Snowball instance is at least β_1 . Analogously, if a conflicting transaction in Avalanche is delivered, then the counter in Snowball is at least β_2 . Hence, a party in Snowball would decide 1 with the respective thresholds. ◀

Looking ahead, we will introduce a modification of Avalanche that ensures the complete equivalence between Snowball and Avalanche. We first assert some safety properties of the Avalanche protocol.

► **Theorem 5.2.** *Avalanche satisfies integrity, partial order, and external validity of a generic broadcast for payload transactions under relation \sim and UTXO-validity.*

Proof. The proof is structured by property:

- **Integrity.** We show that every payload is delivered at most once. A payload tx may potentially be delivered multiple times in two ways: different protocol transactions that both carry tx may be accepted or tx is delivered multiple times as payload of the same protocol transaction.

First, we consider the possibility of accepting two different transactions T_1 and T_2 carrying tx . Assume that party u accepts transaction T_1 and party v accepts transaction T_2 . By definition, T_1 and T_2 are *conflicting* because they spend the same inputs. Using Lemma 5.1, party u and v decide differently in the equivalent execution in Snowball, which contradicts agreement property of the Snowball consensus [29].

The second option is that one protocol transaction T that contains tx is accepted multiple times. However, this is not possible either because tx is delivered only if $accepted[T] = \text{FALSE}$; variable $accepted[T]$ is set to **TRUE** when transaction T is accepted (L 74–78).

- **Partial order.** Avalanche satisfies partial order because no payload is valid unless all payloads creating its inputs have been delivered (L 74–78). Transactions T and T' are related according to Definition 3.1 if and only if T has as input (i.e., spends) at least one output of T' , or vice versa. This implies that related transactions are delivered in the same order for any party.
- **External validity.** The external validity property follows from L 74, as a payload transaction can only be delivered if it is valid, i.e., its inputs have not been previously spent and the cryptographic requirements are satisfied. ◀

Theorem 5.2 shows that Avalanche satisfies the safety properties of a generic broadcast in the presence of an adversary controlling $O(\sqrt{n})$ parties. A hypothetical adversary controlling substantially more parties could violate safety. It is not completely obvious how an adversary could achieve that. Such an adversary would broadcast two conflicting transactions T_1 and T_2 . As we already discussed, and also explained in the whitepaper of Avalanche [29], such an adversary can keep the network in a bivalent state, so the adversary keeps the network divided into two parts: parties in part \mathcal{P}_1 consider T_1 preferred, and parties in part \mathcal{P}_2 prefer T_2 . The adversary behaves as preferring T_1 when communicating with parties in \mathcal{P}_1 and as preferring T_2 when communicating with parties in \mathcal{P}_2 . Eventually, a party $u \in \mathcal{P}_1$ will query only parties in \mathcal{P}_1 or queries the adversary β_2 times in a row. Thus, u will accept transaction T_1 . Similarly, a party $v \in \mathcal{P}_2$ will eventually accept transaction T_2 . Party u will *deliver* the payload contained in T_1 and v the payload contained in T_2 , hence violating agreement. An adversary controlling at most $O(\sqrt{n})$ can also violate agreement, but the required behavior is more sophisticated, as we explain next.

5.2 Delaying transaction acceptance

An adversary aims to prevent that a party u accepts an honest transaction T . A necessary precondition for this is $\text{cnt}[\text{conflictSet}[T]] \geq \beta_1$. Note that whenever a descendant of T is queried, $\text{cnt}[\text{conflictSet}[T]]$ is modified. If the query is successful (L 53), then $\text{cnt}[\text{conflictSet}[T]]$ is incremented by one. If the query is unsuccessful, $\text{cnt}[\text{conflictSet}[T]]$ is reset to zero. Remark, however, $\text{cnt}[\text{conflictSet}[T]]$ cannot be reset to one as a result of another transaction becoming the preferred in $\text{conflictSet}[T]$ (L 62) because T is honest, as there exist no transaction conflicting with T .

Furthermore, a naive adversary that aims to delay transactions by not answering the query of a transaction T' would not succeed because the timers $\text{timeout}[T']$ would be triggered and the query would be aborted. Thus, the honest party would select new k parties to query and proceed in the protocol.

Our adversary proceeds by sending to u a series of cleverly generated transactions that reference T . We describe the steps that will delay the acceptance of T (see also Algorithm 5):

1. **Preparation phase.** The adversary submits conflicting transactions T_1 and T_2 . For simplicity, we assume that she submits first T_1 and then T_2 , so the preferred transaction in both conflict sets will be T_1 . The adversary then waits until the target transaction T is submitted.
2. **Main phase.** The adversary repeatedly sends malicious transactions referencing the target T and T_2 to u . These transactions are valid but they reference a particular set of transactions.
3. **Searching phase.** Concurrently to the main phase, the adversary looks for transactions containing the same payload as T . If some are found, she references them as well from the newly generated transactions.

■ **Algorithm 5** Liveness attack: Delaying transaction T .

Initialization

```

111:   create two conflicting transactions  $T_1$  and  $T_2$ 
112:   gossip two messages [BROADCAST,  $T_1$ ] and [BROADCAST,  $T_2$ ]
113:    $\mathcal{A} \leftarrow \emptyset$ 

114: upon hearing message [BROADCAST,  $T$ ] do                                // target transaction
115:    $\mathcal{A} \leftarrow \{T\}$ 

116: upon  $\text{cnt}[\text{conflictSet}[T]] = \lfloor \frac{\beta_1}{2} \rfloor$  in the local view of  $u$  do
117:   create  $\hat{T}$  such that  $T_2 \in \text{ancestors}(\hat{T})$  and for all  $T' \in \mathcal{A}$ , also  $T' \in \text{ancestors}(\hat{T})$ 
118:   send message [BROADCAST,  $\hat{T}$ ] to party  $u$  // pretend to gossip the message

119: upon hearing message [BROADCAST,  $\tilde{T}$ ] such  $\tilde{T}$  and  $T$  contain the same payload do
120:    $\mathcal{A} \leftarrow \mathcal{A} \cup \{\tilde{T}\}$ 

```

For simplicity, we assume that the adversary knows the acceptance counter of T at u , so she can send a malicious transaction whenever T is close to being accepted. In practice, she can guess this only with a certain probability, which will degrade the success rate of the attack. We also assume that the query of an honest transaction is always successful, which is the worst case for the adversary.

10:16 When Is Spring Coming? A Security Analysis of Avalanche Consensus

After u submits T , the adversary starts the main phase of the attack. If u queries an honest transaction \hat{T} , and if \hat{T} references a descendant of T , then $\text{cnt}[\text{conflictSet}[T]]$ increases by one. If it does not, then \hat{T} may cause u to submit a no-op transaction referencing a descendant of T . Hence, honest transactions always increase $\text{cnt}[\text{conflictSet}[T]]$ by one, this is the worst case for an adversary aiming to delay the acceptance of T .

If u queries a malicious transaction \hat{T} , then honest parties compute $\text{stronglyPreferred}(\hat{T})$ and reply with this value. Since T_2 is an ancestor of \hat{T} and not the preferred transaction in its conflict set (as we have assumed that T_1 is preferred), all queried parties return FALSE. Thus, u sets acceptance counter of every ancestor of \hat{T} to zero (L 68), in particular, $\text{cnt}[\text{conflictSet}[T]] \leftarrow 0$. However, since \hat{T} does not reference the virtuous frontier, u submits a no-op transaction that references a descendant of T , thus increasing $\text{cnt}[\text{conflictSet}[T]]$ to one.

We show that when the number of transactions is low, in particular when $|\mathcal{T} \setminus \mathcal{Q}| \leq 1$ for every party, then Avalanche may lose liveness.

► **Theorem 5.3.** *Avalanche does not satisfy validity nor agreement of generic broadcast with relation \sim with one single malicious party if $|\mathcal{T} \setminus \mathcal{Q}| \leq 1$ for every party.*

Proof. We consider again the adversary described above that targets T and u .

- **Validity.** Whenever $\text{cnt}[\text{conflictSet}[T]]$ in the local view of u reaches $\lfloor \frac{\beta_1}{2} \rfloor$, the adversary sends a malicious transaction to party u , who immediately queries it (since $|\mathcal{T} \setminus \mathcal{Q}| \leq 1$). It follows that u sets $\text{cnt}[\text{conflictSet}[T]]$ to zero and increases it intermediately afterwards, due to a no-op transaction. This process repeats indefinitely over time and prevents u from delivering the payload in T .
- **Agreement.** Assume that an honest party broadcasts the payload contained in T . The adversary forces a violation of agreement by finding honest parties u and v such that $\text{cnt}[\text{conflictSet}[T]] = \beta_1 - 1$ at v and $\text{cnt}[\text{conflictSet}[T]] < \beta_1 - 1$ at u (such parties exist because in the absence of an adversary, as $\text{cnt}[\text{conflictSet}[T]]$ increases monotonically over time). The adversary then sends an honest transaction T_h that references T to v and a malicious transaction T_m , as described before, to u . On the one hand, party v queries T_h , increments $\text{cnt}[\text{conflictSet}[T]]$ to β_1 , accepts transaction T , and delivers the payload. On the other hand, party u queries T_m and sets $\text{cnt}[\text{conflictSet}[T]]$ to one. After that, the adversary behaves as discussed before. Notice that v has delivered the payload within T but u will never do so. ◀

An adversary may thus cause Avalanche to violate validity and agreement. For this attack, however, the number of transactions in the network must be low, in particular, $|\mathcal{T} \setminus \mathcal{Q}| \leq 1$. In July 2022, the Avalanche network processed an average of 647238 transactions per day (<https://subnets.avax.network/stats/network>). Assuming two seconds per query, four times the value observed in our local implementation, the recommended values of 30 transactions per batch, and four concurrent polls, the condition $|\mathcal{T} \setminus \mathcal{Q}| \leq 1$ is satisfied 88% of the time. However, the adversary still needs to know the value of the counter for acceptance of the different parties.

5.3 A more general attack

We may relax the assumption of knowing the acceptance counters and also send the malicious transaction to more parties through gossip. After selecting a target transaction, the adversary continuously gossips malicious transactions to the network instead of sending them only to one party as in Algorithm 5. For analyzing the performance of this attack, our figure of merit will be the number of transactions to be queried by an honest party (not counting

no-ops) for confirming the target transaction T . The larger this number becomes, the longer it will take the party until it may accept T . We assume that $\mathcal{T} \setminus \mathcal{Q} \neq \emptyset$ and that a fraction γ of those transactions are malicious at any point in time¹. A non-obvious implication is that the repoll function never queries the same transaction twice.

► **Lemma 5.4.** *Avalanche requires every party to query at least β_1 transactions before accepting transaction T in the absence of an adversary.*

Proof. The absence of an adversary carries several simplifications. Firstly, there are no conflicting transactions, thus every transaction is the preferred one in its respecting conflict set and every query is successful. Secondly, due to the no-op transactions, the counter for acceptance of every transaction in the DAG is incremented by one after each query. Finally, a transaction T is accepted when its counter for acceptance reaches β_1 , since the counter of the parent of any transaction reaches β_1 strictly before T (L 102). ◀

► **Lemma 5.5.** *The average number of queried transactions before accepting transaction T in the presence of the adversary, as described in the text, is at least*

$$\beta_1 + \frac{1 + (2 + \beta_1\gamma)(1 - \gamma)^{\beta_1} - (1 - \gamma)^{2\beta_1}(1 + \beta_1\gamma)}{\gamma(1 - \gamma)^{\beta_1}(1 - (1 - \gamma)^{\beta_1})}.$$

Proof. We recall that in the worst-case scenario for the adversary, the query of an honest transaction increments the counter for acceptance of the target transaction T by one, while the query of a malicious transaction, effectively, resets the counter for acceptance to one, as a result of a no-op transaction.

Let a random variable W denote the number of transactions queried by u until T is accepted, and let $X \in \{0, 1\}$ model the outcome of the following experiment. Party u samples transactions until it picks a malicious transaction or until it has sampled $\beta_1 - 1$ honest transactions. In the first case, X takes the value zero, and otherwise, X takes the value one. By definition, X is a Bernoulli variable with parameter $p = (1 - \gamma)^{(\beta_1 - 1)}$. Thus, the number of attempts until X returns one is a random variable Y with geometric distribution, $Y \sim \mathcal{G}(p)$, with the same parameter p . We let W_a be the random variable denoting the number of queried transactions per attempt of this experiment. The expected number of failed attempts is $E[Y] = \frac{1}{(1 - \gamma)^{\beta_1}}$. Furthermore, the probability that an attempt fails after sampling exactly k transactions, for $k \leq \beta_1$, is

$$P[W_a = k | X = 0] = \frac{\gamma(1 - \gamma)^{k-1}}{1 - (1 - \gamma)^{\beta_1}}.$$

Thus, the expected number of transactions per failed attempt can be expressed as

$$E[W | X = 0] = \frac{1 - (1 - \gamma)^{\beta_1}(1 + \beta_1\gamma)}{\gamma(1 - (1 - \gamma)^{\beta_1})}. \quad (1)$$

The expected number of transaction queried during a successful attempt is at least β_1 by Lemma 5.4. Finally, the total expected number of queried transactions can be written as the expected number of transaction per failed attempt multiplied by the expected number of failed attempts plus the expected number of transactions in the successful attempt,

$$E[W] = E[W_a | X = 0] \cdot (E[Y] - 1) + E[W_a | X = 1] \cdot 1. \quad (2)$$

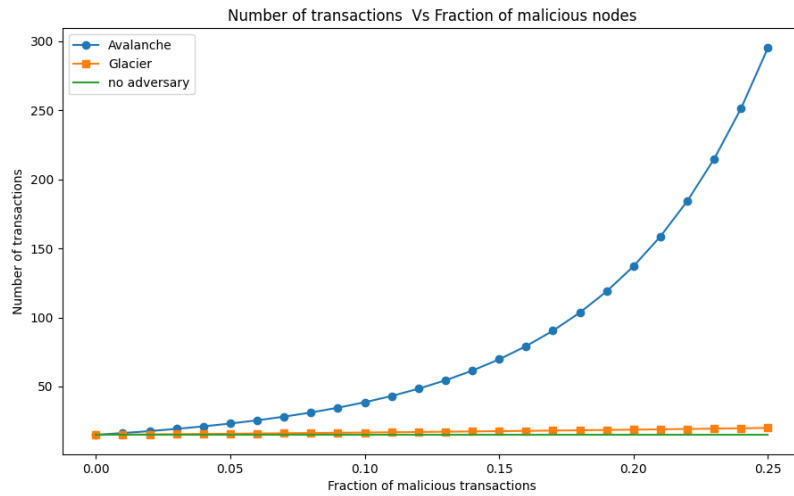
¹ Avalanche may impose a transaction fee for processing transactions. However, since the malicious transactions cannot be delivered, this mechanism does not prevent the adversary from submitting a large number of transactions.

10:18 When Is Spring Coming? A Security Analysis of Avalanche Consensus

From equations (1) and (2) and basic algebra, we obtain

$$E[W] = \beta_1 + \frac{1 + (2 + \beta_1\gamma)(1 - \gamma)^{\beta_1} - (1 - \gamma)^{2\beta_1}(1 + \beta_1\gamma)}{\gamma(1 - \gamma)^{\beta_1}(1 - (1 - \gamma)^{\beta_1})}.$$

This expression is complex to analyze. Hence, a graphical representation of this bound is given in Figure 2. It shows the expected smallest number of transactions to be queried by an honest party (not counting no-ops) until it can confirm the target transaction T . The larger this gets, the more the protocol loses liveness. It is relevant that this bound grows proportional to $\frac{1}{(1-\gamma)^{\beta_1}}$, i.e., exponential in acceptance threshold β_1 since $(1 - \gamma) < 1$.



■ **Figure 2** Expected delay in number of transactions needed to confirm a given transaction with acceptance threshold $\beta_1 = 15$, the recommended value [4], and assuming that the queries of honest transactions are successful. The (green) horizontal line shows β_1 , the expected delay without attacker. The (blue) dotted line represents the expected confirmation delay in Avalanche depending on the fraction of malicious transactions. The (orange) squared line denotes the delay in Glacier (Section 6).

The Avalanche team has acknowledged our findings and the vulnerability. The protocol deployed in the actual network, however, differs from our formalization in a way that should prevent the problem.

6 Fixing liveness with Glacier

The adversary is able to delay the acceptance of an honest transaction T because T is directly influenced by the queries of its descendants. Note the issuer of T has no control over its descendants according to the protocol. A unsuccessful query of a descendant of T carries a negative consequence for the acceptance of T , regardless of the status of T inside its conflict set. This influence is the root of the problem described earlier. An immediate, but inefficient remedy might be to run one Snowball consensus instance for each transaction. However, this would greatly degrade the throughput and increase the latency of the protocol, as many more messages would be exchanged.

We propose here a modification, called *Glacier*, in which an unsuccessful query of a transaction T carries negative consequences *only* for those of its ancestors that led to negative votes and caused the query to be unsuccessful. Our protocol is shown in Algorithm 6. It

specifically modifies the voting protocol and adds to each VOTE message for T a list L with all ancestors of T that are not preferred in their respective conflict sets (L 123–127). When party u receives a negative vote like [VOTE, v , T , FALSE, L], it performs the same actions as before. Additionally, it increments a counter for each ancestor T^* of T to denote how many parties have reported T^* as not preferred while accepting T (L 135). If u receives a positive vote, the protocol remains unchanged.

If the query is successful because u receives at least α positive votes on T , then it proceeds as before (Algorithm 3, L 53). But before u declares the query to be unsuccessful, it furthermore waits until having received a vote on T from all k parties sampled in the query (L 137). When this is the case, u only resets the counter for acceptance of those ancestors T^* of T that have been reported as non-preferred by more than $k - \alpha$ queried parties (L 141–143). If T^* is preferred by at least α parties, however, then u increments its confidence level as before (L 145).

■ **Algorithm 6** Modifications to Avalanche (Algorithm 1–4) for Glacier (party u).

State

```

121:   nonpref: HashMap[ $\mathcal{T} \times \mathcal{T} \rightarrow \mathbb{N}$ ]           // votes on  $T$  saying  $T'$  is not preferred
122: upon receiving message [QUERY,  $T$ ] from party  $v$  do           // replaces L 49
123:    $L \leftarrow []$                                            // contains the non-preferred ancestors of  $T$ 
124:   for  $T' \in \text{ancestors}(T)$  do
125:     if  $\neg \text{preferred}(T')$  then
126:       append  $T'$  to  $L$ 
127:   send message [VOTE,  $v$ ,  $T$ , stronglyPreferred( $T$ ),  $L$ ] to party  $v$ 

128: // replaces code at L 51
129: upon receiving message [VOTE,  $v$ ,  $T$ ,  $w$ ,  $L$ ] from a party  $v \in \mathcal{S}[T]$  do           //  $w$  is the vote
130:   votes[ $T$ ,  $v$ ]  $\leftarrow w$ 
131:   for  $T' \in L$  do
132:     if nonpref[ $T$ ,  $T'$ ] =  $\perp$  then
133:       nonpref[ $T$ ,  $T'$ ]  $\leftarrow 1$ 
134:     else
135:       nonpref[ $T$ ,  $T'$ ]  $\leftarrow \text{nonpref}[T, T'] + 1$ 

136: // replaces code at L 68
137: upon  $\exists T \in \mathcal{T}$  such that  $|\text{votes}[T, v]| = k \wedge \left| \{v \in \mathcal{S}[T] \mid \text{votes}[T, v] = \text{FALSE}\} \right| > k - \alpha$  do
138:   stop timer[ $T$ ]
139:   votes[ $T$ , *]  $\leftarrow \perp$                                      // remove all entries in votes for  $T$ 
140:    $\mathcal{S}[T] \leftarrow []$                                        // reset the HashMap  $\mathcal{S}$ 
141:   for  $T'$  such that nonpref[ $T$ ,  $T'$ ]  $\neq \perp$  do           // all ancestors of  $T$ 
142:     if nonpref[ $T$ ,  $T'$ ]  $> k - \alpha$  then
143:       cnt[conflictSet[ $T'$ ]]  $\leftarrow 0$ 
144:     else // nonpref[ $T$ ,  $T'$ ]  $\leq \alpha$ 
145:       cnt[conflictSet[ $T'$ ]]  $\leftarrow \text{cnt}[\text{conflictSet}[T']] + 1$ 
146:   nonpref[ $T$ , *]  $\leftarrow \perp$ 

```

Considering the adversary introduced in Section 5.3, a negative reply to the query of a descendant of the target transaction T does not carry any negative consequence for the acceptance of T here. In particular, the counter for acceptance of transaction T is never reset, even when a query is unsuccessful, because T is the only transaction in its conflicting set, then always preferred. Thus, transaction T will be accepted after β_1 successful queries, if all its parents are accepted, or β_2 successful queries if they are not accepted. Assuming that queries of honest transactions are successful, on average $\frac{\beta}{1-p}$ transactions are required

to accept T for $\beta \in [\beta_1, \beta_2]$ depending on the state of the parents of T . For simplicity we assume that the parents are accepted, thus, the counter needs to achieve the value β_1 . If this were not the case, then it is sufficient to substitute β_1 with β in the upcoming expression. Avalanche requires on average $\beta_1 + \frac{1+(2+\beta_1\gamma)(1-\gamma)^{\beta_1}-(1-\gamma)^{2\beta_1}(1+\beta_1\gamma)}{\gamma(1-\gamma)^{\beta_1}(1-(1-\gamma)^{\beta_1})}$ transactions to accept T by Lemma 5.5. The assumption that the query of honest transactions is always successful is more beneficial to Avalanche than to Glacier, since in Avalanche such a query resets the counter for acceptance of T . But in Glacier, the query simply leaves the counter as it is. The value of the acceptance threshold β_1 is also more beneficial for Avalanche since the number of required transactions increases linearly in Glacier and exponentially in Avalanche. Figure 2 shows a comparison of both expressions.

In Glacier, the vote for a transaction is independent of the vote of its descendant and ancestors, even if a query of a transaction carries an implicit query of all its ancestors. Thus, Lemma 5.1 can be extended.

► **Lemma 6.1.** *Party u delivers a transaction T with counter for acceptance with value $\text{cnt}[\text{conflictSet}[T]] \geq \beta_1$ in Glacier if and only if u decides 1 in the equivalent execution of Snowball with threshold $\text{cnt}[\text{conflictSet}[T]]$.*

Proof. Consider a transaction T in the equivalent execution of Snowball. The counter for acceptance of the value 1 in Snowball is always the same as the counter for acceptance of transaction T in Glacier because of the modifications introduced by Glacier. Thus, following the same argument as in Lemma 5.1, transaction T is accepted in Glacier with counter $\text{cnt}[\text{conflictSet}[T]]$ if and only if 1 is decided with counter $\text{cnt}[\text{conflictSet}[T]]$ in the equivalent execution of Snowball. ◀

► **Theorem 6.2.** *The Glacier algorithm satisfies the properties of generic broadcast in the presence of an adversary that controls up to $\mathcal{O}(\sqrt{n})$ parties.*

Proof. Lemma 5.1 is a special case of Lemma 6.1. Theorem 5.2 shows that Lemma 5.1 and the properties of Snowball [2] guarantee that Avalanche satisfies integrity, partial order, and external validity. In the same way, Lemma 6.1 guarantees that Glacier satisfies these same properties. Thus, it is sufficient to prove that Glacier satisfies validity and agreement.

- **Validity.** Assume that an honest party broadcasts a payload tx . Because the party is honest, the transaction T containing tx is valid and non-conflicting. In the equivalent execution of Snowball, every honest party that proposes a value proposes 1. Hence, using the validity and termination properties of Snowball, every honest party eventually decides 1. Using Lemma 6.1, every honest party eventually delivers tx .
- **Agreement.** Assume that an honest party delivers a payload transaction tx contained in transaction T . Using Lemma 6.1, an honest party decides 1 in the equivalent execution of Snowball. Because of the termination and agreement properties of Snowball, every honest party decides 1. Using Lemma 6.1 again, every honest party eventually delivers payload tx .

We conclude that Glacier satisfies the properties of generic broadcast. ◀

With the modification to Glacier, Avalanche can be safely used as the basis for a payment system. Notice that the sample mechanism is not modified, thus remains the same as in the original protocol. The only possible concern with Glacier could be a decrease in performance compared to Avalanche. However, Glacier does not reduce the performance but rather improves it. Glacier only modifies the update in the local state of party u after a query has been unsuccessful. The counter of acceptance of a given transaction T in Glacier implementation is always greater or equal than its counterpart in Avalanche. This follows

because a reset of $\text{cnt}[\text{conflictSet}[T]]$ in Glacier implies the same reset in Avalanche. Such a reset in Glacier occurs if the query of a descendant of T fails and T was reported as non-preferred by more than $k - \alpha$ parties, whereas in Avalanche it is enough if the query of the descendant failed. In Avalanche, $\text{cnt}[\text{conflictSet}[T]]$ is incremented if the query of a descendant of T succeeds, and the same occurs in Glacier. Thus, $\text{cnt}[\text{conflictSet}[T]]$ in Glacier is at least as large as in Avalanche. We recall that a transaction is accepted when $\text{cnt}[\text{conflictSet}[T]]$ reaches a threshold depending on some conditions of the local view of the DAG, but these are identical for Glacier and Avalanche. Hence, every transaction that is accepted in Avalanche is accepted in Glacier with equal or smaller latency. This implies not only that the latency of Glacier is smaller than the latency of Avalanche, but also that the throughput of Glacier is at least as good as the throughput of Avalanche.

7 Conclusion

Avalanche is well-known for its remarkable throughput and latency that are achieved through a metastable sampling technique. Our pseudocode captures in a compact and relatively simple manner the intricacies of the protocol. We show that Avalanche, as originally introduced, possesses a vulnerability allowing an adversary to delay transactions arbitrarily. We also address such vulnerability with a modification of the protocol, Glacier, that allows Avalanche to satisfy both safety and liveness.

The developers of Avalanche have acknowledged the vulnerability, and the actual implementation does not suffer from it due to an alternative fix. Understanding this variant of Avalanche remains open and is subject of future work.

References

- 1 Ignacio Amores-Sesar, Christian Cachin, and Jovana Micic. Security analysis of ripple consensus. In *OPODIS*, volume 184 of *LIPICs*, pages 10:1–10:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.
- 2 Ignacio Amores-Sesar, Christian Cachin, and Enrico Tedeschi. When is spring coming? A security analysis of avalanche consensus. *CoRR*, abs/2210.03423, 2022. [arXiv:2210.03423](https://arxiv.org/abs/2210.03423).
- 3 Frederik Armknecht, Ghassan O. Karame, Avikarsha Mandal, Franck Youssef, and Erik Zenner. Ripple: Overview and outlook. In *TRUST*, volume 9229 of *Lecture Notes in Computer Science*, pages 163–180. Springer, 2015.
- 4 Ava Labs, Inc. Avalanche documentation. <https://docs.avax.network/>.
- 5 Ava Labs, Inc. Node implementation for the Avalanche network. <https://github.com/ava-labs/avalanchego>.
- 6 Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.
- 7 Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *CRYPTO*, volume 2139 of *Lecture Notes in Computer Science*, pages 524–541. Springer, 2001.
- 8 Christian Cachin and Marko Vukolic. Blockchain consensus protocols in the wild (keynote talk). In *DISC*, volume 91 of *LIPICs*, pages 1:1–1:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.
- 9 Coinmarketcap: Today’s cryptocurrency prices by market cap. <https://coinmarketcap.com/>, 2022.
- 10 Bernardo David, Peter Gazi, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In *EUROCRYPT (2)*, volume 10821 of *Lecture Notes in Computer Science*, pages 66–98. Springer, 2018.

- 11 Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.
- 12 Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *Financial Cryptography*, volume 8437 of *Lecture Notes in Computer Science*, pages 436–454. Springer, 2014.
- 13 Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *EUROCRYPT (2)*, volume 9057 of *Lecture Notes in Computer Science*, pages 281–310. Springer, 2015.
- 14 Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol with chains of variable difficulty. In *CRYPTO (1)*, volume 10401 of *Lecture Notes in Computer Science*, pages 291–323. Springer, 2017.
- 15 Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *SOSP*, pages 51–68. ACM, 2017.
- 16 Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is DAG. In *PODC*, pages 165–175. ACM, 2021.
- 17 Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *CRYPTO (1)*, volume 10401 of *Lecture Notes in Computer Science*, pages 357–388. Springer, 2017.
- 18 Minjeong Kim, Yujin Kwon, and Yongdae Kim. Is stellar as secure as you think? In *EuroS&P Workshops*, pages 377–385. IEEE, 2019.
- 19 Chenxing Li, Peilun Li, Wei Xu, Fan Long, and Andrew Chi-Chih Yao. Scaling nakamoto consensus to thousands of transactions per second. *CoRR*, abs/1805.03870, 2018. [arXiv:1805.03870](https://arxiv.org/abs/1805.03870).
- 20 Chenxing Li, Peilun Li, Dong Zhou, Zhe Yang, Ming Wu, Guang Yang, Wei Xu, Fan Long, and Andrew Chi-Chih Yao. A decentralized blockchain with high throughput and fast confirmation. In *USENIX Annual Technical Conference*, pages 515–528. USENIX Association, 2020.
- 21 Marta Lohkava, Giuliano Losa, David Mazières, Graydon Hoare, Nicolas Barry, Eli Gafni, Jonathan Jove, Rafal Malinowsky, and Jed McCaleb. Fast and secure global payments with stellar. In *SOSP*, pages 80–96. ACM, 2019.
- 22 Hamed Mamache, Gabin Mazué, Osama Rashid, Gewu Bu, and Maria Potop-Butucaru. Resilience of IOTA consensus. *CoRR*, abs/2111.07805, 2021. [arXiv:2111.07805](https://arxiv.org/abs/2111.07805).
- 23 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Whitepaper, <https://bitcoin.org/bitcoin.pdf>, 2009.
- 24 Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.
- 25 Fernando Pedone and André Schiper. Generic broadcast. In *DISC*, volume 1693 of *Lecture Notes in Computer Science*, pages 94–108. Springer, 1999.
- 26 Sheldon Ross. *Stochastic Processes*. Wiley, second edition, 1996.
- 27 Yonatan Sompolinsky, Shai Wyborski, and Aviv Zohar. PHANTOM GHOSTDAG: a scalable generalization of nakamoto consensus: September 2, 2021. In *AFT*, pages 57–70. ACM, 2021.
- 28 W. Y. Tan. On the absorption probabilities and absorption times of finite homogeneous birth-death processes. *Biometrics*, 32(4):745–752, 1976.
- 29 Team Rocket, Maofan Yin, Kevin Sekniqi, Robbert van Renesse, and Emin Gün Sirer. Scalable and probabilistic leaderless BFT consensus through metastability. e-print, [arXiv:1906.08936](https://arxiv.org/abs/1906.08936) [cs.CR], 2019.
- 30 Bozhi Wang, Qin Wang, Shiping Chen, and Yang Xiang. Security analysis on tangle-based blockchain through simulation. *CoRR*, abs/2008.04863, 2020. [arXiv:2008.04863](https://arxiv.org/abs/2008.04863).
- 31 Qin Wang, Jiangshan Yu, Zhiniang Peng, Van Cuong Bui, Shiping Chen, Yong Ding, and Yang Xiang. Security analysis on dbft protocol of NEO. In *Financial Cryptography*, volume 12059 of *Lecture Notes in Computer Science*, pages 20–31. Springer, 2020.
- 32 Maofan Yin. *Scaling the Infrastructure of Practical Blockchain Systems*. PhD thesis, Cornell University, USA, 2021.

Computational Power of a Single Oblivious Mobile Agent in Two-Edge-Connected Graphs

Taichi Inoue ✉

Osaka University, Japan

Naoki Kitamura ✉

Osaka University, Japan

Taisuke Izumi ✉

Osaka University, Japan

Toshimitsu Masuzawa ✉

Osaka University, Japan

Abstract

We investigated the computational power of a single mobile agent in an n -node graph with storage (i.e., node memory). Generally, a system with one-bit agent memory and $O(1)$ -bit storage is as powerful as that with $O(n)$ -bit agent memory and $O(1)$ -bit storage. Thus, we focus on the difference between one-bit memory and oblivious (i.e., zero-bit memory) agents. Although their computational powers are not equivalent, all the known results exhibiting such a difference rely on the fact that oblivious agents cannot transfer any information from one side to the other across the bridge edge. Hence, our main question is as follows: Are the computational powers of one-bit memory and oblivious agents equivalent in 2-edge-connected graphs or not? The main contribution of this study is to answer this question under the relaxed assumption that each node has $O(\log \Delta)$ -bit storage (where Δ is the maximum degree of the graph). We present an algorithm for simulating any algorithm for a single one-bit memory agent using an oblivious agent with $O(n^2)$ -time overhead per round. Our results imply that the topological structure of graphs differentiating the computational powers of oblivious and non-oblivious agents is completely characterized by the existence of bridge edges.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases mobile agent, depth-first search, space complexity

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2022.11

Funding This work was supported by JSPS KAKENHI Grant Number JP22K21277, JP19H04085, JP20KK0232, JP20H04140, and JP21H05854.

Acknowledgements This study was initiated by the discussion with Prof. Shantanu Das when he visited the third author (Taisuke Izumi) in 2018. We greatly appreciate his valuable comments at the discussion.

1 Introduction

1.1 Background and Our Result

A *mobile agent* (hereinafter called an *agent*) is an individual entity that performs a given task by autonomously moving in a graph (or network). This is one of the main computational paradigms of distributed algorithms. In the theory of mobile agent systems, there exist various models that differ in memory resources, asynchrony, observation capability of the system, and so on. Revealing the computational power of each model is recognized as a central question in this research field. This study focuses on the computational power of a system with a single agent, despite that it is fairly commonplace to consider the cooperation of multiple agents.



© Taichi Inoue, Naoki Kitamura, Taisuke Izumi, and Toshimitsu Masuzawa;
licensed under Creative Commons License CC-BY 4.0

26th International Conference on Principles of Distributed Systems (OPODIS 2022).

Editors: Eshcar Hillel, Roberto Palmieri, and Etienne Rivière; Article No. 11; pp. 11:1–11:18

Leibniz International Proceedings in Informatics



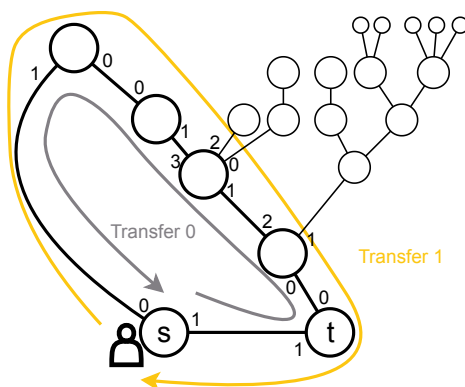
LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In single-agent systems, the amount of (persistent) memory held by the agent and nodes is a major quantitative parameter of their computational capability. The computational cost is measured by *rounds* (i.e., the number of movements by the agent). Throughout this paper, we refer to the memory of the agent as *memory* and that of each node as *storage*. It is almost evident that an agent with sufficiently large memory can simulate any centralized algorithm. That is, any computable problem can be solved. Hence, theoretical interest lies in problem solvability when the amount of memory is limited. For example, the graph exploration problem, which requires the agent to visit all nodes in the graph, is a fundamental problem. Generally, $\Theta(\log n)$ -bit memory is necessary and sufficient to solve this problem when the nodes have no storage [5, 10]. It is also proved that the system with $O(1)$ -bit memory and $O(1)$ -bit storage can solve the graph exploration problem [1].

In this study, we investigated the effect of the amount of memory on problem solvability. A known result on this research line is that, in any n -node graph with $O(1)$ -bit storage per node, the agent with one-bit memory is as powerful as that with $O(n)$ -bit memory [6]. More precisely, any property of the graph G decidable by the $O(n)$ -bit memory agent within polynomial movements and polynomial-time local computation is also decidable by the one-bit memory agent within polynomial movements and polynomial-time local computation. This result is not limited to decision problems, but actually provides a general technique for simulating the execution of any $O(n)$ -bit memory agent using a one-bit memory agent with a polynomial-time multiplicative overhead per round. It has also been shown that the computational powers of the one-bit and zero-bit (i.e., oblivious) memory agents are inherently different regardless of the amount of available storage. Thus, our question is already closed in general settings. However, whether the separation between the one-bit memory and oblivious agents is exhibited in a restricted graph class remains unclear. All known results exhibiting such a separation [1, 6] rely on the existence of a bridge edge (i.e., a single edge such that its removal disconnects the graph) in the graph. More precisely, they are derived from the fact that the oblivious agent cannot transfer any information from one side to the other side across the bridge edge, whereas the one-bit memory agent can. Therefore, our central question is then rephrased as: Are the computational powers of the one-bit memory and oblivious agents equivalent in 2-edge-connected graphs or not? The main contribution of this study is to answer this question positively. We focus on 2-edge-connected graphs with a maximum degree Δ and relax the constraint on the amount of storage from $O(1)$ bits to $O(\log \Delta)$ bits. In this setting, we present a polynomial-time overhead algorithm that simulates the execution of a single one-bit memory agent by an oblivious agent. By combining the results of [6], we can deduce the equivalence between an $O(n)$ -bit memory agent and an oblivious agent in 2-edge-connected graphs. This implies that the topological structure of graphs differentiating the computational powers of oblivious and non-oblivious agents is completely characterized by the existence of bridge edges. The authors believe that this result provides a sharp insight into the computational complexity theory of mobile agents.

1.2 Technical Idea

The model considered in this study follows the standard assumptions on mobile agent systems. The graph G is *anonymous* (i.e., the agent cannot refer to the unique IDs of nodes), and the neighbors of a node are identified by the *local port numbers* assigned to the edges incident to the node. When an agent enters a node, it recognizes the *entry port number* (assigned to the edge used to visit the current node). In one round, the agent performs local computation, which includes updating the information stored in its own memory and the storage at the current node, and decides the *outgoing port number* (assigned to the edge used to leave the node).



■ **Figure 1** Example of the construction of an ITC (drawn by bold lines) on a DFS tree and imitative transmission of one-bit data. Arrows are moving directions of the agent. Some local port numbers are omitted.

The main technical issue in simulating a one-bit memory agent is how we can transfer the information to an oblivious agent. We resolve this by utilizing information on entry port numbers. To clarify our idea, we first consider the simple case in which the graph is an oriented cycle and assume that the agent at a node s wants to transfer one-bit information to the neighbor t of s . There are two distinct paths from s to t (i.e., clockwise and counterclockwise, specified by the orientation), which result in different entry port numbers for t . The agent can transfer an information bit $b \in \{0, 1\}$ from s to t using the path ending up with port number b .

When the graph topology is arbitrary, the simulation algorithm becomes more complicated; however, the fundamental idea is the same. The algorithm first finds a cycle C of G containing edge (s, t) , which we refer to as the *information transfer cycle (ITC)*, to transfer one-bit information from node s to its neighbor t . The algorithm provides a consistent orientation to the ITC and applies the algorithm above for oriented cycles to the ITC. Because we assume that G is 2-edge-connected, there necessarily exists an ITC for any edge (s, t) in G .

Our algorithm runs a depth-first search (DFS) to compute ITC. The agent starts DFS from s with the choice of (s, t) as the first traversal edge. Because G is 2-edge-connected, the agent eventually reaches s again through edge e different from (s, t) . Figure 1 shows an example of transmission on an ITC. Let $P_{s,t}$ be the path from s to t managed by the DFS algorithm. Then, the algorithm obtains the ITC $P_{s,t} + e$. The agent must reset all information because the garbage information is left at the nodes that are traversed by the DFS process but not contained in the constructed ITC. It is implemented by careful re-execution of the DFS starting from s with the first traversal edge (s, t) .

We emphasize that its implementation is far from triviality, although the intuition of our algorithm stated above is simple and easy to follow. A technical hurdle is that the agent itself is oblivious. The agent cannot memorize the subtask currently executed despite our algorithmic idea being constituted of several subtasks. Our algorithm conducts the sequential phase-by-phase composition of all subtasks carefully, using only node storage. Details of the design are presented in Section 3.

1.3 Related Work

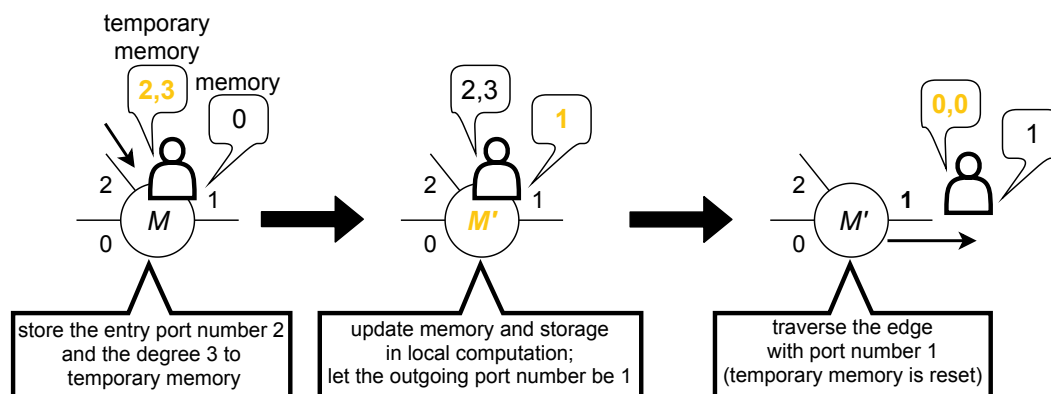
To the best of our knowledge, the authors' prior work [6] is the first to consider the effect of memory size on the solvability of general tasks in graphs with storage, but several studies have been conducted on specific problems. One benchmark problem is the graph exploration

problem. In the case where nodes have no storage, it has been shown that $\Theta(\log n)$ -bit memory is necessary and sufficient to solve it within polynomial time [5, 10]. When allowing nodes to be stored, the graph exploration algorithm can be solved by an oblivious agent using $O(\log \Delta)$ -bit storage per node (where Δ is the maximum degree of the graph) [11]. Minimizing the storage size, there exists a graph exploration algorithm for an $O(1)$ -bit memory agent using $O(1)$ -bit storage [1]. It also shows that exploration using a single oblivious agent is impossible if the storage size is $O(1)$ bits. Note that the topology used in their impossibility proof has a bridge edge; that is, it is not 2-edge-connected. An interesting open problem inspired by our result is whether an oblivious agent with $O(1)$ -bit storage can solve the graph exploration problem for 2-edge-connected graphs or not. It is possible to explore 2-edge-connected graphs if one can construct the ITC using only $O(1)$ -bit storage because our simulation algorithm requires $\omega(1)$ -bit storage only during the construction of the ITC. There also exists a model in which the agent uses a “pebble” device that can be put on and picked up from nodes by the agent. Disser et al. [4] showed that n -node graph exploration can be solved using $\Theta(\log \log n)$ pebbles. However, this method has the disadvantage of being time inefficient since it takes $n^{O(\log \log n)}$ rounds to complete the exploration. Although it is a crucial assumption in our simulation algorithm that the agent knows the entry port number, there is a weaker model called the *myopic robot*, in which the entry port number is undetectable by agents [2, 7, 8, 9]. Although no formal proof is given, it is almost trivial that such a weaker agent cannot simulate the agent memory as in our result, even under the assumption of 2-edge-connectivity. While the model and problem are significantly different, Dieudonne et al. [3] presented a solution based on an approach similar to ours. In that study, the authors considered the information transfer between the robots in the 2D plane, which cannot have an explicit communication channel. In their approach, the transferred information is not embedded into movement patterns as in our approach but is embedded into the (real-value) distance between two robots.

2 Preliminaries

2.1 Graph

We used $[a, b]$ to denote the set of integers at least a and at most b . Let G be a graph with n nodes. This is formally defined as a port-numbered graph $G = (V, E, \Pi)$, where V is the set of nodes, E is the set of edges, and Π is the set of port-numbering functions (explained later). Throughout this paper, we assume that G is simple, undirected, and 2-edge-connected. Each node in G is identified by a non-negative integer i ($0 \leq i \leq n - 1$). However, the agent operating in G cannot see those values, that is, G is anonymous (the behavior of the agent is formally defined in Section 2.2). For $e \in E$, we define $G - e$ as a graph obtained by removing e from G . Note that $G - e$ is always connected because we assume that G is 2-edge-connected. The edges incident on each node i are distinguished by *local port numbers*. The *port numbering function* at node i is defined as $\pi_i : [0, \Delta_i - 1] \rightarrow N_i$, where Δ_i is the degree of node i and N_i is the set of neighbors of i . The maximum degree of G is defined as Δ . Note that Δ_i is necessarily greater than one because G is 2-edge-connected. We define π_i^{-1} as the inverse function of π_i . The set Π consists of port-numbering functions for all the nodes in V . Each node in G has storage (i.e., node memory) that keeps the stored information even after the agent leaves the node.



■ **Figure 2** Workflow of one-round operation (where M and M' mean the storage value of the node).

2.2 Mobile Agent

We consider a single-agent model in which one mobile agent moves in graph G . A mobile agent has two distinct memory spaces: *temporary memory* and *persistent memory*. Temporary memory is a working space used for computation at each node, but the stored information is completely reset when the agent leaves the node. In contrast, persistent memory space retains the stored information even when the agent moves from one node to another. In the following argument, we use the terminology “(agent) memory” to refer to persistent memory and measure the space complexity of the agent by the amount of persistent memory. Although the size of the available temporary memory is unbounded, $O(\log \Delta)$ bits of temporary memory are sufficient to implement our algorithm. In this study, we focus on the *oblivious agent* (or the agent with zero-bit memory) and *one-bit agent* (or the agent with one-bit memory).

The execution of the agent follows discrete rounds $t = 0, 1, \dots$. In each round, the agent performed the following two computational steps:

1. Let i be the node where the agent currently stays. First, the agent starts the local computation specified by the algorithm. At the beginning of the local computation, the local port number corresponding to the edge through which it has entered the current node i (i.e., *entry port number*) and the degree Δ_i of i are stored in the temporary memory. Note that the entry port number can be arbitrary at the beginning of the execution. Following the information stored in the temporary memory, persistent memory, and storage of i , the agent updates the storage of i and its own persistent memory. It also determines the port number corresponding to the edge through which it leaves the node i (i.e., *outgoing port number*).
2. The agent moves to the neighbor of i specified by the outgoing port number. Note that the agent is guaranteed to arrive at the neighbor within the current round.

Figure 2 illustrated the behavior of a single round. An algorithm A for a one-bit agent is defined as $A = (m, M, M', l, \phi)$, a 5-tuple of an initial memory value m , an initial storage value M for each node, an initial storage value M' for the initial location of the agent, an initial location l of the agent, and a transition function ϕ . Note that the special initialization value M' is crucial for implementing the simulation because a symmetry-breaking mechanism is required to distinguish the initial location of the simulated agent from other nodes. The *transition function* ϕ for a one-bit agent is defined as $\phi : \mathbb{N} \times \mathbb{Z}_{\geq -1} \times \{0, 1\}^* \times \{0, 1\} \rightarrow \mathbb{Z}_{\geq -1} \times \{0, 1\}^* \times \{0, 1\}$, where $\mathbb{Z}_{\geq -1}$ is a set of integers greater than or equal to -1. The

arguments given to ϕ are the degree of the node, entry port number, storage value of the current node, and memory value. Note that the bit length of the storage is not fixed by the algorithm because our model allows the storage size to depend on the parameters of graph G . The returned values are the outgoing port number, storage value to be left at the current node, and memory value after local computation. Outgoing port number -1 implies that the agent terminates the execution of the algorithm and stays at the same node. For convenience, we consider an algorithm for an oblivious agent such that the memory value given to and returned by ϕ is always 0. Note that ϕ does not consider the identifier of the current node as an argument because we assume anonymous graphs.

Next, we define the system configuration and execution.

► **Definition 1 (configuration).** A configuration C of the system is represented as $C = ((M_0, M_1, \dots, M_{n-1}), m, p, l) \in \{\{0, 1\}^*\}^n \times \{0, 1\} \times \mathbb{Z}_{\geq -1} \times [0, n - 1]$, where M_i is the storage value of node i , m is the memory value of the agent, p is the entry port number, and l is the location of the agent. A valid configuration in G is such that the value of the entry port number is in the range $[-1, \Delta_l - 1]$.

► **Definition 2 (execution).** Let $A = (m, M, M', l, \phi)$ be an algorithm. Given two valid configurations, $C_t = ((M_{0,t}, \dots, M_{n-1,t}), m_t, p_t, l_t)$ and $C_{t+1} = ((M_{0,t+1}, \dots, M_{n-1,t+1}), m_{t+1}, p_{t+1}, l_{t+1})$, we consider that they follow the transition function ϕ of A in G , denoted by $C_t \xrightarrow{\phi, G} C_{t+1}$, if the following three conditions are satisfied:

- $\phi(\Delta_{l_t}, p_t, M_{l_t,t}, m_t) = (\pi_{l_t}^{-1}(l_{t+1}), M_{l_t,t+1}, m_{t+1})$
- $M_{i,t+1} = M_{i,t}$ for all $i \neq l_t$
- $p_{t+1} = \pi_{l_{t+1}}^{-1}(l_t)$

Let C be any valid configuration on G such that the initial value of the memory is m , the location of the agent is l , the initial storage value for node $i \neq l$ is M and M' for node l . The execution $E(G, C, A)$ of the system is defined as an infinite sequence of configurations $(C_t)_{t \geq 0}$ such that $C_0 = C$ and $C_t \xrightarrow{\phi, G} C_{t+1}$ holds for any $t \geq 0$.

In Definition 2, any execution is defined as an infinite sequence; however, if the algorithm terminates in a finite round, then the execution is defined as an infinite sequence such that $C_t = C_{t+1} = \dots$ holds for the termination round t . As the transition function ϕ is assumed to be deterministic, the execution of the algorithm is uniquely defined from the initial configuration $C_0 = C$. That is, $E(G, C, A)$ is uniquely defined. In addition, throughout this study, we only consider algorithms that do not depend on the initial entry port number. Then, execution $E(G, C, A)$ is determined only by algorithm A , graph G , and initial location l of the agent. Hence, we refer to the execution $E(G, C, A)$ as $E(G, v, A)$ with the initial location $v \in V$ of the agent.

2.3 Simulation

Let $\mathcal{A}(k, \lambda)$ be the set of all algorithms that utilize k -bit memory and λ -bit storage. Note that k and λ are not necessarily constant values, but might be functions that depend on the parameters of the graph. The goal of this study was to design an algorithm $A \in \mathcal{A}(0, \lambda)$ that simulates any algorithm $A^* \in \mathcal{A}(1, \lambda^*)$, where λ and λ^* represent the storage sizes of the simulator and simulated algorithms, respectively. To this end, we provide a formal definition for the simulation of the algorithm. First, we present the notion of γ -configurations.

► **Definition 3 (γ -configuration).** Let $\gamma = (\gamma_M, \gamma_A)$ be a pair of mappings, $\gamma_M : \{0, 1\}^\lambda \rightarrow \{0, 1\}^{\lambda^*}$ and $\gamma_A : \{0, 1\}^\lambda \rightarrow \{0, 1\} \times \mathbb{N}$. The γ -configuration $\gamma(C)$ is defined as $\gamma(C) = ((\gamma_M(M_0), \gamma_M(M_1), \dots, \gamma_M(M_{n-1})), \gamma_A(M_l), l)$ for configuration $C = ((M_0, M_1, \dots, M_{n-1}), m, p, l)$.

Intuitively, the functions γ_M and γ_A work as the “interpreter,” which transforms a valid configuration of A into that of A^* under the assumption that the simulated agent executing A^* stays at the same location as the simulator agent executing A . The γ -configuration $\gamma(C)$ is the configuration of A^* obtained by interpretation. It may seem unusual that $\gamma(C)$ does not explicitly depend on the value m of the agent memory. However, it is indispensable that algorithm A stores the value of the agent memory of A^* into the storage of the current node of A to simulate the behavior of the simulated agent running A^* by an oblivious agent. The simulation algorithm is defined as follows:

► **Definition 4** (simulation of algorithm). *Let $A^* \in \mathcal{A}(1, \lambda^*)$ be an algorithm. We consider that an algorithm $A \in \mathcal{A}(0, \lambda)$ simulates A^* in G , if there exists a pair of mappings $\gamma = (\gamma_M, \gamma_A)$ of Definition 3 that satisfy the following condition:*

Let v be any node of G , $E(G, v, A) = (C_t)_{t \geq 0}$, and $E(G, v, A^) = (C_t^*)_{t \geq 0}$. There uniquely exists the monotonically increasing sequence of round numbers $T(0), T(1), T(2), \dots$ such that $T(0) = 0$ holds and $\gamma(C_{T(t)}) = C_t^*$ holds for each $t \geq 0$. That is, $E(G, v, A^*) = \gamma(C_{T(0)}), \gamma(C_{T(1)}), \gamma(C_{T(2)}), \dots$ holds.*

Finally, we defined the round complexity of the simulation as follows:

► **Definition 5.** *Let $A \in \mathcal{A}(0, \lambda)$ be the algorithm that simulates $A^* \in \mathcal{A}(1, \lambda^*)$. Then, the time required to simulate round t is defined as $T(t+1) - T(t)$. If $\max_{t \geq 0} (T(t+1) - T(t)) = \text{poly}(n)$ holds for any $t \geq 0$, we consider that A is a polynomial-time simulator of A^* in G .*

3 Our Algorithm

We show a polynomial-time simulator $A \in \mathcal{A}(0, \lambda)$ that simulates any one-bit agent algorithm $A^* \in \mathcal{A}(1, \lambda^*)$ in any 2-edge-connected graph G . In the following argument, we denote the oblivious agent executing A by the symbol a and the one-bit simulated agent executing A^* by a^* . We also denote $A^* = (m^*, M^*, (M')^*, l^*, \phi^*)$. The storage size of algorithm A is denoted by λ , which satisfies $\lambda = \lambda^* + O(\log \Delta)$.

3.1 Mappings and Variables on Storage

We first introduce all the variables stored in the storage of each node in Table 1. Each variable is referred to with an additional subscript to clarify the node that stores it. For example, $sloc_i$ denotes the variable $sloc$ stored in node i . For a tuple of variables (X, Y) (where X and Y are the names of some variables), we also use the notation $(X, Y)_i$ to represent (X_i, Y_i) .

To formalize the correctness criteria of our simulation algorithm, we introduce *legal configurations*, defined as those satisfying $(dfsstat, sim, smemupd)_i = (0, 0, 0)$ for all nodes $i \in V$, $sloc_v = 1$ for the current location v of a , and $sloc_i = 0$ for all other nodes $i \neq v$. Our algorithm satisfies the condition that the sequence of all legal configurations appearing during the execution of A forms the simulated execution of A^* . More precisely, let $E(G, v, A) = (C_t)_{t \geq 0}$ and t_0, t_1, t_2, \dots be the maximal sequence of the round numbers such that C_{t_i} is legal. Then, our simulation algorithm is correct with respect to $T(i) = t_i$ and $\gamma = (\gamma_M, \gamma_A)$ such that $\gamma_M(M_i) = svars_i$ and $\gamma_A(M_i) = (smem_i, spin_i)$ hold.

Let \mathcal{C} be a set of all valid legal configurations. Our algorithm starts from any configuration $C \in \mathcal{C}$ and guarantees that the legal configuration $C' \in \mathcal{C}$ satisfies $\gamma(C) \xrightarrow{\phi^*.G} \gamma(C')$ without explicit termination. Evidently, this algorithm provides simulated execution $E(G, v, A^*)$: Following the definition of the initial values specified in Table 1, the initial configuration $C_0 = C_{t_0}$ of the corresponding execution $E(G, v, A)$ for any $v \in V$ is legal, and $\gamma(C_0)$ becomes

■ **Table 1** Variables used in our simulation algorithm. The initial location of a (and a^*) is referred to as $v \in V$.

Name	Role
<i>sloc</i>	The binary flag representing the current location of a^* , i.e., $sloc_i = 1$ implies that the simulated agent a^* is currently at node i . At any time, at most one node i satisfies $sloc_i = 1$. When $sloc_i = 0$ holds for every node i , the algorithm is simulating the movement of a^* . Initially, $sloc_v = 1$ for the initial location $v \in V$ of a and a^* , and $sloc_j = 0$ for any $j \neq v$.
<i>smem</i>	The variable storing the current memory value of a^* at the current node of a^* . It can store an arbitrary value at any other node.
<i>smemupd</i>	The flag indicating the completion of the information transfer along the constructed ITC. At the beginning of the one-round simulation of A^* , the value is 0 for all nodes. When a propagates a binary memory value m' of a^* along the ITC, the node i that has already received m' sets $smemupd_i = 1$. The initial value is zero for all nodes.
<i>spin</i>	The entry port number of a^* at the current node. It can store an arbitrary value at any other node. Initially, an arbitrary value is stored.
<i>spout</i>	The variable for storing the computed outgoing port number of a^* at the current node. It can store an arbitrary value at any other node.
<i>svars</i>	The storage for the simulated algorithm A^* . The initial value is the one specified by the algorithm A^* .
<i>dfsstat</i>	The three-state variable used inside the DFS subroutine for constructing the ITC. If the agent a does not execute the DFS subroutine, the value zero is stored at all nodes. The condition $dfsstat_i = 1$ implies that the agent a has visited i , but i still has the ports not probed yet, and $dfsstat_i = 2$ implies that all the ports of i have been probed. The initial value is zero at all nodes.
<i>par</i>	The variable storing the port number to the parent of the DFS tree. After the construction of an ITC, this variable represents the predecessor of the ITC. The initial value is arbitrary.
<i>cld</i>	The counter indicating the ports that have been already probed: $cld_i = x$ implies that up to the $(x - 1)$ -th port of the node i has been probed. After the construction of an ITC, this variable represents the successor in the ITC. The initial value is zero for all nodes.
<i>sim</i>	The flag indicating the subtask in which the agent a is engaged. If $sim_i = 0$ holds for all nodes i , the agent is executing the DFS subroutine for constructing an ITC. Once an ITC is constructed, $sim_i = 1$ holds if and only if i is contained in the constructed ITC.
<i>lastin/out</i>	In execution of the DFS subroutine, these variables manage the latest entry/outgoing port numbers at each node. The stored information is used for deleting the “garbage” information in the storage left by the DFS subtask.

the initial configuration of $E(G, v, A^*)$. The algorithm then generates the configuration C_{t_1} such that $\gamma(C_{t_0}) \xrightarrow{\phi^*, G} \gamma(C_{t_1})$ holds. The algorithm immediately starts the next round of simulations, which generates C_{t_2} because C_{t_1} is also legal. A simulates the execution of $E(G, v, A^*)$ by repeating this process.

3.2 Overviews of Our Algorithm

First, we explain the high-level structure of the proposed algorithm. The simulator algorithm consists of five subtasks: local computation, DFS, clean-up, memory transfer, and move-and-reset phases. We denote by s the node where agent a remains at the beginning of the one-round simulation. Because the configuration at the beginning is legal, the simulated agent a^* also stays at s by definition. We present an outline of each phase as follows:

Algorithm 1 Main Part and *LocalComp()*.

```

1:  $p_{in} \leftarrow$  (entry port number)
2:  $\Delta_i \leftarrow$  (the degree of the node  $i$ )
3: if  $smemupd_i = 1$  then
4:   MoveReset()
5: else if  $dfsstat_i > 0$  then
6:   if  $sim_i = 1 \vee p_{in} = par_i \vee sloc_i = 1$  then
7:     CleanUp()
8:   else
9:     DFS()
10: else if  $sim_i = 1$  then
11:   TransMem()
12: else
13:   if  $sloc_i = 1$  then  $\triangleright$  LocalComp()
14:      $(spout, svars, smem)_i \leftarrow \phi^*(\Delta_i, spin_i, svars_i, smem_i)$ 
15:   DFS()

```

Local Computation Phase: This phase was executed once at the beginning of the one-round simulation of A^* . The simulator agent locally simulates the local computation of A^* at node s (i.e., the location of a^*) and then updates the corresponding storage variables. After finishing the local computation phase, agent a immediately proceeds to the DFS phase.

DFS Phase: Let t be the node to which the simulated agent a^* moves. Agent a executes the DFS from s with the first traversal edge (s, t) until it revisits s (i.e., the node such that the variable $sloc$ stores one). This determines a cycle containing edge (s, t) , which becomes the ITC.

Clean-up Phase: The agent repeats the same DFS traversal again from s (with the first traversal edge (s, t)) to revisit s to reset the storage values left in the DFS phase, except for the one necessary to recognize the ITC.

Memory Transfer Phase: The agent circulates the ITC in the direction based on the memory value $smem_s$ of simulated agent a^* .

Move-and-Reset Phase: The agent moves simulated agent a^* from s to t by setting $sloc_s = 0$ and $sloc_t = 1$. It then resets the storage of all nodes in the ITC to recover the legality of the configurations.

The agent determines the phase in which it is currently running based on the storage values $(dfsstat, sim, smemupd, sloc, par)$ of the current location and the current value of p_{in} . The details of this are explained in the following section.

3.3 Details of Our Algorithm

In this subsection, we present the details of each phase. Pseudocodes are presented in Algorithms 1–5. Algorithm 1 presents the main part, including the local computation phase. We denote the procedures of the five phases above as *LocalComp()*, *DFS()*, *CleanUp()*, *TransMem()*, and *MoveReset()*, respectively. The procedure *LocalComp()* is described in line 14 of Algorithm 1 (see the comment in the pseudocode), and all other phases are separately described in Algorithms 2–5. Hereinafter, we refer to a line Y or lines Y – Z of Algorithm X

11:10 Computational Power of a Single Oblivious Mobile Agent

■ Algorithm 2 $DFS()$.

```

1:  $lastin_i \leftarrow p_{in}$ 
2: if  $sloc_i = 1$  then
3:    $(dfsstat, cld, lastout)_i \leftarrow (1, spout_i, spout_i)$ 
4: else if  $dfsstat_i = 0$  then ▷ visiting by forward
5:    $(dfsstat, par)_i \leftarrow (1, p_{in})$ 
6:   if  $par_i = 0$  then ▷ skip the parent port
7:      $(cld, lastout)_i \leftarrow (1, 1)$ 
8:   else
9:      $(cld, lastout)_i \leftarrow (0, 0)$ 
10: else if  $dfsstat_i = 1 \wedge p_{in} = cld_i$  then ▷ visiting by backtrack
11:   if  $par_i = p_{in} + 1$  then ▷ skip the parent port
12:      $cld_i \leftarrow p_{in} + 2$ 
13:   else
14:      $cld_i \leftarrow p_{in} + 1$ 
15:   if  $cld_i < \Delta_i$  then ▷ unprobed ports exist
16:      $lastout_i \leftarrow cld_i$ 
17:   else ▷ no unprobed port exists
18:      $(lastout, dfsstat)_i \leftarrow (par_i, 2)$ 
19: else ▷ invoking backtrack
20:    $lastout_i \leftarrow p_{in}$ 
21: Move to  $\pi_i(lastout_i)$ 

```

as (X, Y) or $(X, Y-Z)$. In all the pseudocodes, the current location of agent a is referred to as i . As in Section 3.2, we denote the initial location and destination of a^* in the simulated round by s and t , respectively.

Main part and local computation phase

The main part shown in Algorithm 1 calls for an appropriate procedure according to the local storage value of node i . The local computation phase is executed if $(dfsstat, sim, smemupd, sloc)_i = (0, 0, 0, 1)$ holds. As the initial configuration is legal, every node i satisfies $(dfsstat, sim, smemupd)_i = (0, 0, 0)$. We also have $sloc_s = 1$ and $sloc_i = 0$ for all $i \neq s$ based on the constraint of the variable $sloc$. According to the definition of $sloc$, the current location of a^* is also at s , and $smem_s$, $spin_s$, and $svars_s$ store the situation of agent a^* in the simulated execution. The simulator agent a locally simulates the local computation of A^* by a^* at node s and stores the computed results in $spout_s$, $svars_s$, and $smem_s$ (1,14). The DFS phase was invoked immediately after the local computation phase (1,15).

DFS phase

We show the pseudocode for the DFS phase in Algorithm 2. This phase is executed if one of the following conditions is satisfied:

- $(dfsstat, sim, smemupd, sloc)_i = (0, 0, 0, 0)$ or immediately after the local computation phase (1,15),
- $dfsstat_i > 0$, $(sim, smemupd, sloc)_i = (0, 0, 0)$, and $p_{in} \neq par_i$ (1,9).

Intuitively, the first and second conditions are applied to the cases in which the agent visits i first and re-visits i . Lines (2,2-3) are for the exceptional behavior of the first invocation of $DFS()$ immediately after the local computation phase. Then, the choice of the probed port must correspond to edge (s, t) , that is, the port stored in $spout_i$. Note that, when the agent visits node $i = s$ again, $dfsstat_i > 0$ and $sloc_i = 1$ are satisfied, and (2,2-3) are not executed because the main routine invokes $CleanUp()$ in (1,7). Because the simulation starts from a legal configuration, the variables sim_i , $smemupd_i$, and $sloc_i$ for any $i \neq s$ store 0 at the beginning of the DFS phase and are not modified in the subroutine $DFS()$. Consequently, the DFS phase continues if the agent visits nodes other than s . Throughout the DFS phase, the agent visiting node i stores the entry and outgoing port numbers in $lastin$ and $lastout$, respectively. Line (2,1) is for storing the entry port number. To store the outgoing port number, the agent first writes the port number to which the agent will move into $lastout_i$ (lines 3, 7, 9, 16, 18, and 20) and finally moves to the port indicated by $lastout_i$ (2,21). The information of $lastin$ and $lastout$ is used in procedure $CleanUp()$. During the DFS phase, the agent explored G following the standard DFS. Specifically, it probes the ports of the visited node individually in ascending order, which enables the agent to recognize the ports already probed by storing only one port number. The latest probed port number was stored in cld_i . When the agent returns to the current node i through an edge with port number cld_i by backtrack, cld_i is incremented. If $cld_i + 1$ is the port indicating the parent of the DFS tree, then it is skipped (2,6 and 2,11).¹ If the value of cld_i after the increment exceeds $\Delta_i - 1$, the agent has no neighbor of i to be checked and thus performs the backtrack with setting $dfsstat_i = 2$ (2,18). When the agent visits i more than once, the variable $dfsstat_i$ stores a nonzero value at the second or later visit. Then, the condition $(dfsstat, sim, smemupd)_i = (1, 0, 0)$ is satisfied, and the entry port number is necessarily different from par_i because, in such a case, the agent traverses a non-DFS-tree edge or performs the backtrack. Consequently, the agent can distinguish whether it visits i by forward or non-forward movement under conditions $dfsstat_i = 1$ and $p_{in} \neq par_i$ (2,10).

Clean-up Phase

We show the pseudocode of the clean-up phase in Algorithm 3. This phase is executed if $dfsstat_i > 0$ holds, and either $sim_i = 1$, $p_{in} = par_i$, or $sloc_i = 1$ is satisfied (1,5-6). This phase begins immediately after the agent returns to s during the DFS phase. At the beginning of this phase, the set of nodes that satisfy $dfsstat = 1$ forms an ITC. It first updates $par_i (= par_s)$ by p_{in} , resulting in consistent orientation of the ITC by the variable par . In addition, the variable cld for all nodes in the ITC presents the inverse orientation. The goal of this phase is to reset all the nodes i with $dfsstat_i > 0$ by setting $dfsstat_i = 0$ and marking the nodes i in the ITC with $sim_i = 1$. The agent in the clean-up phase performs DFS up to the first revisit of s similar to the DFS phase. Every node i visited in the clean-up phase must satisfy $dfsstat_i > 0$ (every node i satisfying $dfsstat_i > 0$ is necessarily visited) because the DFS traversal in this phase is the same as that in the DFS phase. The main technical challenge is to make the agent distinguish between the DFS and clean-up phases, particularly at the first visit of i . This issue was resolved as follows: In the DFS phase, the

¹ Since the standard (centralized) DFS always moves to a neighboring unvisited node, agent-based algorithms cannot identify whether a neighbor is already visited. Hence, the agent must check all the neighbors. Then, the following case can occur: the agent exits from i through port p , but the destination is already visited. Thus, it returns immediately to i . In our algorithm, this case is treated as a backtrack.

11:12 Computational Power of a Single Oblivious Mobile Agent

■ **Algorithm 3** *CleanUp()*.

```

1: if  $sim_i = 0$  then
2:   if  $(dfsstat, sloc)_i = (1, 1)$  then
3:      $(par, dfsstat, sim)_i \leftarrow (p_{in}, 0, 1)$ 
4:     Move to  $\pi_i(cld_i)$ 
5:   else
6:      $sim_i \leftarrow 1$ 
7:     if  $par_i = 0$  then ▷ skip the parent port
8:        $cld_i \leftarrow 1$ 
9:     else
10:       $cld_i \leftarrow 0$ 
11:      if  $(dfsstat, lastin, lastout)_i = (1, p_{in}, cld_i)$  then ▷ initialization
12:         $dfsstat_i \leftarrow 0$ 
13:        Move to  $\pi_i(cld_i)$ 
14:      else if  $p_{in} = cld_i$  then ▷ visiting by backtrack
15:        if  $par_i = p_{in} + 1$  then ▷ skip the parent port
16:           $cld_i \leftarrow cld_i + 2$ 
17:        else
18:           $cld \leftarrow cld_i + 1$ 
19:        if  $dfsstat_i = 1$  then
20:          if  $(lastin, lastout)_i = (p_{in}, cld_i)$  then ▷ initialization
21:             $dfsstat_i \leftarrow 0$ 
22:            Move to  $\pi_i(cld_i)$ 
23:          else if  $cld_i < \Delta_i$  then ▷ unprobed ports exist
24:            Move to  $\pi_i(cld_i)$ 
25:          else ▷ no unprobed port exists
26:            if  $(lastin, lastout)_i = (p_{in}, par_i)$  then
27:               $(dfsstat, sim)_i \leftarrow (0, 0)$  ▷ initialization
28:              Move to  $\pi_i(par_i)$ 
29:          else ▷ invoking backtrack
30:            if  $(lastin, lastout)_i = (p_{in}, p_{in})$  then
31:              if  $dfsstat_i = 1$  then
32:                 $dfsstat_i \leftarrow 0$  ▷ initialization
33:              else
34:                 $(dfsstat, sim)_i \leftarrow (0, 0)$  ▷ initialization
35:              Move to  $\pi_i(p_{in})$ 

```

agent never enters node i with $dfsstat_i > 0$ (i.e., the second or later visit of i) through the edge with port number par_i . Hence, if $p_{in} = par_i$ and $dfsstat_i > 0$ holds in the clean-up phase, agent a correctly recognizes that it enters i by the forward movement of the DFS in the clean-up phase. The value update of sim_i from zero to one occurs when the agent visits i first in the clean-up phase (3,6), which enables the agent to distinguish between the clean-up and DFS phases when revisiting i . During the execution, the agent must initialize the information of the storages written in the DFS phase, except for the information for the ITC (i.e., the information of $sim_i = 1$ for i in the ITC). However, resetting $dfsstat_i$ can result in the loss

Algorithm 4 *TransMem()*.

```

1: if  $sloc_i = 1$  then
2:    $smemupd_i \leftarrow 1$ 
3:   if  $smem_i = 0$  then  $\triangleright$  transfer 0 ( $p_{in} = par_i$ )
4:     Move to  $\pi_i(cld_i)$ 
5:   else  $\triangleright$  transfer 1 ( $p_{in} = cld_i$ )
6:     Move to  $\pi_i(par_i)$ 
7: else  $\triangleright$  receive the transferred value
8:    $smemupd_i \leftarrow 1$ 
9:   if  $p_{in} = par_i$  then  $\triangleright$  transfer 0
10:     $smem_i \leftarrow 0$ 
11:    Move to  $\pi_i(cld_i)$ 
12:  else  $\triangleright$  transfer 1
13:     $smem_i \leftarrow 1$ 
14:    Move to  $\pi_i(par_i)$ 

```

of recognition in the current phase when the agent visits i again in the subsequent execution of the clean-up. To avoid this, the agent checks whether the current visit of i is the final one by comparing the number of entry and outgoing ports to the values of $(lastin, lastout)_i$ left in the DFS phase. One can show that the agent never visits i again in the subsequent execution of the clean-up phase if they are the same (formally proved as Claim 9). Finally, when the agent returns to s , all unnecessary information is reset, and the agent proceeds to the memory transfer phase under the condition $(dfsstat, sim, smemupd)_s = (0, 1, 0)$ (1,10). It is noteworthy that $sim_j = 1$ holds if and only if j is contained in the constructed ITC.

Memory Transfer Phase

We show the pseudocode of the memory transfer phase in Algorithm 4. In this phase, the agent circulates ITC. Recall that node i in the ITC is identified by condition $sim_i = 1$. The agent recognizes that the current round is in the memory transfer phase if $(sim, dfsstat)_i = (1, 0)$ holds (1,10-11). It sets $smemupd_i = 1$ and moves to one of i 's neighbors in the ITC, specified by the value of $smem_i$. As stated above, the variables par and cld at the nodes in the ITC present two opposite orientations of the ITC; thus, we observe that agent a transfers 0 if $p_{in} = par_i$ and 1 otherwise. The transferred value is written for all nodes in the ITC (4,10 and 4,13). If $smemupd_i = 1$ at the visited node, the agent detects the termination of circulating the ITC, which triggers the following move-and-reset phase (1,3-4).

Move-and-Reset Phase

We show the pseudocode for the move-and-reset phases in Algorithm 5. This phase is executed when $smemupd_i = 1$ (1,3). First, agent a sets $sloc_s = 0$ at node s , which implies that a^* leaves s and moves to $\pi_s(spout_s) = t$. At node t , the agent updates $sloc_t$ and $spin_t$ by 1 and p_{in} , respectively (5,5). The one-round simulation of a^* finishes because $smem_t$ is already updated by the memory transfer phase. The remaining task is to reset the expired information of variables sim and $smemupd$ left in the ITC. To reset this, the agent circulates the ITC following the orientation by cld . The requirement for the orientation is to make the

Algorithm 5 *MoveReset()*.

```

1: if  $sloc_i = 1$  then  $\triangleright$  leave from  $s$ 
2:    $sloc_i \leftarrow 0$ 
3:   Move to  $\pi_i(spout_i)$ 
4: else if  $p_{in} = par_i$  then  $\triangleright$  reach to  $t$ 
5:    $(sloc, spin, sim, smemupd)_i \leftarrow (1, p_{in}, 0, 0)$ 
6:   Move to  $\pi_i(par_i)$ 
7: else  $\triangleright$  initialization
8:    $(sim, smemupd)_i \leftarrow (0, 0)$ 
9:   Move to  $\pi_i(par_i)$ 

```

agent distinguish the first movement (for updating $sloc$ and $spin$) from the following reset movement. Since the port number of the edge (s, t) at t is par_t , if the agent visits t with $p_{in} = par_t$, it can recognize that the current round is for updating $sloc$ and $spin$ (5,4), in contrast with the fact that $p_{in} = cld_t$ always holds in the reset circulation. If $sloc_i = 1$ holds in the reset circulation, this implies that $i = t$, and $(dfsstat, sim, smemupd)_i = (0, 0, 0)$ also holds for all i . The configuration becomes legal since all nodes not in the ITC have been reset correctly in the clean-up phase.

3.4 Correctness

In this section, we prove that each procedure correctly executes the corresponding phase and bound their running times. To prove this, we first introduce the standard (centralized) DFS process under the following conditions:

1. Initially, the search head points at s and moves to t at the first neighborhood search.
2. When visiting $i \neq s$, the search head sequentially probes its neighbors along the order of the corresponding port numberings.
3. If the search head moves to a node i already searched, it immediately goes back to the previous node.
4. The search process terminates when the search head reaches s again.

where s and t denote the nodes defined in the previous section. We define V_{dfs} as the set of all nodes reached by the search head in this process and $S = i_0, i_1, \dots, i_M$ as the trajectory of the search head. Note that, in this sequence, one node can appear more than once. In addition, for node s' visited immediately before re-visiting s , we define V_{ITC} as the set of nodes on the path from s to s' in the DFS tree. All V_{dfs} , S , and V_{ITC} are uniquely determined only by nodes s and t , graph G , and its port numbering functions Π .

For any phase X , if the agent invokes the procedure corresponding to X in round r , we say that round r is of phase X . To make it well-defined, we slightly modified our algorithm by separating the round executing *LocalComp()* and executing *DFS()*. That is, after the local computation phase, the agent finishes the current round with no movement, and in the next round, it invokes *DFS()*. Trivially, the correctness of the modified algorithm was lower than that of the original algorithm. Let r_X be the first round of phase X during the execution. The *sub-execution* of phase X is defined as the maximal period from r_X in which every round is of phase X . For the final round r' of the sub-execution of X , the configuration at the beginning of round $r' + 1$ is called the *resultant configuration* of X .

A node i is called *legal* if it satisfies $(dfsstat, sim, smemupd)_i = (0, 0, 0)$. First, we present the correctness criteria for each phase.

► **Definition 6** (correctness of phases). *Each phase is called correct if the sub-execution of the phase satisfies the following condition:*

Local Computation Phase *Starting from any valid legal configuration $C \in \mathcal{C}$, the resultant configuration C' satisfies $(spout, svars, smem)_s = \phi^*(\Delta_{l^*}, p_{in}^*, M_{l^*}, m^*)$ and $C' \in \mathcal{C}$.*

DFS Phase *Starting from the resultant configuration of any correct local computation phase, the sub-execution reaches the resultant configuration satisfying as follows:*

1. *For any $u \in V$, $(sim, smemupd)_u = (0, 0)$ holds.*
2. *For any $u \in V$, $dfsstat_u = 1$ holds if $u \in V_{ITC}$, $dfsstat_u = 2$ holds if $u \in V_{dfs} \setminus V_{ITC}$, or $dfsstat_u = 0$ holds otherwise.*
3. *The variables $(spout, svars, smem)_s$ store the same values as the starting configuration.*

Clean-up Phase *Starting from the resultant configuration of any correct DFS phase, the sub-execution reaches the resultant configuration satisfying as follows:*

1. *For each $u \in V_{ITC}$, $(dfsstat, sim, smemupd)_u = (0, 1, 0)$ holds.*
2. *For each $u \in V_{ITC}$, $\pi_u(par_u)$ and $\pi_u(cld_u)$ are also contained in V_{ITC} , and letting $\pi_u(par_u) = x$, $\pi_x(cld_x) = u$ is satisfied.*
3. *Any node $u \notin V_{ITC}$ is legal, and $(spout, svars, smem)_s$ store the same values as the starting configuration.*

Memory Transfer Phase *Starting from the resultant configuration of any correct clean-up phase, the sub-execution reaches the resultant configuration satisfying as follows:*

1. *For any $u \in V_{ITC}$, $(dfsstat, sim, smemupd)_u = (0, 1, 1)$ and $smem_u = smem_s$ holds.*
2. *Any node $w \notin V_{ITC}$ is legal, and $(spout, svars, smem)_s$ store the same values as the starting configuration.*

Move-and-Reset Phase *Starting from the resultant configuration of any correct memory transfer phase, the sub-execution reaches the resultant configuration satisfying as follows:*

1. *Any node $i \in V$ is legal, and $sloc_t = 1$ holds only for t .*
2. *The variable $spin_t$ stores $\pi_t^{-1}(s)$ (i.e., the entry port number of a^*).*
3. *$smem_t = smem_s$.*

It is easy to verify that the resultant configuration of the phase triggers the subsequent phase. Hence, one can trivially deduce the correctness of our algorithm based on the correctness of all phases. We focus on the correctness of the remaining four phases because procedure *LocalComp()* is trivially correct.

► **Lemma 7.** *The DFS phase is correct.*

Proof. Procedure *DFS()* does not modify *sim* or *smemupd*. Because we assume that the initial configuration is legal, this proof does not consider the conditions of *sim* and *smemupd*, but focuses on the conditions of variable *dfsstat*. We first show that the agent moves along the trajectory i_0, i_1, \dots, i_M during the sub-execution of the DFS phase. If the procedure *DFS()* is invoked in some rounds, it chooses the next visited node in the same way as the centralized DFS explained above. Hence, it suffices to show that the DFS phase continues until the round when the agent visits s . If $dfsstat_i = 0$ holds, procedure *DFS()* is necessarily executed (1,12 and 1,15). In addition, during the DFS phase, $sim_i = 0$ and $p_{in} \neq par_i$ hold for any $i \in V$ such that $dfsstat_i > 0$ because $dfsstat_i > 0$ implies that i is already visited in the DFS phase. Thus, par_i appropriately points to the parent of i in the DFS tree (evidently, the DFS search never visits node i twice through the edge from the parent). Thus, the sub-execution of the DFS phase terminates if and only if $sloc_i = 1$ and $dfsstat_i > 0$ hold (1,5-7). When agent a revisits s , $dfsstat_s = 1$ and $sloc_s = 1$ hold. Hence, the DFS

11:16 Computational Power of a Single Oblivious Mobile Agent

phase continues until node s has been revisited. We show that the resultant configuration satisfies the condition of Definition 6. Let s' be the last newly visited node in the DFS phase (i.e., the node immediately before the revisit of s). Variable $dfsstat_i$ is set to 2 (and the agent backtracks) if and only if all ports except that par_i are probed. According to the definition of V_{ITC} , the agent does not backtrack at any node in V_{ITC} . That is, at any node $u \in V_{ITC}$, the agent does not return to u through the edge indicated by cl_d_u . This implies that $dfsstat_u = 1$ holds and the condition of Definition 6 is satisfied at any $u \in V_{ITC}$. ◀

The proof of the clean-up phase contains a slight technical non-triviality in the distinction of the DFS phase.

► **Lemma 8.** *The clean-up phase is correct.*

Proof. As a proof of Lemma 7, we do not consider the condition of the variable $smemupd$. In this phase, the agent traces the same route as in the DFS phase if resetting $dfsstat$ and sim at node i occurs correctly at the last visit of i . We prove this by inducing the indices of trajectory i_0, i_1, \dots, i_M . Let $c_0, c_1, \dots, c_{M'}$ be the trajectory of the agent in the clean-up phase. We show that $c_k = i_k$ holds and c_k satisfies the conditions for executing the clean-up phase for any k . Because this phase starts at node s , $c_0 = i_0$ and the condition for executing the clean-up phase is satisfied. Then, as the induction hypothesis, we assume that $c_k = i_k$ is satisfied for each $k \in [0, j]$ and the condition for executing the clean-up phase is satisfied at c_j . Because the method of deciding the next probed neighbor is the same as the DFS phase, $c_{j+1} = i_{j+1}$ is satisfied. Now, we have to show that the condition for executing the clean-up phase is satisfied at c_{j+1} . If c_{j+1} is visited for the first time in this phase, $p_{in} = par_{c_{j+1}}$ holds, and the agent sets $sim_{c_{j+1}} = 1$. Otherwise, the proof is performed if $sim_{c_{j+1}} = 1$ holds; that is, c_{j+1} is not reset up to round j . As we have already shown in Lemma 7 that i_0, i_1, \dots, i_M is also the trajectory of the agent in the DFS phase, the agent stores the entry/outgoing port numbers in the variables $lastin$ and $lastout$ of node i_k in the k -th round of the DFS phase. Let L_k be a pair of values stored in $(lastin, lastout)_{i_k}$ when visiting i_k . Then, we have the following claim.

▷ **Claim 9.** For any $k', k \in [0, M]$, $L_{k'} \neq L_k$ holds, if $i_k = i_{k'}$ holds.

Proof. Assume $k' < k$ without loss of generality, and let $u = i_{k'}$ for short. We denote the values of $lastin$ and $lastout$ in L_x as $lastin(x)$ and $lastout(x)$ respectively. First, consider the case in which round k' is the first visit of u in the DFS process. Subsequently, $lastin(k') = par_u$ holds. Because the agent visits u from its parent only at the first visit, we have $lastin(k) \neq par_u$, and thus the claim holds. Next, consider the case in which round k' is not the first visit of u . If $lastin(k') = lastin(k)$ holds, then the agent visits u in rounds k' and k through the same edge (w, u) . There exist two scenarios in which the agent traverses edge (w, u) in the second or later visit of u : In the first scenario, the agent moves forward from u to w , but w has already been visited and thus returns to u by backtracking. Second, the agent moves forward from w to u . These two are all possible scenarios, and thus they occur at rounds $k' - 1$ and $k - 1$, respectively. While the agent leaves from u with the port necessarily different from $\pi_u^{-1}(w)$ in the first scenario, it leaves with the edge to w in the second scenario (because the next movement is backtracked to w). That is, the two scenarios choose different numbers of outgoing ports to leave u . Thus, $lastout(k') \neq lastout(k)$ holds. The claim holds. ◀

Suppose that, for contradiction, c_{j+1} is reset in round $k \leq j$. Because $c_{j+1} = i_{j+1}$ holds, round k is not the last visit of $c_k = c_{j+1}$. Let $c_{k'}$ be the last visit of c_{j+1} ($k' > k$), and Claim 9 implies that $L_{k'} \neq L_k$. This contradicts the assumption that c_{j+1} is reset to the

round k . Consequently, c_{j+1} is not reset to round j and it can be concluded that the trace of the agent is the same as that of the DFS phase. In addition, it has been proven that reset at node i occurs at the last visit of i . From these two facts, it is easy to verify that the resultant configuration satisfies the condition of Definition 6. The lemma is proved. ◀

The correctness of the memory transfer and the move-and-reset phases is relatively straightforward.

► **Lemma 10.** *The memory transfer phase is correct.*

Proof. Under the condition of the initial configuration, the nodes i with $sim_i = 1$ induce ITC, and variables par and cld provide two opposite orientations. Hence, evidently, the agent correctly circulates the ITC in the memory transfer phase and terminates at the initial position s (note that, at the beginning of the phase, the agent sets $smemupd_s = 1$; thus, when it returns to s , the memory transfer phase necessarily terminates). Assuming that variables par and cld provide two opposite orientations if the agent leaves s through port par_s , it enters the next node x through port cld_x and vice versa. This observation implies that the simulated memory value $smem_s$ is correctly transferred to all nodes in the ITC, and the condition of the resultant configuration in Definition 6 is satisfied. ◀

► **Lemma 11.** *The move-and-reset phase is correct.*

Proof. Under the condition of the initial configuration, all nodes not in V_{ITC} have already been legal (i.e., $(dfsstat, sim, smemupd) = (0, 0, 0)$). As explained in Section 3.3, the agent correctly circulates the ITC along the orientation using cld and resets sim and $smemupd$. Because $dfsstat$ is already zero, we can conclude that the resulting configuration is legal. The remaining issue is to show that it is the configuration interpreted as that for A^* after a one-round simulation. It has already been shown that the memory value is transferred correctly and $svars_s$ is updated correctly in the local computation phase. In addition, the agent updates $spin_t$ by p_{in} at the first movement from s to t in the move-and-reset phase; that is, $spin_t$ stores the port number of edge (s, t) . Because a^* moves from s to t in this round, $spin_t$ is equal to the entry port number of a^* in the next simulated round. ◀

By combining all of the lemmas above, we obtain the main theorem.

► **Theorem 12.** *Algorithm A shown in Algorithm 1 correctly simulates A^* in any 2-edge-connected graph G . The round complexity of the algorithm to simulate one round of A^* is $O(n^2)$. The required storage size is $O(\lambda^* + \log \Delta)$ bits per node.*

Proof. The correctness clearly follows Definition 6, and all phases are correct. To limit time complexity, we bound the running time of each phase. The first two phases (i.e., DFS and clean-up) depend on the running time of DFS. To execute DFS for the graph such that the number of nodes and edges are $|V|$ and $|E|$, respectively, it needs $O(|E|) = O(n^2)$ rounds. The running time of the remaining two phases was bounded by $O(|V_{ITC}|) = O(n)$ rounds. Hence, the total running time (per round of simulation) is $O(n^2)$.

The storage size consumes λ^* bits for $svars$, which corresponds to the storage of A^* . The sizes of the other variables are bounded by $O(\log \Delta)$. The storage size was $O(\lambda^* + \log \Delta)$ bits per node. ◀

4 Conclusion

In this study, we demonstrated the equivalence of the computational power of a single oblivious agent and a single one-bit memory agent in a 2-edge-connected graph with $O(\log \Delta)$ -bit storage by presenting the algorithm $A \in \mathcal{A}(0, \lambda)$ that simulates any $A^* \in \mathcal{A}(1, \lambda^*)$ on 2-edge-connected graphs with $\lambda = \lambda^* + O(\log \Delta)$. The time overhead of our simulation algorithm was $O(n^2)$ per round. That is, if the original algorithm A^* runs in polynomial time, then our simulator also works in polynomial time. Finally, we conclude this study by explaining the open problem. The primary open problem deduced from our results is whether an oblivious agent can simulate any one-bit agent on 2-edge-connected graphs under the assumption that only $O(1)$ -bit storage per node is available. As a weaker form of this problem, 2-edge-connected graphs with $O(1)$ -bit storage per node can be also explored.

References

- 1 Reuven Cohen, Pierre Fraigniaud, David Ilcinkas, Amos Korman, and David Peleg. Label-guided graph exploration by a finite automaton. *ACM Trans. Algorithms*, 4(4):42:1–42:18, 2008. doi:10.1145/1383369.1383373.
- 2 Ajoy Kumar Datta, Anissa Lamani, Lawrence L. Larmore, and Franck Petit. Enabling ring exploration with myopic oblivious robots. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop, IPDPS 2015, Hyderabad, India, May 25-29, 2015*, pages 490–499. IEEE Computer Society, 2015. doi:10.1109/IPDPSW.2015.137.
- 3 Yoann Dieudonné, Shlomi Dolev, Franck Petit, and Michael Segal. Explicit communication among stigmergic robots. *Int. J. Found. Comput. Sci.*, 30(2):315–332, 2019. doi:10.1142/S0129054119500072.
- 4 Yann Disser, Jan Hackfeld, and Max Klimm. Undirected graph exploration with $\Theta(\log \log n)$ pebbles. In Robert Krauthgamer, editor, *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 25–39. SIAM, 2016. doi:10.1137/1.9781611974331.ch3.
- 5 Pierre Fraigniaud, David Ilcinkas, Guy Peer, Andrzej Pelc, and David Peleg. Graph exploration by a finite automaton. *Theor. Comput. Sci.*, 345(2-3):331–344, 2005. doi:10.1016/j.tcs.2005.07.014.
- 6 Taisuke Izumi, Kazuki Kakizawa, Yuya Kawabata, Naoki Kitamura, and Toshimitsu Masuzawa. Deciding a graph property by a single mobile agent: One-bit memory suffices, 2022. doi:10.48550/arXiv.2209.01906.
- 7 Sayaka Kamei. Autonomous distributed systems of myopic mobile robots with lights. In *ICDCN '21: International Conference on Distributed Computing and Networking, Virtual Event, Nara, Japan, January 5-8, 2021*, page 5. ACM, 2021. doi:10.1145/3427796.3432715.
- 8 Sayaka Kamei, Anissa Lamani, Fukuhito Ooshita, Sébastien Tixeuil, and Koichi Wada. Gathering on rings for myopic asynchronous robots with lights. 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 27:1–27:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICs.OPODIS.2019.27.
- 9 Fukuhito Ooshita and Sébastien Tixeuil. Ring exploration with myopic luminous robots. *Inf. Comput.*, 285(Part):104702, 2022. doi:10.1016/j.ic.2021.104702.
- 10 Omer Reingold. Undirected connectivity in log-space. *J. ACM*, 55(4):17:1–17:24, 2008. doi:10.1145/1391289.1391291.
- 11 Yuichi Sudo, Daisuke Baba, Junya Nakamura, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. A single agent exploration in unknown undirected graphs with whiteboards. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 98-A(10):2117–2128, 2015. doi:10.1587/transfun.E98.A.2117.

Line Search for an Oblivious Moving Target

Jared Coleman   

University of Southern California, Los Angeles, CA, USA

Evangelos Kranakis   

Carleton University, Ottawa, Canada

Danny Krizanc  

Wesleyan University, Middletown, CT, USA

Oscar Morales-Ponce   

California State University, Long Beach, CA, USA

Abstract

Consider search on an infinite line involving an autonomous robot starting at the origin of the line and an oblivious moving target at initial distance $d \geq 1$ from it. The robot can change direction and move anywhere on the line with constant maximum speed 1 while the target is also moving on the line with constant speed $v > 0$ but is unable to change its speed or direction. The goal is for the robot to catch up to the target in as little time as possible.

The classic case where $v = 0$ and the target's initial distance d is unknown to the robot is the well-studied “cow-path problem”. Alpert and Gal [2] gave an optimal algorithm for the case where a target with unknown initial distance d is moving *away* from the robot with a known speed $v < 1$. In this paper we design and analyze search algorithms for the remaining possible knowledge situations, namely, when d and v are known, when v is known but d is unknown, when d is known but v is unknown, and when both v and d are unknown. Furthermore, for each of these knowledge models we consider separately the case where the target is moving away from the origin and the case where it is moving toward the origin. We design algorithms and analyze competitive ratios for all eight cases above. The resulting competitive ratios are shown to be optimal when the target is moving towards the origin as well as when v is known and the target is moving away from the origin.

2012 ACM Subject Classification Theory of computation → Online algorithms; Theory of computation → Adversary models

Keywords and phrases Infinite Line, Knowledge, Oblivious, Robot, Search, Search-Time, Speed, Target

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2022.12

Related Version *arXiv Version*: <https://doi.org/10.48550/arXiv.2211.03686> [7]

Funding *Evangelos Kranakis*: Research supported in part by NSERC Discovery grant.

1 Introduction

Search is important to many areas of computer science and mathematics and has received the attention of numerous studies. In the simplest search scenario, one is interested in the optimal trajectory of a single autonomous mobile agent (also referred to simply as a robot) tasked with finding a target placed at an unknown location on the infinite line. The line search problem is to give an algorithm for the agent so as to minimize the competitive ratio defined as the supremum over all possible target locations of the ratio of the time the agent takes to find the target and the time it would take if the target's initial position was known to the robot ahead of time. This classic problem has led to many variations (see [2] for more on its history).



© Jared Coleman, Evangelos Kranakis, Danny Krizanc, and Oscar Morales-Ponce; licensed under Creative Commons License CC-BY 4.0

26th International Conference on Principles of Distributed Systems (OPODIS 2022).

Editors: Eshcar Hillel, Roberto Palmieri, and Etienne Rivière; Article No. 12; pp. 12:1–12:19



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In this paper we consider an extension of the line search problem involving an autonomous robot and an oblivious moving target. The search is again performed on an infinite line and concerns an autonomous robot starting at the origin of the line but differs from the previously studied case in that the search is for a *moving* target whose speed and direction are not necessarily known to the searching robot. The robot starts at the origin and the target at an arbitrary distance d from the origin. The target is moving with constant speed and is oblivious in that it cannot change its speed and/or direction of movement. We consider and analyze several alternative knowledge-based scenarios in which the target's speed and initial distance from the origin may be known or unknown to the searching robot. The case where a target with unknown initial distance from the origin is moving away from the origin was solved by Alpern and Gal [2]. As far as we are aware, these are the first results for the remaining cases.

1.1 Notation and terminology

On the infinite real line, consider an autonomous robot which is initially placed at the origin whose maximum speed is 1 and an oblivious robot (also referred to as the moving target) initially placed at a distance d to the right or left of the origin and moving with constant speed $v > 0$. As is usually done in linear search and in order to avoid trivial considerations on the competitive ratio by adversarially placing the target very close to the robot, we assume that d is not smaller than the unit distance, i.e., $d \geq 1$.

The target may be moving away from or toward the origin. If it is moving away, we assume its speed is strictly less than 1 as otherwise the problem can not be solved. Further, we assume that the autonomous robot knows the direction the target is moving (away from or toward the origin). The search is completed as soon as the robot and target are co-located.

The movement of the autonomous robot is determined by a trajectory which is defined as a continuous function $t \rightarrow f(t)$, with $f(t)$ denoting the location of the robot at time t . Moreover, it is true that $|f(t) - f(t')| \leq u|t - t'|$, for all t, t' , where u is the speed of the agent (be that the searching robot or the oblivious target). The autonomous robot can move with its own constant speed and during the traversal of its trajectory it may stop and/or change direction instantaneously and at any time as specified by the search algorithm.

A search strategy is a sequence of movements followed by the robot. The competitive ratio of a search strategy X , denoted CR_X , is defined as the supremum over all possible initial target locations and speeds of the ratio of the time the agent takes to find the target and the time it would take if the target's initial position was known to the robot ahead of time. The competitive ratio of a certain type of search problem is the infimum of CR_X taken over all possible strategies X for this problem. By abuse of notation we may drop mention of X when this is easily implied from the context.

Our goal in this paper is to prove bounds on the competitive ratios of algorithms under four different knowledge models:

1. **FullKnowledge:** The robot knows both the target's speed v and its initial distance d .
2. **NoDistance:** The robot knows the target's speed v but not its initial distance d .
3. **NoSpeed:** The robot knows the target's initial distance d , but not its speed v .
4. **NoKnowledge:** The robot knows neither the target's speed v nor its initial distance d .

For all knowledge models, the robot does not know the target's initial position. We study each of the above knowledge models for the case where the target is moving toward the origin (**Toward**) and where it is moving away (**Away**) from the origin. In each case, we assume the robot knows the direction of travel of the target.

1.2 Related Work

Several research papers have considered the search problem for a robot searching for a static (fixed) target placed at an unknown location on the real line, see [3, 14]. The problem was first independently considered in a stochastic setting by Bellman and Beck in the 1960's (cf. [4, 5] as well as [3, 14]). In a deterministic setting it is now well known that the optimal trajectory for this single agent search uses a doubling strategy whose trajectory attains a competitive ratio of 9. Linear search has attracted much attention and been the focus of books including [1, 2, 15].

The case of a moving target appears to have been first considered by McCabe [13]. In that paper, the problem of searching for an oblivious target that follows a Bernoulli random walk on the integers is considered. For the case of a deterministic oblivious searcher, the only result we are aware of is found in Alpern and Gal [2]. There they consider the case where the target is moving away from the origin at a constant speed $v < 1$ which is known to the searching robot. Only the initial distance of the target is unknown. They give an algorithm with optimal competitive ratio for this case.

Our problem is reminiscent of the problem of catching a fugitive in a given domain which is generally referred to as the cops and robbers problem [6]. The main difference is that in those problems, the target (robber) is itself an autonomous agent. As a result, the techniques considered there do not apply to our case.

Our problem is also related to rendezvous (of two robots) on an infinite line but it differs because in our case only one of the robots is autonomous while the other is oblivious. Related studies on the infinite line include rendezvous with asymmetric clocks [9] and asynchronous deterministic rendezvous [12]. More recent work on linear search concerns searching for a static target by a group of cooperating robots, some of which may have suffered either crash [10] or Byzantine [8] faults.

1.3 Results of the paper

In all situations considered it is unknown to the robot whether the target is initially to the left or to the right of the origin. We analyze the competitive ratio in four situations which reflect what knowledge the robot has about the target. We present results on the **FullKnowledge** model (the robot knows v and d) in Section 2, the **NoDistance** model (the robot knows v but not d) in Section 3, the **NoSpeed** model (the robot knows d but not v) in Section 4, and the **NoKnowledge** model (the robot knows neither v nor d) in Section 5. For each of these models we study separately the case when the target is moving away or toward the origin (this knowledge being available to the robot). The results are summarized in Table 1. We conclude with a summary and additional open problems.

2 The FullKnowledge Model

We first study the model where the robot knows the target's speed v and its initial distance from the origin d .

2.1 The FullKnowledge/Away Model

For the case when the target is moving away from the origin, clearly if $v \geq 1$ then the robot can never catch the target. Thus, for this model (and all other **Away** models), we assume $v < 1$. In this section, we will analyze an algorithm where the robot chooses a direction and moves for time $\frac{d}{1-v}$. If the robot does not find the target after moving for time $\frac{d}{1-v}$ in one direction, then it changes direction and continues moving until it does.

12:4 Line Search for an Oblivious Moving Target

■ **Table 1** Table of competitive ratio bounds proven for each knowledge model for cases with the target moving away from or towards the origin with speed v and initial distance d from the robot which is moving with speed 1. Equalities indicate that tight upper and lower bounds are proven.

Knowledge	Movement	Competitive Ratio	Section
v, d	Away	$CR = 1 + \frac{2}{1-v}$	2.1
	Toward	$CR = 1 + \frac{2}{1+v}$ if $v < 1$ $CR = 1 + \frac{1}{v}$ otherwise	2.2
v	Away	$CR = 1 + 8 \frac{1+v}{(1-v)^2}$	3.1 [2]
	Toward	$CR = 1 + \frac{1}{v}$ if $v \geq \frac{1}{3}$ $CR = 1 + 8 \frac{1-v}{(1+v)^2}$ otherwise	3.2
d	Away	$CR \leq 5$ if $v \leq \frac{1}{2}$ $CR \leq 1 + 16 \frac{(\log \frac{1}{1-v})^2}{(1-v)^4}$ otherwise	4.1
	Toward	$CR = 3$	4.2
\emptyset	Away	$CR \leq 1 + \frac{16}{d} [\log \log (\max(d, \frac{1}{1-v})) + 3]$ $\cdot \max(d, \frac{1}{1-v})^8 \cdot \log^2 [\max(d, \frac{1}{1-v})]$	5.1
	Toward	$CR = 1 + \frac{1}{v}$	5.2

■ **Algorithm 1** Online Algorithm for FullKnowledge/Away Model.

-
- 1: **input:** target speed v and initial distance d
 - 2: choose any direction and go for time $\frac{d}{1-v}$
 - 3: **if** target not found **then**
 - 4: change direction and go until target is found
-

► **Theorem 1.** *For the FullKnowledge/Away model, Algorithm 1 has an optimal competitive ratio of*

$$1 + \frac{2}{1-v}. \quad (1)$$

Proof. By Algorithm 1, the robot goes in one direction for a time $\frac{d}{1-v}$. Observe that if the robot does not encounter the target after this amount of time, it must be on the opposite side of the origin (in the other direction). At the time the robot changes direction, its distance to the target will be equal to $\frac{d}{1-v} + d + \frac{dv}{1-v} = \frac{2d}{1-v}$. Thus, the total time required until the robot catches up to the target is at most

$$\frac{d}{1-v} + \frac{2d}{(1-v)^2}.$$

Clearly then, the competitive ratio is at most

$$\frac{\frac{d}{1-v} + \frac{2d}{(1-v)^2}}{\frac{d}{1-v}} = 1 + \frac{2}{1-v}$$

which is as claimed in Equation (1) above.

Optimality follows from the fact that regardless of which direction the robot chooses to travel, the adversary can place the target in the opposite direction. Moreover, for the robot to catch up to the target it must visit one of the points $\pm \frac{d}{1-v}$. If the robot visits location $\frac{d}{1-v}$ to the right (resp. left) the adversary places the target on the left (resp. right). Therefore the completion time will be at least $\frac{d}{1-v} + \frac{2d}{(1-v)^2}$. This shows the upper bound is tight and completes the proof of Theorem 1. ◀

2.2 The FullKnowledge/Toward Model

Consider the following algorithm which is similar to Algorithm 1.

■ **Algorithm 2** Online Algorithm for the FullKnowledge/Toward Model.

-
- 1: **input:** target speed v and initial distance d
 - 2: choose any direction and go for time $\frac{d}{1+v}$
 - 3: **if** target not found **then**
 - 4: change direction and go until target is found
-

► **Theorem 2.** *For the FullKnowledge/Toward model, Algorithm 2 has competitive ratio at most*

$$1 + \frac{2}{1+v}. \quad (2)$$

Proof. The robot goes in one direction for a time $\frac{d}{1+v}$. If the robot finds the target in this time, the algorithm is clearly optimal. If, however, the robot does not find the target, then it must be on the opposite side of the origin (in the other direction). If this is the case, then by time $\frac{d}{1+v}$ the target has moved a distance $\frac{dv}{1+v}$ and is at distance $d - \frac{dv}{1+v} = \frac{d}{1+v}$ from the origin. Therefore at the time the robot changes direction, the distance between robot and target is $\frac{2d}{1+v}$. Thus, the robot will encounter the target in additional time $\frac{2d}{(1+v)^2}$. It follows that the total time required for the robot to meet the target is $\frac{d}{1+v} + \frac{2d}{(1+v)^2}$ and the resulting competitive ratio satisfies

$$CR \leq \frac{\frac{d}{1+v} + \frac{2d}{(1+v)^2}}{\frac{d}{v+1}} = 1 + \frac{2}{1+v}.$$

This completes the proof of Theorem 2. ◀

► **Theorem 3.** *For the FullKnowledge/Toward model, the competitive ratio of any online algorithm is at least $1 + \frac{2}{1+v}$, provided that $v < 1$. In particular, Algorithm 2 is optimal for $v < 1$.*

Proof. Consider any algorithm for a robot starting at the origin to meet a target initially placed at an unknown location distance d from the origin. For any point at distance a from the origin, the target takes exactly $\frac{d-a}{v}$ time to reach a . Then, let t denote the time the robot first passes through a point at distance a from the origin. If $t < \frac{d-a}{v}$, then the robot cannot know whether the target is on the same or opposite side of the origin. On the other hand, if $t \geq \frac{d-a}{v}$ and it has not encountered the target, then the target must be on the opposite side of the origin. Thus, given a trajectory, let $\pm a$ be the first point such that the robot is at position $\pm a$ at time exactly $\frac{d-a}{v}$. Clearly such a point must exist for any trajectory since the target is moving toward the origin. Then whichever side of the origin the robot is on, consider the instance where the target started on the opposite side. Clearly then, the robot takes an additional time at least $\frac{2a}{1+v}$ to reach the target. Thus, the competitive ratio is given by:

$$\frac{\frac{d-a}{v} + \frac{2a}{1+v}}{\frac{d}{1+v}} = \frac{(d-a)(1+v) + 2av}{vd} = 1 + \frac{1}{v} + \frac{a(v-1)}{dv}. \quad (3)$$

Observe that whenever $v < 1$, the right-hand side of Equation (3) satisfies

$$1 + \frac{1}{v} + \frac{a(v-1)}{dv} \geq 1 + \frac{1}{v} + \frac{\frac{d}{1+v}(v-1)}{dv} = 1 + \frac{2}{1+v}$$

which completes the proof of Theorem 3. ◀

12:6 Line Search for an Oblivious Moving Target

With Theorem 3 proved, we know Algorithm 2 is optimal for any value of v between 0 and 1, but what about when $v > 1$? In this case, we'll prove the following algorithm is optimal: the robot waits at the origin forever. We call this algorithm “the waiting algorithm”.

► **Theorem 4.** *Whenever the target is moving toward the origin with speed $v \geq 1$, the waiting algorithm has an optimal competitive ratio of $1 + \frac{1}{v}$.*

Proof. Clearly the algorithm takes exactly time d/v to complete and so the upper bound follows trivially. For the lower bound, we build upon the proof of Theorem 3. It simply remains to consider Equation (3) for $v > 1$. In this case, the right-hand side of Equation (3) is increasing with respect to $a \geq 0$, so

$$1 + \frac{1}{v} + \frac{a(v-1)}{dv} \geq 1 + \frac{1}{v}.$$

This completes the proof of Theorem 4. ◀

► **Remark 5.** Observe that the waiting algorithm makes no use of the target's speed or initial distance and therefore, as long as the target is moving toward the origin, applies directly to the other knowledge models.

3 The NoDistance Model

In this section we assume that the robot knows v but not d . Consider the following zig-zag algorithm with “expansion ratio” $a > 0$ (with the value of a to be determined).

■ **Algorithm 3** Online Algorithm for NoDistance/Away and NoDistance/Toward Models.

```
1: input: target speed  $v$  and expansion ratio  $a$ 
2:  $i \leftarrow 0$ 
3: while target not found do
4:   if at origin then
5:      $d \leftarrow (-a)^i$ 
6:      $i \leftarrow i + 1$ 
7:   else if at  $d$  then
8:      $d \leftarrow 0$ 
9:     move toward  $d$ 
```

3.1 The NoDistance/Away Model

The following result was shown by Alpern and Gal [2].

► **Theorem 6.** *For the NoDistance/Away model, Algorithm 3 with $a = 2\frac{1+v}{1-v}$ has an optimal competitive ratio of*

$$1 + 8\frac{1+v}{(1-v)^2}.$$

3.2 The NoDistance/Toward Model

Recall first the statement made in Remark 5, that the optimality of the waiting algorithm (which makes no use of any knowledge of d) holds for any d as long as $v \geq 1$. Thus, we need only consider scenarios where $0 \leq v < 1$. As we will see, however, when the target is moving

toward the origin, the waiting algorithm is optimal for far slower targets! In general, since the target is moving toward the origin, the robot need not search ever-increasing distances away from the origin (i.e. execute Algorithm 3 with an expansion ratio $a > 1$). We call any algorithm which involves the robot *never* traveling further than some finite distance from the origin (in one or both directions) a *contracting algorithm*. Note that Algorithm 3 for $0 < a \leq 1$ is a contracting algorithm and $a = 0$ is exactly the waiting algorithm. We'll start by showing that any contracting algorithm cannot have a better competitive ratio than the waiting algorithm:

► **Theorem 7.** *The competitive ratio of Algorithm 3 for any $0 \leq a \leq 1$ is $1 + \frac{1}{v}$.*

Proof. Let d' be the finite distance further than which the robot will never travel in at least one direction. Then consider the scenario where the target is initially a distance $d = c \cdot d' \gg d'$ from the origin in the same direction. Then the competitive ratio is at least

$$\sup_c \frac{\frac{cd' - d'}{v}}{\frac{cd'}{v+1}} = \sup_c \frac{c-1}{c} \frac{1+v}{v} = \lim_{c \rightarrow \infty} \frac{c-1}{c} \left(1 + \frac{1}{v}\right) = 1 + \frac{1}{v}$$

which proves Theorem 7. ◀

By Theorem 7, any algorithm which hopes to out-perform the waiting algorithm must be expanding. Now we show that the following hybrid algorithm, Algorithm 4, is optimal.

■ **Algorithm 4** Wait or Zig-Zag Search Algorithm for NoDistance/Toward model.

```

1: input: target speed  $v$ 
2: if  $v \geq \frac{1}{3}$  then
3:   execute waiting algorithm
4: else
5:   execute Algorithm 3 with  $a = 2\frac{1-v}{1+v}$ 

```

► **Theorem 8.** *For the NoDistance/Toward model, the competitive ratio of Algorithm 4 is at most*

$$\begin{cases} 1 + \frac{1}{v} & \text{if } v \geq \frac{1}{3} \\ 1 + 8\frac{1-v}{(1+v)^2} & \text{if } v < \frac{1}{3} \end{cases} \quad (4)$$

Proof. The first case is trivial: the competitive ratio of the waiting algorithm is exactly $1 + \frac{1}{v}$ by Theorem 4. The second case, however, is a bit more complicated. First, observe that if $v < \frac{1}{3}$ then a must be less than 3. Indeed, consider the scenario where the robot “just misses” the target on the very first round of the algorithm (after traveling a distance 1 in some direction and then turning around). Then the competitive ratio of the algorithm is

$$1 + \frac{2a+2}{1+v}$$

which is greater than $1 + 8\frac{1-v}{(1+v)^2}$ for any $a > 3$ and $v > 0$:

$$1 + \frac{2a+2}{1+v} > 1 + \frac{8}{1+v} > 1 + \frac{8}{1+v} \cdot \frac{1-v}{1+v}$$

since $\frac{1-v}{1+v} < 1$.

12:8 Line Search for an Oblivious Moving Target

Now, consider the round k when the robot catches up to the target and observe that

$$a^{k-2} < d - \left(2 \sum_{i=0}^{k-3} a^i + a^{k-2} \right) v = d - \left(2 \frac{a^{k-2} - 1}{a - 1} + a^{k-2} \right) v$$

since otherwise, the robot would have caught up to the target in round $k - 2$. This yields the following inequality which will prove useful in analyzing the competitive ratio below:

$$\begin{aligned} a^{k-2} &< d - \left(2 \frac{a^{k-2}}{a-1} + a^{k-2} \right) v \\ &\leq d \frac{a-1}{a-1+v(a+1)} + \frac{v}{a-1+v(a+1)} \\ &\leq d \frac{a-1}{a-1+v(a+1)} + \frac{1}{4a-2} \end{aligned} \quad (5)$$

Observe the worst competitive ratio, then, is given by the situation where the robot “just misses” the target on the $(k - 2)^{\text{th}}$ round and catches up to it only on round k . It follows the competitive ratio of Algorithm 4 is

$$\frac{2 \sum_{i=0}^{k-3} a^i + a^{k-2} + \frac{2(a^{k-2} + a^{k-1})}{1+v}}{\frac{d}{1-v}} \leq \frac{2 \frac{a^{k-2}-1}{a-1} + a^{k-2} + \frac{2(a^{k-2} + a^{k-1})}{1+v}}{\frac{d}{1-v}}$$

which, by Inequality (5) (and by substituting each a^{k-2} with the right-hand side of Inequality (5)), is less than or equal to

$$CR \leq 1 + \frac{1}{2} \left[\frac{1}{d} \left(\frac{a-3}{a-1} \cdot \frac{v(5-7a)}{1-3a+2a^2} \right) + \frac{4a^2}{(a-1)+v(a+1)} \right] \quad (6)$$

$$\begin{aligned} &\leq \lim_{d \rightarrow \infty} 1 + \frac{1}{2} \left[\frac{1}{d} \left(\frac{a-3}{a-1} \cdot \frac{v(5-7a)}{1-3a+2a^2} \right) + \frac{4a^2}{(a-1)+v(a+1)} \right] \\ &= 1 + \frac{2a^2}{(a-1)+v(a+1)} \end{aligned} \quad (7)$$

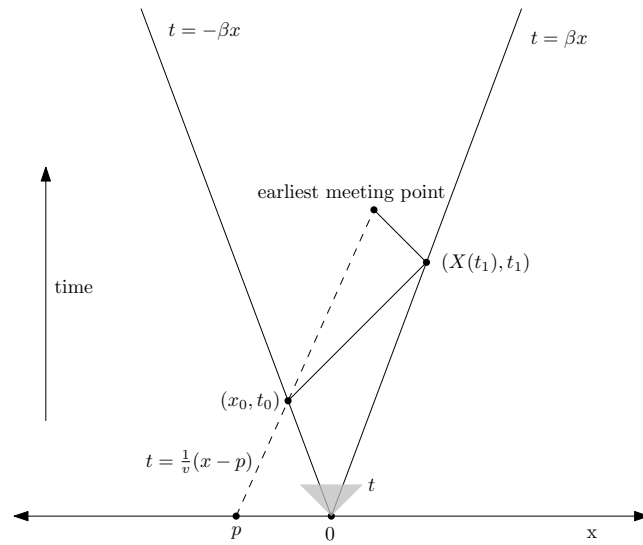
which follows since the right-hand side of Inequality (6) is increasing with respect to d on $1 < a \leq 3$. Finally, the right-hand side of Inequality (7) is minimized at $a = 2 \frac{v-1}{v+1}$ with a value of $1 + 8 \frac{1-v}{(1+v)^2}$, which proves Theorem 8. ◀

Now we show that Algorithm 4 is optimal by proving a tight lower bound on the competitive ratio for any online algorithm. Our proof is based on techniques developed in [11]. Let $X(t)$ denote be the robot’s position at time t according to a given strategy.

► **Theorem 9.** *For the NoDistance/Toward model, any strategy X has a competitive ratio of at least*

$$\begin{cases} 1 + \frac{1}{v} & \text{if } v \geq \frac{1}{3} \\ 1 + 8 \frac{1-v}{(1+v)^2} & \text{otherwise} \end{cases}$$

Proof. Let $\beta_t = \inf_{t' > t} \frac{t'}{|X(t')|}$. Clearly, then, if $t_1 \leq t_2$ then $\beta_{t_1} \leq \beta_{t_2}$. Furthermore, $\beta_t \geq 1$ for all t since the maximum speed of the robot is 1. Now let $\beta = \lim_{t \rightarrow \infty} \beta_t$. By definition of the limit infimum, there must exist a finite time t such that $\beta \leq \frac{t'}{|X(t')|}$ for all $t' \geq t$ and thus there must exist a time $t_1 > \frac{\beta(\beta+1)}{\beta-1}t$ such that the robot reaches a point (without loss of generality, on the right side of the origin) $X(t_1) = \frac{t_1}{\beta+\epsilon_1}$ for any arbitrarily small $\epsilon_1 > 0$.



■ **Figure 1** The cone-bounded trajectory of the robot and worst-case placement p of the target. The small gray triangle is to remind the reader that, by the definition of β , the robot trajectory is only guaranteed to be contained by the cone after some finite time t . Thus, in order to maximize the competitive ratio, we (as the adversary) should place the target so that its trajectory does not intersect (x_0, t_0) or the gray triangle.

Consider such a time and observe that, by construction, the robot could not have reached any point to the left of $x_0 = -\frac{t_1 - X(t_1)}{1 + \beta}$ after time $t_0 = \frac{\beta(t_1 - X(t_1))}{1 + \beta}$ since $x_0 \leq -t$ and $t_0 > t$ (see Figure 1). Now, consider a target starting at initial position p (to be determined) moving at speed $v > 0$ toward a robot which starts at the origin and has a speed of 1. Thus, by placing the target at a starting location so that the farthest *right* the robot could have reached is $x_0 - \epsilon_0$ for any arbitrarily small ϵ_0 , the robot *can not* have reached the target by time t_1 . Such a target has an initial position of

$$p = -\frac{(1 + \beta v)(t_1 - X(t_1))}{1 + \beta} - \epsilon_0$$

and follows the trajectory

$$X_{\text{target}}(t) = vt + p \tag{8}$$

where $X_{\text{target}}(t)$ denotes the robot's position at time t . Observe also, if the robot moves directly toward the target after t_1 , then its trajectory after time t_1 is given by

$$X(t) = X(t_1) + t_1 - t \tag{9}$$

Thus, the earliest time the robot could possibly encounter the target can be computed by finding the intersection between the robot trajectory (Equation (9)) and the target's trajectory (Equation (8)) and solving for t :

$$\begin{aligned} vt + p &= X(t_1) + t_1 - t \\ t &= \frac{X(t_1) + t_1 - p}{1 + v}. \end{aligned} \tag{10}$$

12:10 Line Search for an Oblivious Moving Target

Then the competitive ratio (Equation (10) divided by $-p/(1+v)$, the optimal search time) is

$$\begin{aligned} CR &\geq \sup_{\epsilon_0, \epsilon_1} \frac{(X(t_1) + t_1 - p)/(1+v)}{-p/(1+v)} = \sup_{\epsilon_0, \epsilon_1} \frac{X(t_1) + t_1 - p}{-p} = \sup_{\epsilon_0, \epsilon_1} \left[1 - \frac{X(t_1) + t_1}{p} \right] \\ &= \sup_{\epsilon_1} \left[1 + \frac{(1+\beta)(t_1 + X(t_1))}{(1+\beta v)(t_1 - X(t_1))} \right] \end{aligned} \quad (11)$$

$$= 1 + \frac{(1+\beta)^2}{(1+\beta v)(\beta-1)} \quad (12)$$

where Inequality (11) follows since $p = -\frac{(1+\beta v)(t_1 - X(t_1))}{1+\beta} - \epsilon_0$ for arbitrarily small $\epsilon_0 > 0$ and Inequality (12) follows since $X(t_1) = \frac{t_1}{\beta + \epsilon_1}$ for arbitrarily small $\epsilon_1 > 0$. Finally, observe that if $v < \frac{1}{3}$, then the right-hand side of Equality (12) has a single minimum of $1 + 8\frac{1-v}{(1+v)^2}$ at $\beta = \frac{v-3}{3v-1}$. On the other hand, if $v \geq \frac{1}{3}$, then the right-hand side of Equality (12) is decreasing with respect to β and thus the competitive ratio satisfies

$$CR \geq \lim_{\beta \rightarrow \infty} \left[1 + \frac{(1+\beta)^2}{(1+\beta v)(\beta-1)} \right] = 1 + \frac{1}{v}. \quad \blacktriangleleft$$

4 The NoSpeed Model

In this section we assume that the robot knows d but not v .

4.1 The NoSpeed/Away

For this model, it is clear that the robot cannot execute an algorithm like Algorithm 1 since no upper bound on the target's speed is known. Note that, if any upper bound $\hat{v} < 1$ on the target's speed were known, the robot could execute Algorithm 1 by assuming the target speed to be equal to \hat{v} , resulting in a competitive ratio of at most $1 + \frac{2}{1-\hat{v}}$. Since the target speed v is unknown (and potentially very close to 1), however, we propose another strategy. Consider a monotone increasing non-negative integer sequence $\{f_i : i \geq 0\}$ such that $f_0 = 1$ and $f_i < f_{i+1}$, for all $i \geq 0$. The idea of the algorithm is to search for the target by making a guess about its speed in rounds as follows. We start from the origin and alternate searching right and left. On the i -th round, we use the guess $v_i = 1 - 2^{-f_i}$ and search the necessary distance away from the origin such that, if the target's speed is less than or equal to v_i and the target's initial position is in the same direction from the origin that the robot moves in round i , then the target will be found in round i . Otherwise, we can conclude that either the target is moving with a speed greater than v_i or else it is on the opposite side of the origin. In this case the robot returns to the origin and repeats the algorithm in the opposite direction.

Later in the analysis we will show how to select the integer sequence $\{f_i : i \geq 0\}$ so as to obtain bounds on the competitive ratio. The algorithm explained above is formalized as Algorithm 5.

Algorithm 5 Online Algorithm for NoSpeed/Away Model.

- 1: **input:** target initial distance d
 - 2: integer sequence $\{f_i : i \geq 0\}$ such that $f_i < f_{i+1}$, for $i \geq 0$ and $f_0 = 1$;
 - 3: $t \leftarrow 0$
 - 4: **for** $i \leftarrow 0, 1, 2, \dots$ until target found **do**
 - 5: $v_i \leftarrow 1 - 2^{-f_i}$
 - 6: $x_i \leftarrow (-1)^i \cdot \frac{d+tv_i}{1-v_i}$
 - 7: move to x_i and back to the origin
 - 8: $t \leftarrow t + |x_i|$
-

To compensate for the fact that the starting speed of the robot in the algorithm is $v_0 = 1 - 2^{-1} = 1/2$ we first need to consider the case $v \leq \frac{1}{2}$.

► **Lemma 10.** *For the NoSpeed/Away model, if the unknown speed v of the target is less than or equal to $\frac{1}{2}$ then the competitive ratio of Algorithm 5 is at most 5.*

Proof. According to Algorithm 5 and since $v < 1/2$, the robot will find the target either on its first trip away from the origin, after time at most $2d$, or after the first time it changes direction of movement. In the worst case it will spend time $2d$ in one direction and then additional time $\frac{2d+d+2dv}{1-v}$. It follows that the competitive ratio is at most

$$\frac{2d + \frac{2d+d+2dv}{1-v}}{\frac{d}{1-v}} = 5$$

which proves Lemma 10. ◀

Next we analyze the competitive ratio of the algorithm for $v > \frac{1}{2}$.

► **Lemma 11.** *For the NoSpeed/Away model, if the unknown speed v of the target is greater than $\frac{1}{2}$ then the competitive ratio of Algorithm 5 is at most $1 + 2^{1+\sum_{j=0}^k f_j} \cdot 4^{k+1}$ where k is the first k such that $v_k \geq v$.*

Proof. Let d_i be the distance from the origin the target would be if its speed was equal to v_i , where $v_i = 1 - 2^{-f_i}$ at time $\sum_{j=0}^{i-1} (1 - 2^{-f_j})$. In other words, if $v_i \geq v$, then d_i is the maximum distance of the target from the origin (and thus, the robot) at the beginning of round i of the algorithm. Thus, if the speed of the target is less than or equal to v_i and the robot moves toward it in round i , then it would take at most $x_i = \frac{d_i}{1-v_i} = 2^{f_i} d_i$ additional time for the robot to catch up to the target, for $i \geq 0$. Recall the algorithm involves the robot moving a distance x_i (in time x_i , since the robot's speed is 1) away from the origin and back in round i . Observe then that $d_0 = d$, $v_0 = 1/2$, and

$$d_i = d + 2v_i \sum_{j=0}^{i-1} x_j. \tag{13}$$

Therefore, it follows from the definition of x_i that

$$x_i = 2^{f_i} \left(d + 2v_i \sum_{j=0}^{i-1} x_j \right). \tag{14}$$

12:12 Line Search for an Oblivious Moving Target

As a consequence

$$\sum_{j=0}^{i-1} x_j = \frac{x_i - 2^{f_i} d}{2^{f_i} \cdot 2 \cdot v_i} \quad (15)$$

Similarly, if we replace i with $i + 1$ we have that

$$\sum_{j=0}^i x_j = \frac{x_{i+1} - 2^{f_{i+1}} d}{2^{f_{i+1}} \cdot 2 \cdot v_{i+1}} \quad (16)$$

Subtracting Equation (15) from Equation (16), we derive the recurrence

$$x_i = \frac{x_{i+1} - 2^{f_{i+1}} d}{2^{f_{i+1}} \cdot 2 \cdot v_{i+1}} - \frac{x_i - 2^{f_i} d}{2^{f_i} \cdot 2 \cdot v_i} \quad (17)$$

Collecting similar terms and simplifying Equation (17) yields

$$\begin{aligned} \frac{x_{i+1}}{2^{f_{i+1}} \cdot 2 \cdot v_{i+1}} &= \left(1 + \frac{1}{2^{f_i} \cdot 2 \cdot v_i}\right) x_i + \left(\frac{2^{f_{i+1}}}{2^{f_{i+1}} \cdot 2 \cdot v_{i+1}} - \frac{2^{f_i}}{2^{f_i} \cdot 2 \cdot v_i}\right) d \\ &= x_i \left(1 + \frac{1}{2^{f_i} \cdot 2 \cdot v_i}\right) + \frac{d}{2v_{i+1}} - \frac{d}{2v_i} \end{aligned} \quad (18)$$

$$\leq \left(1 + \frac{1}{2^{f_i} \cdot 2 \cdot v_i}\right) x_i \quad (19)$$

following from the fact the sum of the last two terms in Inequality 18 is less than or equal to 0.

If we simplify the right-hand side of Equation (19), we derive the following recursive inequalities

$$\begin{aligned} x_{i+1} &\leq 2^{f_{i+1}} \cdot 2 \cdot v_{i+1} \left(1 + \frac{1}{2^{f_i} \cdot 2 \cdot v_i}\right) x_i \\ &\leq 2^{f_{i+1}} \cdot 2 \cdot \left(1 + \frac{1}{2^{f_i}}\right) x_i \\ &\leq (2 \cdot 2^{f_{i+1}} + 2 \cdot 2^{f_{i+1}-f_i}) x_i \\ &\leq 2^{f_{i+1}} \cdot 4 \cdot x_i, \end{aligned} \quad (20)$$

which follows since $\frac{1}{2} \leq v_i < 1$ and $f_i < f_{i+1}$ for all i .

By repeated application of the last Recurrence (20) above and using the fact that by definition $x_0 = 2^{f_0} d$, it follows easily by induction that

$$\begin{aligned} x_{i+1} &\leq 2^{f_{i+1}} \cdot 4 \cdot x_i \\ &\leq 2^{f_{i+1}+f_i} \cdot 4^2 \cdot x_{i-1} \\ &\vdots \\ &\leq 2^{f_{i+1}+f_i+f_{i-1}+\dots+f_1} \cdot 4^{i+1} \cdot x_0 \\ &\leq 2^{\sum_{j=0}^{i+1} f_j} \cdot 4^{i+1} \cdot d \end{aligned} \quad (21)$$

Consider the first i such that $v_i \geq v$. It follows that and $v_{i-1} < v$ which yields $1 - v < 1 - v_{i-1} = 2^{-f_{i-1}}$ and implies that $2^{f_{i-1}} < \frac{1}{1-v}$. Note that although $v_i \geq v$, the robot may not find the target in round i because it is located in the opposite direction. It is

guaranteed, however, to find the target by round $i + 1$. Moreover the total time that has elapsed from the start until round i is $2 \sum_{j=0}^i x_j$ at which time the target is at distance $d + v2 \sum_{j=0}^i x_j$ from the origin.

As a consequence the competitive ratio of Algorithm 5 is at most

$$\begin{aligned} \frac{2 \sum_{j=0}^i x_j + \frac{d+2v \sum_{j=0}^i x_j}{1-v}}{\frac{d}{1-v}} &= 1 + \frac{2(v+1-v)}{d} \sum_{j=0}^i x_j \\ &= 1 + \frac{x_{i+1} - 2^{f_{i+1}} d}{d} \cdot \frac{2}{2^{f_{i+1}} \cdot 2 \cdot v_{i+1}} \end{aligned} \quad (\text{By (16)})$$

$$\begin{aligned} &\leq 1 + \frac{x_{i+1}}{d} \cdot \frac{2}{2^{f_{i+1}} \cdot 2 \cdot v_{i+1}} \\ &\leq 1 + \frac{1}{v_{i+1}} 2^{\sum_{j=0}^i f_j} \cdot 4^{i+1} \end{aligned} \quad (\text{By (21)})$$

Since $v_{i+1} \geq 1/2$ we conclude with an upper bound on the competitive ratio of Algorithm 5 of

$$1 + 2^{1+\sum_{j=0}^i f_j} \cdot 4^{i+1} \quad (22)$$

which proves Lemma 11. ◀

We are now ready to prove the main theorem about the competitive ratio of Algorithm 5.

► **Theorem 12.** *For the NoSpeed/Away model, the competitive ratio of Algorithm 5 when applied to the sequence $f_j = 2^j$, for all $j \geq 0$, is at most*

$$\begin{cases} 5 & \text{if } v \leq \frac{1}{2} \\ 1 + \frac{16(\log \frac{1}{1-v})^2}{(1-v)^4} & \text{otherwise} \end{cases}$$

where \log is the base-2 logarithm.

Proof. Consider the first index i such that $v_i \geq v$. It follows that $v_{i-1} < v$, and so

$$1 - 2^{-2^{i-1}} < v \Rightarrow 2^{2^{i-1}} < \frac{1}{1-v}.$$

Then, by Lemma 11, the competitive ratio of is at most

$$\begin{aligned} 1 + 2^{1+\sum_{j=0}^i f_j} \cdot 4^{i+1} &= 1 + 2^{1+\sum_{j=0}^i 2^j} \cdot 4^{i+1} \\ &= 1 + 2^{2^{i+1}} \cdot 4^{i+1} \leq 1 + \left(\frac{1}{1-v}\right)^4 \cdot \left(4 \cdot \log\left(\frac{1}{1-v}\right)\right)^2 \\ &\leq 1 + \frac{16 \left(\log \frac{1}{1-v}\right)^2}{(1-v)^4} \end{aligned}$$

This proves Theorem 12. ◀

► **Remark 13.** By Theorem 1, $1 + \frac{2}{1-v}$ is a lower bound on any algorithm when both v, d are known. As a consequence it must also be a lower bound when d is known but v is not.

4.2 The NoSpeed/Toward Model

Now we consider the case where the target is moving toward the origin.

■ **Algorithm 6** Online Algorithm for NoSpeed/Toward Model.

-
- 1: **input:** target initial distance d
 - 2: choose any direction and go for time d
 - 3: **if** target not found **then**
 - 4: change direction and go until target is found
-

► **Theorem 14.** *For the NoSpeed/Toward model, Algorithm 6 achieves an optimal competitive ratio at most 3.*

Proof. The robot chooses a direction (without loss of generality, say to the right) and goes for a time d (this is where the robot makes use of its knowledge of the distance d). If it does not find the target it changes direction. In the meantime the target has moved for a distance dv and now must be at location $-d + dv$. Therefore at the time the robot changes direction the distance between robot and target is equal to $d - (-d + dv) = 2d - dv$, and hence the meeting will take place in additional time $\frac{2d-dv}{1+v}$. It follows that the total time required for the robot to meet the target must be equal to $d + \frac{2d-dv}{1+v}$. The resulting competitive ratio satisfies

$$CR \leq \frac{d + \frac{2d-dv}{1+v}}{\frac{d}{v+1}} = 3.$$

This proves the upper bound.

To prove the lower bound we argue as follows. If the searcher never visits either of the points $\pm d$ then the competitive ratio is arbitrarily large for very small values of v . Let $\epsilon > 0$ be sufficiently small and let the speed of the target be $v = \epsilon/3$. Consider the first time, say t , that the robot reaches one of the points $\pm(d - \epsilon)$. Without loss of generality let this point be $d - \epsilon$ and suppose the target is adversarially placed at $-d$. Then at time t it will be located at $-d + tv$. Therefore the distance between the robot and the target at time t will be $d - \epsilon - (-d + tv) = 2d - tv - \epsilon$. The time it takes for robot to find the target, then, is at least $d - \epsilon + \frac{2d-tv-\epsilon}{1+v}$ and the competitive ratio is at least

$$\frac{d - \epsilon + \frac{2d-tv-\epsilon}{1+v}}{\frac{d}{1+v}} \geq 3 - \frac{2\epsilon + (t + \epsilon)v}{d}$$

It follows easily that if $t \geq 3d - \epsilon$ then $CR \geq \frac{t}{d/(1+v)} \geq 3 - 3\epsilon$. However, if $t \leq 3d - \epsilon$ then $\frac{2\epsilon + (t + \epsilon)v}{d} \leq 2\epsilon + 3v \leq 3\epsilon$, since by assumption $v = \epsilon/3$. Therefore again $CR \geq 3 - 3\epsilon$. This completes the proof of Theorem 14. ◀

5 The NoKnowledge Model

In this section we assume that neither the initial distance d nor the speed v of the target is known to the robot.

5.1 The NoKnowledge/Away Model

We now describe an approximation strategy resembling that described in Section 4.1. For this strategy though, the robot will need to guess both the target's speed *and* its initial distance.

Consider the situation where neither the distance d to the target nor its speed $v < 1$ is known to the robot. Also consider two monotone increasing non-negative integer sequences $\{f_i, g_i : i \geq 0\}$ such that $f_0 = 1$ and $g_0 = 0$ and $f_i < f_{i+1}$ and $g_i < g_{i+1}$, for all $i \geq 0$. The idea of the algorithm is to search for the target by making a guess for its speed and starting distance in rounds as follows. The robot, starting from the origin, alternates searching to the right and left. On the i -th round, it guesses that the target's speed does not exceed $v_i = 1 - 2^{-f_i}$ and that its initial distance from the origin does not exceed 2^{g_i} . Using these guesses, the robot searches exactly the distance required (which we will later denote d_i) to catch the target, given its guesses are correct *and* that the target is in the direction the robot searches in round i . If robot does not find the target after searching this distance, it returns to the origin and begins the next round. Later in the analysis we will show how to select the integer sequences $\{f_i, g_i : i \geq 0\}$ so as to obtain bounds on the competitive ratio. We formalize the algorithm described above as Algorithm 7.

■ **Algorithm 7** Online Algorithm for NoKnowledge/Away Model.

```

1: Inputs; Integer sequences  $\{f_i, g_i : i \geq 0\}$  such that  $f_i < f_{i+1}$  and  $g_i < g_{i+1}$ , for  $i \geq 0$ 
   and  $f_0 = 1$  and  $g_0 = 0$ ;
2:  $t \leftarrow 0$ 
3: for  $i \leftarrow 0, 1, 2, \dots$  until target found do
4:    $d_i \leftarrow 2^{g_i}$ 
5:    $v_i \leftarrow 1 - 2^{-f_i}$ 
6:    $x_i \leftarrow (-1)^i \cdot \frac{d_i + tv_i}{1 - v_i}$ 
7:   move to  $x_i$  and back to the origin
8:    $t \leftarrow t + |x_i|$ 

```

Since there is always an integer $i \geq 1$ such that both $v_i = 1 - 2^{-f_i} \geq v$ and $2^{g_i} \geq d$, it is clear that the robot will eventually succeed in catching the target. Next we analyze the competitive ratio of the algorithm.

► **Lemma 15.** *For the NoKnowledge/Away model, if Algorithm 7 terminates successfully in round $i + 1$ then its competitive ratio must satisfy*

$$CR \leq 1 + \frac{2(i+2)}{d} \cdot 2^{g_{i+1}} \cdot 2^{\sum_{j=0}^i f_j} \cdot 4^{i+1}. \quad (23)$$

Proof. We call each iteration of the loop in Algorithm 7 a *round*. For any round i , let d_i be the distance from the origin to where the target would be if its speed was equal to $v_i = 1 - 2^{-f_i}$ and its starting position 2^{g_i} . Recall that during the first $i - 1$ unsuccessful rounds, the target is moving further and further away from the origin. If the robot is at the origin and the speed of the target is v_i then it takes time at most $x_i = \frac{d_i}{1 - v_i} = 2^{f_i} d_i$ for the robot to catch up to the target, for $i \geq 0$. Observe from the algorithm that $d_0 = 1$ and $v_0 = 1/2$ and

$$d_i = 2^{g_i} + 2v_i \sum_{j=0}^{i-1} x_j. \quad (24)$$

Therefore, it follows from the definition of x_i that

$$x_i = 2^{f_i} \left(2^{g_i} + 2v_i \sum_{j=0}^{i-1} x_j \right). \quad (25)$$

12:16 Line Search for an Oblivious Moving Target

As a consequence

$$\sum_{j=0}^{i-1} x_j = \frac{x_i - 2^{f_i+g_i}}{2^{f_i} \cdot 2 \cdot v_i} \quad (26)$$

Similarly, if we replace i with $i + 1$ we have that

$$\sum_{j=0}^i x_j = \frac{x_{i+1} - 2^{f_{i+1}+g_{i+1}}}{2^{f_{i+1}} \cdot 2 \cdot v_{i+1}}. \quad (27)$$

Subtracting Equation (26) from Equation (27) we derive the recurrence

$$x_i = \frac{x_{i+1} - 2^{f_{i+1}+g_{i+1}}}{2^{f_{i+1}} \cdot 2 \cdot v_{i+1}} - \frac{x_i - 2^{f_i+g_i}}{2^{f_i} \cdot 2 \cdot v_i} \quad (28)$$

Collecting similar terms and simplifying Equation (28) yields

$$\begin{aligned} \frac{x_{i+1}}{2^{f_{i+1}} \cdot 2 \cdot v_{i+1}} &= \left(1 + \frac{1}{2^{f_i} \cdot 2 \cdot v_i}\right) x_i + \left(\frac{2^{f_{i+1}+g_{i+1}}}{2^{f_{i+1}} \cdot 2 \cdot v_{i+1}} - \frac{2^{f_i+g_i}}{2^{f_i} \cdot 2 \cdot v_i}\right) \\ &= \left(1 + \frac{1}{2^{f_i} \cdot 2 \cdot v_i}\right) x_i + 2^{g_{i+1}-1} \left(\frac{1}{v_{i+1}} - \frac{2^{g_i}}{2^{g_{i+1}} \cdot v_i}\right) \\ &\leq \left(1 + \frac{1}{2^{f_i} \cdot 2 \cdot v_i}\right) x_i + 2^{g_{i+1}-1} \end{aligned} \quad (29)$$

where Inequality (29) follows since $\frac{1}{v_{i+1}} - \frac{2^{g_i}}{2^{g_{i+1}} \cdot v_i} \leq 1$.

If we multiply out with the denominator in the lefthand side of Inequality (29) and simplify the righthand side we derive the following recursive inequalities

$$\begin{aligned} x_{i+1} &\leq 2^{f_{i+1}} \cdot 2 \cdot v_{i+1} \left(1 + \frac{1}{2^{f_i} \cdot 2 \cdot v_i}\right) x_i + 2^{f_{i+1}+g_{i+1}} v_{i+1} \\ &\leq \left(2(2^{f_{i+1}} - 1) + 2^{f_{i+1}-f_i} \cdot \frac{v_{i+1}}{v_i}\right) x_i + 2^{f_{i+1}+g_{i+1}} \end{aligned} \quad (30)$$

$$\leq 2^{f_{i+1}} \cdot 4 \cdot x_i + 2^{f_{i+1}+g_{i+1}}, \quad (31)$$

where in the derivation of Inequality (31) from the previous Inequality (30) we used the fact that $\frac{v_{i+1}}{v_i} \leq 2$.

By repeated application of the last Recurrence (31) above and using the fact that by definition $x_0 = 2^{f_0}d$, it follows easily by induction that

$$\begin{aligned} x_{i+1} &\leq 2^{f_{i+1}} \cdot 4 \cdot x_i + 2^{f_{i+1}+g_{i+1}} \\ &\leq 2^{f_{i+1}+f_i} \cdot 4^2 \cdot x_{i-1} + 2^{f_{i+1}+f_i+g_i} \cdot 4^1 + 2^{f_{i+1}+g_{i+1}} \\ &\vdots \\ &\leq 2^{g_0+\sum_{j=0}^{i+1} f_j} \cdot 4^{i+1} \cdot x_0 + 2^{g_1+\sum_{j=1}^{i+1} f_j} \cdot 4^i + 2^{g_2+\sum_{j=2}^{i+1} f_j} \cdot 4^{i-1} \\ &\quad + \dots + 2^{f_{i+1}+g_{i+1}} \\ &= \sum_{k=0}^{i+1} 2^{g_k+\sum_{j=k}^{i+1} f_j} \cdot 4^{i-k+1} \end{aligned} \quad (32)$$

since $x_0 = 1$.

The total time that has elapsed from the start until the beginning of last round i (when the robot visits the origin for the last time before catching the target) will be $\sum_{j=0}^i 2x_j$ at which time the target is at distance $d + v \sum_{j=0}^i 2x_j$ from the origin. As a consequence the competitive ratio of Algorithm 7 must satisfy the inequality

$$CR \leq \frac{2 \sum_{j=0}^i x_j + \frac{d+2v \sum_{j=0}^i x_j}{1-v}}{\frac{d}{1-v}}. \quad (33)$$

Simplifying the righthand side of Inequality (33) and using Identity (26) yields

$$\begin{aligned} CR &\leq 1 + \frac{2}{d} \sum_{j=0}^i x_j \\ &\leq 1 + \frac{x_{i+1}}{d} \cdot \frac{1}{2^{f_{i+1}} \cdot v_{i+1}} \quad (\text{Use Equation (26)}) \\ &\leq 1 + \frac{1}{v_{i+1} d 2^{f_{i+1}}} \sum_{k=0}^{i+1} 2^{g_k + \sum_{j=k}^{i+1} f_j} \cdot 4^{i-k+1} \quad (\text{Use Equation (32)}) \\ &\leq 1 + \frac{1}{v_{i+1} d} \sum_{k=0}^{i+1} 2^{g_k + \sum_{j=k}^i f_j} \cdot 4^{i-k+1}. \end{aligned}$$

Since $v_{i+1} \geq 1/2$ we conclude with

$$\begin{aligned} CR &\leq 1 + \frac{2}{d} \sum_{k=0}^{i+1} 2^{g_k + \sum_{j=k}^i f_j} \cdot 4^{i-k+1} \\ &\leq 1 + \frac{2}{d} \sum_{k=0}^{i+1} 2^{g_k + \sum_{j=k}^i f_j} \cdot 4^{i-k+1} \\ &\leq 1 + \frac{2(i+2)}{d} \cdot 2^{g_{i+1}} \cdot 2^{\sum_{j=0}^i f_j} \cdot 4^{i+1} \end{aligned} \quad (34)$$

This completes the proof of Lemma 15. ◀

We now prove the following theorem.

► **Theorem 16.** *For the NoKnowledge/Away model, Algorithm 7 with the sequences $g_i = f_i = 2^i$ has a competitive ratio of at most*

$$1 + \frac{16}{d} \left[\log \log \max \left(d, \frac{1}{1-v} \right) + 3 \right] \cdot \max \left(d, \frac{1}{1-v} \right)^8 \cdot \log^2 \max \left(d, \frac{1}{1-v} \right)$$

where \log is the base-2 logarithm.

Proof. Observe that if the robot finds the target in round $i+1$, then by design, one or both of the robot's round $i-1$ guesses for the target's speed $(1 - 2^{-2^{i-1}})$ or initial distance $(2^{2^{i-1}})$ must have been too low, otherwise the robot would have found the target in an earlier round. In other words, either $1 - 2^{-2^{i-1}} < v$ or $2^{2^{i-1}} < d$. It follows, then that $i-1 < \log \log \max \left(d, \frac{1}{1-v} \right)$. Then by Lemma 15, an upper bound on the competitive ratio is given by

12:18 Line Search for an Oblivious Moving Target

$$\begin{aligned}
CR &\leq 1 + \frac{2(i+2)}{d} \cdot 2^{g_{i+1}} \cdot 2^{\sum_{j=0}^i f_j} \cdot 4^{i+1} \\
&= 1 + \frac{2(i+2)}{d} \cdot 2^{2^{i+1}} \cdot 2^{2^{i+1}-1} \cdot 4^{i+1} \\
&= 1 + \frac{(i-1)+3}{d} \cdot (2^{2^{i-1}})^8 \cdot 16(2^{i-1})^2 \\
&= 1 + \frac{16}{d} \left[\log \log \max \left(d, \frac{1}{1-v} \right) + 3 \right] \cdot \max \left(d, \frac{1}{1-v} \right)^8 \cdot \log^2 \max \left(d, \frac{1}{1-v} \right)
\end{aligned}$$

which proves Theorem 16. ◀

► **Remark 17.** Observe that a lower bound of $1 + 8 \frac{1+v}{(1-v)^2}$ follows directly from the NoDistance/Away model.

5.2 The NoKnowledge/Toward Model

We can prove the following theorem.

► **Theorem 18.** *The optimal competitive ratio is $1 + \frac{1}{v}$ and is given by the waiting Algorithm.*

Proof. The upper bound follows directly from Theorem 4. For the lower bound, consider an algorithm where the robot does not wait forever and instead moves a distance $d' > 0$ to the right (without loss of generality – a symmetric argument for the case where the robot moves to the left follows trivially) after waiting at the origin for time $t \geq 0$. Then consider the scenario where the target with speed $v = \frac{d}{t+d'}$ is initially at $-d$ for any distance $d \geq 1$. Thus, the target reaches the origin at exactly the time the robot reaches d' and so their earliest possible meeting time is

$$t + d' + \frac{d'}{1+v} = \frac{d}{v} + \frac{d'}{1+v} \geq \frac{d}{v}$$

Thus, the competitive ratio is at least

$$\frac{d/v}{d/(1+v)} = 1 + \frac{1}{v}$$

This proves Theorem 18. ◀

6 Conclusion

We considered linear search for an autonomous robot searching for an oblivious moving target on an infinite line. Two scenarios were analyzed depending on whether the target is moving towards or away from the origin (and this is known to the robot). In either of these two scenarios we considered the knowledge the robot has about the speed and starting distance of the target. For each scenario we gave search algorithms and analyzed their competitive ratio for the four possible cases arising. Our bounds are tight in all cases when the target is moving towards the origin. They are also shown to be tight when the target is moving away from the origin and its speed $0 < v < 1$ is known to the robot; for this scenario we also obtain upper bounds when v is not known to the robot. It remains an open problem to prove tight bounds for the case when v is unknown to the robot and the target is moving away from the origin. It also remains open to find tight bounds for the case where the direction of movement of the target is unknown.

References

- 1 R. Ahlswede and I. Wegener. *Search problems*. Wiley-Interscience, 1987.
- 2 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.
- 3 Ricardo A. Baeza-Yates, Joseph C. Culberson, and Gregory J. E. Rawlins. Searching in the plane. *Inf. Comput.*, 106(2):234–252, 1993. doi:10.1006/inco.1993.1054.
- 4 A. Beck. On the linear search problem. *Israel Journal of Mathematics*, 2(4):221–228, 1964.
- 5 R. Bellman. An optimal search. *Siam Review*, 5(3):274–274, 1963.
- 6 A. Bonato and R. Nowakowski. *The game of cops and robbers on graphs*. American Mathematical Soc., 2011.
- 7 Jared Coleman, Evangelos Kranakis, Danny Krizanc, and Oscar Morales-Ponce. Line search for an oblivious moving target, 2022. doi:10.48550/arXiv.2211.03686.
- 8 Jurek Czyzowicz, Konstantinos Georgiou, Evangelos Kranakis, Danny Krizanc, Lata Narayanan, Jaroslav Opatrny, and Sunil M. Shende. Search on a line by byzantine robots. *Int. J. Found. Comput. Sci.*, 32(4):369–387, 2021. doi:10.1142/S0129054121500209.
- 9 Jurek Czyzowicz, Ryan Killick, and Evangelos Kranakis. Linear rendezvous with asymmetric clocks. In Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira, editors, *22nd International Conference on Principles of Distributed Systems, OPODIS 2018, December 17-19, 2018, Hong Kong, China*, volume 125 of *LIPICs*, pages 25:1–25:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICs.OPODIS.2018.25.
- 10 Jurek Czyzowicz, Evangelos Kranakis, Danny Krizanc, Lata Narayanan, and Jaroslav Opatrny. Search on a line with faulty robots. *Distributed Comput.*, 32(6):493–504, 2019. doi:10.1007/s00446-017-0296-0.
- 11 Ryan Killick. Lower bound for linear search. Unpublished Manuscript, 2022.
- 12 Gianluca De Marco, Luisa Gargano, Evangelos Kranakis, Danny Krizanc, Andrzej Pelc, and Ugo Vaccaro. Asynchronous deterministic rendezvous in graphs. *Theor. Comput. Sci.*, 355(3):315–326, 2006. doi:10.1016/j.tcs.2005.12.016.
- 13 B. J. McCabe. Searching for a one-dimensional random walker. *J. Applied Probability*, pages 86–93, 1974.
- 14 Sven Schuierer. Lower bounds in on-line geometric searching. *Comput. Geom.*, 18(1):37–53, 2001. doi:10.1016/S0925-7721(00)00030-4.
- 15 L. Stone. *Theory of optimal search*. Academic Press New York, 1975.

Randomized Byzantine Gathering in Rings*

John Augustine ✉ 

Indian Institute of Technology Madras, India

Arnhav Datar ✉ 

Indian Institute of Technology Madras, India
Carnegie Mellon University, Pittsburgh, PA, USA

Nischith Shadagopan ✉ 

Indian Institute of Technology Madras, India

Abstract

We study the problem of gathering k anonymous mobile agents on a ring with n nodes. Importantly, f out of the k anonymous agents are Byzantine. The agents operate synchronously and in an autonomous fashion. In each round, each agent can communicate with other agents co-located with it by broadcasting a message. After receiving all the messages, each agent decides to either move to a neighbouring node or stay put. We begin with the k agents placed arbitrarily on the ring, and the task is to gather all the good agents in a single node. The task is made harder by the presence of Byzantine agents, which are controlled by a single Byzantine adversary. Byzantine agents can deviate arbitrarily from the protocol. The Byzantine adversary is computationally unbounded. Additionally, the Byzantine adversary is adaptive in the sense that it can capitalize on information gained over time (including the current round) to choreograph the actions of Byzantine agents. Specifically, the entire state of the system, which includes messages sent by all the agents and any random bits generated by the agents, is known to the Byzantine adversary before all the agents move. Thus the Byzantine adversary can compute the positioning of good agents across the ring and choreograph the movement of Byzantine agents accordingly. Moreover, we consider two settings: standard and *visual tracking* setting. With visual tracking, agents have the ability to track other agents that are moving along with them. In the standard setting, agents do not have such an ability.

In the standard setting we can achieve gathering in $\mathcal{O}(n \log n \log k)$ rounds with high probability¹ and can handle $\mathcal{O}\left(\frac{k}{\log k}\right)$ number of Byzantine agents. With visual tracking, we can achieve gathering faster in $\mathcal{O}(n \log n)$ rounds whp and can handle any constant fraction of the total number of agents being Byzantine.

2012 ACM Subject Classification Computing methodologies → Self-organization; Theory of computation → Self-organization

Keywords and phrases Mobile agents and robots

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2022.13

Funding *John Augustine*: Supported by the Cybersecurity Research Centre under the IIT Madras Institute of Eminence scheme.

Nischith Shadagopan: Supported by the Young Research Fellowship 2020, IIT Madras.

1 Introduction

Swarm robotics envisage swarms of mobile robots or agents as they are sometimes called, self-organizing and collaborating to achieve shared goals. These smaller agents have various advantages such as robustness, scalability, and efficiency [16, 20]. Significant research has been

* Author names are listed alphabetically. All authors contributed equally to this work.

¹ Throughout this paper, “with high probability” or “whp” in short means with probability at least $1 - 1/n^c$ for some constant $c \geq 1$



conducted in the areas of gathering [1, 21], flocking [23], scattering [30], and exploring [11, 25]. Some research has also been conducted considering that the agents are deployed in adverse environments [12].

We focus on the problem of gathering in a ring against Byzantine agents. Although a ring is a simple structure, the nodes are all symmetric. It is therefore no surprise that fundamental symmetry-breaking problems like leader election [19] were first studied in the context of rings with insights then influencing the development of more general algorithms. It is quite easy to note that $\Omega(n)$ rounds are required for gathering any number of agents in a ring. Our goal is to design algorithms with running times that are close to linear in n .

Initially, k anonymous agents are positioned arbitrarily across the ring. Notably, f out of the k agents are Byzantine. The task is to gather all the good agents in a single node of the ring. The agents cannot distinguish between the nodes of the graph. Each agent can communicate with other agents in the same node through a local broadcast. The agents operate synchronously and can either stay or move to a neighbouring node in each round. A single adversary controls the Byzantine agents. The adversary has complete knowledge of the graph and the states of all the agents. The adversary can deviate arbitrarily from the protocol to prevent gathering. The adversary is also adaptive in the sense that it can make decisions based on continual learning. Further, we consider the visual tracking setting where agents have the ability to track other agents moving along with them.

1.1 Related works

Gathering of mobile agents with unique IDs in the presence of Byzantine agents was first considered by Dieudonné et al. [13]. They focus on finding the minimum number of good agents required to gather successfully. They prove that at least $k = 2f + 1$ agents are required to achieve gathering and also show that it is not possible to deterministically gather $k = 2f$ agents in a ring of known size. They also provide an algorithm to gather $k \geq 3f + 1$ agents in a network of known size. Bouchard et al. [6] provide a deterministic algorithm for gathering $k = 2f + 1$ agents, which is the minimum number of agents required to gather when the size of the network is known. They also prove that $k = 2f + 2$ agents are required to gather when the size of the network is unknown, a slight increase from before. Regrettably, both the aforementioned algorithms from [13, 6] have exponential running times and are hence not practical. Thereafter, Bouchard et al. [7] improved the running time by providing a polynomial-time algorithm that uses a piece of global knowledge of size $\mathcal{O}(\log \log \log n)$ and achieves gathering when $k \geq 5f^2 + 6f + 2$. Hirose et al. [18] provided polynomial-time algorithms with a similar number of Byzantine agents but consider the case when the Byzantine agents cannot change their ID. Their algorithms run in $\mathcal{O}((f + \Lambda_{all}) \cdot X(n))$ time, where Λ_{all} represents the length of the maximum ID of all agents and $X(n)$ is the number of rounds to explore a network of size n .

Sudo et al. [31] introduce a new communication model where each node contains a *whiteboard* where agents can leave information. Gathering in this model is trivial. Each agent can leave its ID in its starting node, and the agents can move to the node containing the smallest ID to achieve gathering in $\mathcal{O}(m)$ time, where m is the number of edges. Tsuchida et al. [33] extend this by allowing each node to have an *authenticated whiteboard*. Authenticated whiteboards allow agents to store information along with their signatures. Their algorithm deterministically achieves gathering in $\mathcal{O}(fm)$ time. But authenticated whiteboards is quite an advanced feature when considering mobile agents with basic functionality.

The (non-Byzantine) gathering problem has been researched extensively in rings [9, 22, 10, 24]. Klasing et al. [24] provided deterministic algorithms for rings in an asynchronous model by only using global weak-multiplicity detection. Izumi et al. [22] provided an optimal

$\mathcal{O}(n)$ time algorithm for rings for some configurations when the agents only had local weak-multiplicity detection. Subsequently, D'Angelo et al. [9, 10] designed gathering algorithms for rings using global or local-weak multiplicity detection, provided the agents can take a snapshot of the ring at all occupied nodes.

Research on randomized algorithms for gathering and other search problems has been ongoing for a few years now and is summarised by Alpern and Gal [3]. Alpern et al. [2] provided an algorithm for rendezvous of two agents in a ring that runs in expected $\mathcal{O}(n)$ rounds. Ooshita et al. [28] consider the problem of gathering agents in an anonymous unidirectional ring under the constraint the agents are unaware of the number of agents and the number of nodes. Furthermore, their communication model employs whiteboards. They prove that there cannot exist randomized algorithms for this problem with termination detection. In our model, the agents can move in both directions and are aware of the number of nodes. Therefore their results do not apply in our context. A relevant result in our context is given by Cooper et al. [8] who introduce coalescing random walks. Here, particles perform independent random walks on the graph. Whenever two particles meet at a node, they combine and continue the random walk. The paper explores the time required to combine all the particles in the graph with such an algorithm. This algorithm can be trivially adapted to our model. The combining of particles is enabled in our context by co-located agents generating common random bits. This allows co-located agents to continue the random walk together. Such an algorithm takes $\tilde{\mathcal{O}}(n^2)$ time in expectation to gather all the agents in a ring, which is quite slow. Notably, this algorithm can handle any number of Byzantine agents. Eguchi et al. [14] provide some results on fast randomized gathering but consider the case when there are only two agents which are placed on adjacent nodes initially.

With this paper, we aim to kick off research on fast randomized algorithms with anonymous agents despite the presence of Byzantine agents. Both our results in this paper are log factors away from being linear in n . Our algorithms rely on the vital ability of agents to produce uniform and independent random bits. This integrated with synchronous operation of agents and local broadcast-based model of communication gives co-located agents the ability to generate common random bits. This can be done in the following way

1. Each agent generates a uniform and independently random string of s bits
2. Each agent broadcasts this string to other co-located agents, including itself
3. Each agent computes the XOR of all the strings received.

This procedure allows co-located agents to generate a common random string of arbitrary length as long as there is at least one good agent. We believe that this is a reasonable abstraction for the following two reasons. Firstly, this is an important abstraction which allows us to provide more scalable, faster and more secure (resilience to Byzantine agents) algorithms. Secondly, such an abstraction is easy to realise using existing technological capabilities. At a physical level, there are several results that allow agents to arrange themselves in the form of a circle [34, 17] or other patterns [32]. Once the agents arrange themselves in a fitting pattern, they can be forced to simultaneously broadcast their random bit through physical means like lights. Algorithmically also, there are several results which enable common coin tossing [5, 29, 26, 4, 15]. Regardless of the technology that is employed, we believe that common random bits will be beneficial for a range of robot coordination problems.

1.2 Our Model

We consider the problem of gathering k agents placed arbitrarily on a ring $G = (V, E)$. We denote the number of nodes in the graph G by $n = |V|$. The nodes of the graph do not have any facility to compute, store or communicate any information. The nodes of the graph can

13:4 Randomized Byzantine Gathering in Rings

be viewed as just a container for agents. Each node $v \in V$ in G has degree 2 and has two labelled ports corresponding to each incident edge. This port labelling is common to all agents. An edge $e = (u, v) \in E$ indicates that an agent can move bidirectionally between node u and node v in one step. The agents cannot distinguish between the nodes of the ring. The agents also do not have a common clockwise/counterclockwise orientation. An agent can fix a particular orientation and remember the number of steps taken to keep track of its position relative to some node.

The agents operate synchronously. Each agent has the ability to compute and store information. Each agent can also communicate with all other co-located agents using local broadcast. In the visual tracking setting, the agents also have the ability to identify other agents that are co-located for at least two consecutive rounds. The agents are aware of the number of nodes n . In the standard setting, the agents are also aware of an upper-bound on the number of Byzantine agents f . The agents have a fixed orientation of direction (clockwise/counterclockwise) but this orientation is not common across all the agents.

Computation is defined by a finite state machine coupled with the ability to generate random bits and communicate messages. At each round $r > 0$, each good agent:

1. broadcasts (to all other co-located agents) a message that is a function of its state and includes public random bits,
2. receives messages broadcasted by other agents in its current node,
3. transitions to a new state as a function of the current state, messages received, and random bits,
4. chooses to either stay in the current location or move through one of the ports (based on the chosen state),
5. and finally updates the state with the direction (clockwise/counterclockwise) in which it moved.

All agents must broadcast a message at the start of each round. Note that this is not a limitation because agents can send an empty \perp message if they have nothing to send.

f of the k anonymous agents are Byzantine, and all these agents are controlled by a single adversary. The remaining $k - f$ agents are called *good agents*. The adversary is computationally unbounded and can deviate arbitrarily from the protocol. The adversary can distinguish between the nodes of G and is also aware of the starting positions of all the good agents. Additionally, the adversary is strongly adaptive in the sense that it can capitalize on information gained over time (including the current round) to choreograph the actions of Byzantine agents. Specifically, the entire state of the system, which includes messages sent by all the agents and any random bits generated by the agents, is known to the adversary at the beginning of the subsequent step in the round. Thus the adversary can compute the positioning of good agents across the ring and choreograph the movement of Byzantine agents accordingly.

1.3 Our Techniques and Contributions

Being sparse and symmetric, rings present a significant challenge. We present algorithms for two settings:

- (a) the standard setting and
- (b) the setting where agents have visual tracking capability.

In the standard setting we can achieve gathering in $\mathcal{O}(n \log n \log k)$ rounds and can handle $\mathcal{O}\left(\frac{k}{\log k}\right)$ number of Byzantine agents. The algorithm splits the groups into leaders and followers. The leader groups just go around the ring and the follower group merges with the

■ **Table 1** Known results on randomized gathering protocols including those inferred from the literature on coalescing random walks. The \tilde{O} notation is used to hide $\text{polylog}(n)$ factors.

Setting	Time Complexity	Max. Byz. Agents
Coalescing random walks [8]	$\tilde{O}(n^2)$	$f < k$
With No Byzantine Agents	$\mathcal{O}(n)$ (on exp.)	$f = 0$
Standard setting	$\mathcal{O}(n \log n \log k)$	$f(35 + 34 \log_2(f + 1)) + 2 < k$
With visual tracking	$\mathcal{O}(n \log n)$	$f = \alpha k, 0 \leq \alpha < 1$
Lower bound	$\Omega(n)$	$f = 0$

first leader group that it meets. With visual tracking capability, i.e., the ability to identify which co-located agents were also co-located with them in the previous round, we can achieve gathering faster in $\mathcal{O}(n \log n)$ rounds as long as the number of Byzantine agents is some constant fraction of the total number of agents. The difference here is that the follower groups make note of all the leader groups that it meets and chooses one of them to merge with probability proportional to the leader group’s size. We can see that both these algorithms follow a common paradigm of splitting into leaders and followers. In the former algorithm, the follower groups merge with the first leader group that they meet but in the latter they take their time to evaluate all the leader groups and then decide to merge with one of them.

The summary of our results can be found in Table 1.

2 Algorithms

We present three algorithms. The first is a warm up that works only when all agents are good and requires an expected $\mathcal{O}(n)$ rounds for all agents to gather. The second algorithm is for the *standard gathering* of anonymous agents setting wherein agents are indistinguishable from round to round. This algorithm takes $\mathcal{O}(n \log n \log k)$ rounds and is resilient against up to a suitable $\mathcal{O}(k/\log k)$ Byzantine agents. Subsequently, we study the variant where agents can *visually track* others that are moving along with them and provide an algorithm that can handle an $f < k$ Byzantine failures. For simplicity we first present our algorithms assuming that all the agents have a common orientation of the ring and discuss briefly how to adapt them to the situation where there is no common orientation.

2.1 Warmup: No Byzantine Agents

We now present a simple warmup gathering algorithm inspired by Alpern et al. [2]. Their work is limited to gathering two agents in a ring whereas ours is more general. Our algorithm runs in expected $\mathcal{O}(1)$ stages. We use the term committee to refer to a set of agents; members of the committee know they are in the committee and those not in the committee know that they are not. A committee is good if it is neither empty nor the full set of agents, i.e., there must be at least one agent in the committee and one that isn’t. At each stage, the agents perform the following steps.

Step 1: Committee Election. The agents attempt a committee election that will result in a good committee members with probability at least a constant. We achieve this simply by allowing each agent to self-sample itself with probability $1/k$.

► **Lemma 1.** *Assuming $k \geq 2$, the elected committee will be good with probability at least a constant.*

13:6 Randomized Byzantine Gathering in Rings

Proof. The two bad events are either that the committee is empty (which can occur with probability $p_1 = (1 - 1/k)^k$) or that all agents are in the committee (which can occur with probability at most $p_2 = (1/k)^k$). Since $1 - p_1 - p_2$ is bounded from below by a constant, the lemma holds. ◀

The elected committee members furthermore choose a random ID – a bit string of a suitable length that is at least $\Omega(\log k)$. This will ensure that the IDs are unique with constant probability.

Step 2: Leader Election. The committee members go around the ring twice in a direction of their choosing. As they go around, they broadcast their ID in all nodes along the way. The agents on the other hand broadcast the smallest ID they have learned about thus far (with a special symbol say \perp dedicated for the case where they have not yet learned an ID). Clearly, once the committee members go around twice, all agents will know the smallest ID (i.e., the leader) assuming the committee is good and committee members have distinct IDs (both occurring with constant probability).

Step 3: Gathering. In this step, the leader stays put for n rounds (but keeps broadcasting its ID). All other agents go around the ring (in a direction of their choosing) and stop when they reach the leader.

Step 4: Verification. If all k agents gather, the procedure stops. Otherwise, we repeat from step 1 onward.

The following theorem follows immediately.

▶ **Theorem 2.** *As long as $k \geq 2$ and $n \geq 1$, all k agents will gather in expected $\mathcal{O}(n)$ rounds.*

We conclude this warmup with a few observations. Firstly, we observe that the algorithm does not require common coin tosses amongst agents within a node. Individual random bits generated by agents suffice. Secondly, we note that it is a Las Vegas algorithm that will terminate only when all agents have gathered (and are aware that gathering has terminated). However, in situations where we prefer that the algorithm runs for a predetermined number of rounds, we can run $\Theta(\log 1/\delta)$ stages and obtain the guarantee that the algorithm gathers all agents with probability at least $1 - \delta$. Finally, if $k \in \Omega(\log n)$ is sufficiently large, we can elect a good committee with high probability in one shot. Each agent can self-sample itself into the committee with probability $p = \Theta(\log n/k)$. Such a self-sampled committee can be shown to be good whp by applying a standard Chernoff bound (for example, Corollary 4.6 from [27]). Thus, the whole algorithm can complete in one stage whp.

2.2 Standard Gathering

The algorithm runs in $\mathcal{O}(\log k)$ stages with stages numbered from $j = 0$ to $\lceil \log_2(f + 1) \rceil - 1$. Each stage has $\lceil 2 \log_{8/7} n \rceil$ phases. At the starting of each stage, all groups below a threshold of 2^j become inactive for this stage. Inactive agents don't do anything for that stage. In each phase, each group decides to be either a leader or a follower (with probability $1/2$ each). The role of the leader group is quite simple, it just moves clockwise for $\lfloor \frac{n}{2} \rfloor$ rounds (out of which one round is set aside for randomizing the parity of its position). The follower group moves counterclockwise till it meets a leader group having number of agents greater than a threshold of 2^j for the first time (or up to $\lfloor \frac{n}{2} \rfloor$ rounds, whichever is sooner). After meeting such a leader group, the follower group starts moving in the clockwise direction and behaves exactly like a leader group till the phase ends. The last stage is similar to the earlier stages except all agents are active. Algorithm 1 presents the pseudocode.

Algorithm 1 Gathering in standard setting.

```

1: function LEADER
2:    $\text{random} \leftarrow 1$  with probability  $\frac{1}{2}$  ▷ This decision is made as a group
3:   Move one step clockwise if  $\text{random} = 1$  ▷ Randomize the parity of the group's
   position
4:   Move one step clockwise and broadcast “I am a leader” message for  $\lfloor \frac{n}{2} \rfloor - 1$  rounds
5: end function
6: function FOLLOWER(j)
7:    $\text{flag} \leftarrow 1$ 
8:   for  $\lfloor \frac{n}{2} \rfloor$  rounds do
9:     If  $\text{flag} = 1$ , go one step counterclockwise else go one step clockwise and broadcast
     “I am a leader” message
10:    Count the number  $d$  of leader agents in the node based on the number of “I am a
     leader” messages received
11:    if  $d \geq 2^j$  and  $\text{flag} = 1$  then ▷  $2^j$  is the threshold
12:       $\text{flag} \leftarrow 0$  ▷ At this point, the follower group has essentially coalesced with the
     leader group
13:    end if
14:  end for
15: end function
16: for  $j = 0$  to  $\lceil \log_2(f + 1) \rceil - 1$  stages do ▷ If  $f$  is not known, an upper bound can be used
17:   Each good agent forms a group with other agents co-located with it.
18:   if the number of agents in the group  $\geq 2^j$  then ▷ These are active agents
19:     for  $\lceil 2 \log_{8/7} n \rceil$  phases do
20:       Each good agent forms a group with other agents co-located with it.
21:       Decide to be a leader group or follower group with equal probability ▷ this
     decision is made as a group
22:       Leaders call LEADER(), followers call FOLLOWER(j)
23:     end for
24:   end if
25: end for
26: for  $\lceil 2 \log_{8/7} n \rceil$  phases do ▷ last stage
27:   Each good agent forms a group with other agents co-located with it.
28:   Decide to be a leader group or follower group with equal probability ▷ this decision is
     made as a group
29:   Leaders call LEADER(), followers call FOLLOWER( $\lceil \log_2(f + 1) \rceil$ )
30: end for

```

We define a *good group* as a group having at least one good agent. Define all the groups that have number of agents \geq the threshold at the start of a stage as *active groups*. Groups below the threshold are said to be *inactive*. We can classify active groups into two types : *true* and *fake*. True active groups have number of good agents greater than or equal to the threshold whereas fake active groups have number of good agents less than the threshold. An agent is said to be *active* if it is part of an active group and otherwise called an *inactive agent*. Two groups are said to be of the same *parity* if in that phase the number of edges between the nodes in which the two groups are present, is even. In this setup, it is important to note that a leader group meets a follower group only if they are of same parity. This is because if the number of edges between them is odd, then they will cross each other while

going across an edge and not realise this. Note that in this algorithm once two good agents become members of the same group, they will never separate. So the number of good groups never increases.

There is not much use for the Byzantine agents to behave as a follower group because the leader groups are quite simple and are not affected by follower groups. But the Byzantine agents can form a leader group and then present itself to a follower group. Then the follower group thinks that it has successfully merged with another group.

Suppose a set of Byzantine agents get together and form a leader group b_1 . Suppose they meet an active follower group g_1 . Group g_1 will think that it has merged with another group and will start behaving like a leader group for the rest of the stage. Group b_1 is now free to do whatever it wants. Group b_1 cannot catch up to any follower group by moving in the counterclockwise direction, it has to move in the clockwise direction to do so. But since g_1 starts moving clockwise from the next round after meeting b_1 , any other follower group will always meet g_1 no later than when it meets any agent in b_1 . Therefore any other follower group will merge with the active g_1 and the Byzantine adversary cannot do anything about this. Only thing agents in b_1 can do is form leader groups and merge with a group of parity different from g_1 as that group will never meet g_1 due to parity difference. Therefore a Byzantine agent can be part of two groups of Byzantine agents that merge with at most 2 active groups of different parity in any phase.

► **Lemma 3.** *In the first $\lceil \log_2(f+1) \rceil$ stages of the algorithm, the number of good agents that are not active at the beginning of the j^{th} stage is at most $34fj$, $0 \leq j \leq \lceil \log_2(f+1) \rceil$ whp. Also at the beginning of the j^{th} stage there is at least one true active group whp.*

Proof. Our proof is by the Principle of Mathematical Induction on j . When $j = 0$, the threshold for this stage is $2^j = 1$. All good groups have size ≥ 1 , therefore the number of good agents that are not active at the beginning of the 0^{th} stage is 0. Also since there is at least one good agent, there is at least one true active group, thereby establishing the base case.

Induction Hypothesis: The number of good agents that are not active at the beginning of the t^{th} stage is at most $34ft$ whp. Also at the beginning of the t^{th} stage there is at least one true active group whp.

Inductive Step: Let us now focus on the start of the $j = (t+1)$ th stage (or equivalently, the end of the t^{th} stage). We will now analyze what happens during the t^{th} stage. At the beginning of the t^{th} stage we know that the number of good agents that are not active is at most $34ft$ from induction hypothesis. Consider the groups that are active at the beginning of the t^{th} stage, we need to see how many of these good agents become inactive at the start of the $(t+1)^{\text{th}}$ stage, say x . Then we can bound the total number of good agents not active at the beginning of $(t+1)^{\text{th}}$ stage as $34ft + x$. This is because for the sake of upper bound we are assuming that agents that were inactive in the beginning of the t^{th} stage are inactive at the beginning of the $t+1^{\text{th}}$ stage and we only need to count the number of agents that were active at the beginning of the t^{th} stage but are inactive at the beginning of the $t+1^{\text{th}}$ stage (call it x).

By the Induction Hypothesis, there is at least one true active group at the beginning of t^{th} stage. Consider any such group g . Then for any other active group g_1 , if g is a leader and g_1 is a follower and the two groups are of same parity then these two groups will merge (ignoring Byzantine influence for now). Each of these are independent events with probability $\frac{1}{2}$. Therefore ignoring the Byzantine influence, there is a constant probability of at least $\frac{1}{8}$ with which a group merges in every phase.

The adversary needs to form a group with at least 2^t agents to cross the threshold. We know that such a group can merge with at most two good active groups. Therefore in one phase the Byzantine adversary can cause at most $\frac{f}{2^{t-1}}$ false merges with good active groups. We call it a false merge as in such a merge the number of active groups doesn't decrease. The other thing the adversary can do is combine with a few good agents to form a group with at least 2^t agents. But to do this, the good agents have to be active and therefore in such a merge the number of active groups decreases. Therefore it is not beneficial for the adversary to do this.

Let $G_{j,i}$ denote the number of active groups in the i^{th} phase of the j^{th} stage. For any group, let the probability of choosing a good leader group (also active) to merge with be P_i . We have seen that $P_i \geq \frac{1}{8}$. Let C_i be the number of merges between good groups and let B_i be the number of merges with Byzantine groups.

We have

$$G_{t,i+1} = G_{t,i} - C_i \tag{1}$$

$$\mathbb{E}[G_{t,i+1}|G_{t,i}] = G_{t,i} - \mathbb{E}[C_i|G_{t,i}] \tag{2}$$

$$\mathbb{E}[G_{t,i+1}|G_{t,i}] = G_{t,i} - \left(\frac{G_{t,i} - 1}{8} - B_i \right) \tag{3}$$

$$\mathbb{E}[G_{t,i+1}|G_{t,i}] \leq G_{t,i} - \left(\frac{G_{t,i} - 1}{8} - \frac{f}{2^{t-1}} \right) \tag{4}$$

$$\mathbb{E}[G_{t,i+1}] \leq \frac{7}{8} \mathbb{E}[G_{t,i}] + \left(\frac{1}{8} + \frac{f}{2^{t-1}} \right) \tag{5}$$

Solving this recurrence relation we get

$$\mathbb{E}[G_{t,i}] \leq \left(G_{t,0} - 1 - \frac{8f}{2^{t-1}} \right) \left(\frac{7}{8} \right)^i + \left(\frac{8f}{2^{t-1}} + 1 \right)$$

Since there are $2 \log_{8/7} n$ phases in each stage, setting i to $\log_{8/7} n^2 = i^*$

$$\mathbb{E}[G_{t,i^*}] \leq \frac{G_{t,0} - 1 - \frac{8f}{2^{t-1}}}{n^2} + \left(\frac{8f}{2^{t-1}} + 1 \right)$$

using Markov's inequality

$$P \left(G_{t,i^*} - \left(\frac{8f}{2^{t-1}} + 1 \right) \geq 1 \right) \leq \mathbb{E} \left[G_{t,i^*} - \left(\frac{8f}{2^{t-1}} + 1 \right) \right]$$

$$P \left(G_{t,i^*} - \left(\frac{8f}{2^{t-1}} + 1 \right) \geq 1 \right) \leq \mathbb{E}[G_{t,i^*}] - \left(\frac{8f}{2^{t-1}} + 1 \right)$$

$$P \left(G_{t,i^*} - \left(\frac{8f}{2^{t-1}} + 1 \right) \geq 1 \right) \leq \frac{G_{t,0} - 1 - \frac{8f}{2^{t-1}}}{n^2} \leq \frac{G_{t,0}}{n^2} \leq \frac{1}{n}$$

Therefore at the end of t^{th} stage there are $\frac{8f}{2^{t-1}} + 1$ active groups whp. Suppose all these active groups become inactive at the beginning of $t + 1^{th}$ stage, then $x \leq \left(\frac{8f}{2^{t-1}} + 1 \right) \cdot (2^{t+1} - 1)$ as if they are inactive at the beginning of $t + 1^{th}$ stage then the number of agents in that group is less than 2^{t+1} .

$$x \leq \left(\frac{8f}{2^{t-1}} + 1 \right) \times (2^{t+1} - 1) = 32f + 2^{t+1} - 1 - \frac{8f}{2^{t-1}}$$

13:10 Randomized Byzantine Gathering in Rings

since we are in the first part of the algorithm $t + 1 \leq \lceil \log_2(f + 1) \rceil$

$$x \leq 32f + 2^{t+1} - 1 - \frac{8f}{2^{t-1}} \leq 32f + 2f + 2 - 1 - 16$$

$$x \leq 34f$$

Therefore the number of good agents that are not active at the beginning of the $t + 1^{th}$ stage is at most $34ft + 34f = 34(t + 1)f$.

Let us count the number of good active agents at the beginning of t^{th} stage. The algorithm starts with $k - f$ good agents. At the beginning of the t^{th} stage there are at most $34ft$ good agents that are inactive. Therefore there are at least $k - f - 34ft$ good agents that are active at the beginning of the t^{th} stage. At the end of the t^{th} stage these good active agents are distributed among $8f/2^{t-1} + 1$ groups. Therefore by Pigeon Hole Principle, at least one group has $\frac{k-f-34ft}{\frac{8f}{2^{t-1}}+1} = 2^{t-1} \times \frac{k-f-34ft}{8f+2^{t-1}}$ good active agents. Since $t \leq \lceil \log_2(f + 1) \rceil - 1$ and $k - f > 34f(1 + \log_2(f + 1)) + 2$, we get that $\frac{k-f-34ft}{8f+2^{t-1}} \geq 4$. Therefore at the end of the t^{th} stage there is at least one group having 2^{t+1} good agents. Therefore at the beginning of the $(t + 1)^{th}$ stage there is at least one true active group, thereby completing the proof. ◀

Note here that since there are only $\mathcal{O}(\log k)$ stages, the whp assumption holds in induction.

Consider the $j^* = \lceil \log_2(f + 1) \rceil^{th}$ stage. The threshold in this stage is $2^{\lceil \log_2(f+1) \rceil} \geq f + 1$. From Lemma 3 there is at least one true active group at the beginning of this stage. Consider any such group g . Then for any other group g_1 , if g is a leader and g_1 is a follower and the two groups are of same parity then these two groups will merge. Each of these are independent events with probability $\frac{1}{2}$. Therefore there is a constant probability of at least $\frac{1}{8}$ with which a group merges in every phase.

In this stage, the threshold is greater than f , therefore the Byzantine agents cannot cause any false merges.

Let G_i denote the number of groups with at least one good agent in the i^{th} phase of this stage. Note that G_i denotes the number of good groups, not necessarily active. For any group, let the probability of choosing a good leader group to merge with be $P_i \geq \frac{1}{8}$. Let C_i denote the number of merges between good groups. Then $G_{i+1} = G_i - C_i$ and we have

$$\mathbb{E}[G_{i+1}|G_i] = G_i - \mathbb{E}[C_i|G_i] = G_i - \left(\frac{G_i - 1}{8}\right)$$

$$\mathbb{E}[G_{i+1}] \leq \frac{7}{8} \mathbb{E}[G_i] + \frac{1}{8}$$

$$\mathbb{E}[G_i] \leq (G_0 - 1) \left(\frac{7}{8}\right)^i + 1 \quad \text{(Solving the above recurrence relation)}$$

$$\mathbb{E}[G_{i^*}] \leq \frac{G_0 - 1}{n^2} + 1 \quad \text{(Setting } i \text{ to } \log_{8/7} n^2 = i^* \text{ which is the number of phases)}$$

Using Markov's inequality, we get $P(G_{i^*} - 1 \geq 1) \leq \frac{G_0 - 1}{n^2}$. Therefore at the end of Algorithm 1, there is one group whp. Thus,

► **Theorem 4.** *Given k agents placed arbitrarily on a graph G of n nodes, there exists a randomized gathering protocol that is resilient to a strongly-adaptive Byzantine adversary that can whp gather all good agents in $\mathcal{O}(n \log n \log k)$ rounds as long as k is greater than $f(35 + 34 \log_2(f + 1)) + 2$ where f is the number of Byzantine agents.*

Extension: No common orientation

Since there are only two possible orientations, by Pigeon Hole Principle, at least $\frac{k-f}{2}$ agents will have the same orientation. Therefore running Algorithm 1 and ensuring that only agents with same orientation interact, we can ensure gathering of at least $\frac{k-f}{2}$ agents. Due to the common port numbering of the graph, the agents can communicate their orientation and ensure they interact only with agents having same orientation as themselves. The restriction on the number of Byzantine agents will change slightly as essentially we now have half the number of good agents than we used to. The new constraint will be $\frac{k-f}{2} > 34f(1 + \log_2(f + 1)) + 2$. After gathering $\frac{k-f}{2} > f$ agents by running Algorithm 1, we can achieve gathering in another $\mathcal{O}(n \log n)$ phases by running the following algorithm:

- All groups choose one of the two orientations with probability $\frac{1}{2}$. The groups also choose whether to be leader or follower with equal probability.
- The follower groups go clockwise and make note of the largest leader group that they meet (ties are broken arbitrarily)
- The follower groups merge with the largest leader group that it met. This is possible as the location of leader groups is predictable

We know there is a group with $\frac{k-f}{2} > f$ number of good agents. So the largest group will be of size greater than f . Therefore in each phase there is a constant probability with which a group merges with the group with size at least $\frac{k-f}{2}$ and therefore gathering is achieved in $\mathcal{O}(n \log n)$ rounds whp.

2.3 With visual tracking

One significant drawback in the standard setting is that it limits $f \in \mathcal{O}(k / \log k)$, so a natural question we ask is whether we can relax the model in some reasonable manner to avoid this restriction on f . In this regard, we consider endowing agents with the ability to visually track other agents that are moving along with them. Consider two agents a and b that have been co-located from round $r - \delta$ to r (for some $\delta \geq 0$) but not in round $r - \delta - 1$. With visual tracking, we assume that a and b know at round r that they have been co-located for δ rounds, but apart from that, don't have any memory of prior encounters. This leverages the common ability of mobile agents to be able to visually see the other objects that are moving along with them.

The algorithm runs in $\lceil 4 \log_{\frac{4(1+\alpha)}{3+4\alpha}} n \rceil$ phases. In each phase, each group decides to be either a leader group or a follower group. The leader group goes clockwise for some specified number of rounds. The follower group goes counterclockwise (after randomizing the parity of its position) and decides to merge with any one leader group that it meets. Again here a follower group meets a leader group only if they are of same parity. The probability with which a follower group chooses a leader group is in proportion to the number of agents in the leader group, which is achieved via reservoir sampling. Algorithm 2 presents the pseudocode.

Similar to before, there is no value for the Byzantine agents to behave as a follower group, but they can form leader groups and present themselves to follower groups. Note that if Byzantine agents tag along with a follower group and repeatedly behaves like a leader group in consecutive rounds, then the follower group can detect this due to visual tracking. Also note that if the Byzantine agent stops tagging along, then it can never catch up with that follower group again in that phase. Therefore a Byzantine agent can act as a leader agent to a good agent only twice in a phase (once for each orientation).

13:12 Randomized Byzantine Gathering in Rings

■ **Algorithm 2** Gathering with visual tracking enabled.

```

1: function SELECTLEADER(counter)
2:   Count the number  $d$  of leader agents in the node based on the number of “I am a
   leader” messages received
3:   if  $d > 0$  then
4:      $temp \leftarrow counter$  ;  $counter \leftarrow counter + d$ 
5:     Choose this leader group with probability  $(1 - \frac{temp}{counter})$   $\triangleright$  reservoir sampling
6:   end if
7: end function
8: function FOLLOWER
9:    $counter \leftarrow 0$ 
10:  go one step counterclockwise with probability  $\frac{1}{2}$   $\triangleright$  group decision to randomize parity
   of position
11:  Call SELECTLEADER(counter)
12:  go one step counterclockwise and call SELECTLEADER(counter) for  $\lfloor \frac{n}{2} \rfloor - 1$  rounds
13:  Move to the chosen leader group (its position is predictable)
14: end function
15: for  $\lceil 4 \log_{\frac{3+4\alpha}{4}} n \rceil$  phases do
16:   Each good agent forms a group with other agents co-located with it.
17:   Decide to be a leader group or follower group with equal probability  $\triangleright$  this decision is
   made as a group
18:   Leaders go clockwise and broadcast “I am a leader” message for  $\lfloor \frac{n}{2} \rfloor$  rounds, followers
   call FOLLOWER()
19: end for

```

Time complexity analysis

Suppose in phase i , there are G_i number of good groups and H_i number of leader groups. We let $f = \alpha k$ for some α . The number of good agents is $k - f$ and out of them let L_i be leader agents. Let C_i be the number of merges between good groups. For any follower group, let the probability of choosing a good leader group to merge with be P_i . Since a Byzantine agent can appear as a leader group only once to a good follower group, we can lower bound P_i as, $P_i \geq \frac{L_i}{L_i + f}$. First lets prove a useful lemma

► **Lemma 5.** $\mathbb{E} \left[\frac{L_i}{L_i + f} \cdot (G_i - H_i) \mid G_i \right] \geq \frac{(1-\alpha)(G_i-1)}{4}$ (with expectation taken over different combinations of leader groups)

Proof. At the start of phase i let the sizes of the G_i groups be $s_1, s_2, s_3, \dots, s_{G_i}$ excluding the Byzantine agents. Out of these let the sizes of the groups which choose to be leader groups be $s_{t_1}, s_{t_2}, s_{t_3}, \dots, s_{t_{H_i}}$.

Define the set I_m as $\{(i_1, i_2, \dots, i_m) \mid i_1 < i_2 < \dots < i_m \text{ and } 1 \leq i_l \leq G_i \forall 1 \leq l \leq m\}$ for all $1 \leq j \leq G_i$. The set I_m denotes the set of possible combinations of choosing m leader groups from the G_i groups. Let $I = I_1 \cup I_2 \cup \dots \cup I_{G_i}$. The probability of exactly those groups becoming leader groups, for any element in I is $\frac{1}{2^{G_i}}$.

$$\begin{aligned}\mathbb{E}\left[\frac{L_i}{L_i+f} \times (G_i - H_i) | G_i\right] &= \sum_{e \in I} \frac{1}{2^{G_i}} \times (G_i - |e|) \times \sum_{i_j \in e} \left(\frac{\sum s_{i_j}}{\sum s_{i_j} + f}\right) \\ \mathbb{E}\left[\frac{L_i}{L_i+f} \times (G_i - H_i) | G_i\right] &= \sum_{m=1}^{G_i} \sum_{e \in I_m} \frac{1}{2^{G_i}} \times (G_i - m) \times \sum_{i_j \in e} \left(\frac{\sum s_{i_j}}{\sum s_{i_j} + f}\right)\end{aligned}$$

Now

$$\begin{aligned}\sum_{e \in I_m} \frac{1}{2^{G_i}} \times (G_i - m) \times \sum_{i_j \in e} \left(\frac{\sum s_{i_j}}{\sum s_{i_j} + f}\right) &\geq \sum_{e \in I_m} \frac{1}{2^{G_i}} \times (G_i - m) \times \sum_{i_j \in e} \left(\frac{\sum s_{i_j}}{k}\right) \\ &= \frac{1}{2^{G_i}} \times (G_i - m) \times \frac{k-f}{k} \times \binom{G_i-1}{m-1}\end{aligned}$$

Therefore

$$\begin{aligned}\mathbb{E}\left[\frac{L_i}{L_i+f} \times (G_i - H_i) | G_i\right] &\geq \sum_{m=1}^{G_i} \frac{1}{2^{G_i}} \times (G_i - m) \times \frac{k-f}{k} \times \binom{G_i-1}{m-1} \\ &\geq \frac{1}{2^{G_i}} \times \frac{k-f}{k} \times \sum_{m=1}^{G_i} (G_i - m) \times \binom{G_i-1}{m-1}\end{aligned}$$

Using a standard result from binomial mathematics, we know $\sum_{m=1}^{G_i} (G_i - m) \times \binom{G_i-1}{m-1} = 2^{G_i-2} (G_i - 1)$. Therefore

$$\mathbb{E}\left[\frac{L_i}{L_i+f} \cdot (G_i - H_i) | G_i\right] \geq \frac{(k-f)(G_i-1)}{4k}$$

Also since $f = \alpha k$

$$\mathbb{E}\left[\frac{L_i}{L_i+f} \cdot (G_i - H_i) | G_i\right] \geq \frac{(1-\alpha)(G_i-1)}{4} \quad \blacktriangleleft$$

Now lets analyze how the number of good groups varies

$$\begin{aligned}\mathbb{E}[G_{i+1} | G_i] &= G_i - \mathbb{E}[C_i | G_i] = G_i - \mathbb{E}[P_i \times (G_i - H_i) | G_i] \\ \mathbb{E}[G_{i+1} | G_i] &\leq G_i - \mathbb{E}\left[\frac{L_i}{L_i+f} \times (G_i - H_i) | G_i\right] \\ \mathbb{E}[G_{i+1} | G_i] &\leq G_i - \frac{(1-\alpha)(G_i-1)}{4} \leq G_i \times \left(\frac{3+\alpha}{4}\right) + \frac{1-\alpha}{4} \quad (\text{Using Lemma 5}) \\ \mathbb{E}[G_i] &\leq G_0 \left(\frac{3+\alpha}{4}\right)^i + 1 \quad (\text{Solving the recurrence relation}) \\ \mathbb{E}[G_{i^*}] &\leq \frac{1}{n^2} + 1 \quad (\text{Setting } i \text{ to } \log_{\frac{4}{3+\alpha}} G_0 n^2 = i^*)\end{aligned}$$

Using Markov's inequality, we get $P(G_{i^*} - 1 \geq 1) \leq \frac{1}{n^2}$. Therefore after $\log_{\frac{4}{3+\alpha}} G_0 n^2$ phases, the number of groups is 1 whp i.e. the algorithm terminates in $\log_{\frac{4}{3+\alpha}} G_0 n^2$ phases whp. Thus,

► **Theorem 6.** *Given k agents capable of visual tracking placed arbitrarily on a graph G of n nodes, there exists a randomized gathering protocol that is resilient to a strongly-adaptive Byzantine adversary that can whp gather all good agents in $\mathcal{O}\left(\frac{n \log n}{\log(4/(3+\alpha))}\right)$ rounds as long as k is greater than f where f is the number of Byzantine agents.*

13:14 Randomized Byzantine Gathering in Rings

Extension: No common orientation

Let $g = k - f$ denote the number of good agents. For simplicity we assume g is $\Omega(\log n)$. At the beginning, each agent chooses one of the two possible orientations with equal probability. Let C denote the number of good agents having one of the orientations. Then from Chernoff bound we get

$$\begin{aligned} P\left(|C - \mathbb{E}[C]| \geq \frac{\mathbb{E}[C]}{\delta}\right) &\leq 2 \exp\left(-\frac{\mathbb{E}[C]\delta^2}{3}\right) \\ P\left(|C - \frac{g}{2}| \geq \frac{g}{4}\right) &\leq 2 \exp\left(-\frac{g}{24}\right) \quad (\mathbb{E}[C] = \frac{g}{2} \text{ and setting } \delta = \frac{1}{2}) \end{aligned}$$

Since g is $\Omega(\log n)$, with high probability there are at least $\frac{g}{4}$ good agents with each orientation. After randomizing the orientation in the above mentioned way, the agents execute Algorithm 2 with agents interacting only with other agents of same orientation. Lets compute the time complexity for gathering the agents with the orientation having lesser number of good agents. We need to compute the α' value for this orientation

$$\begin{aligned} \alpha' &\leq \frac{f}{g/4 + f} \\ &= \frac{f}{(k - f)/4 + f} \\ &= \frac{\alpha}{(1 - \alpha)/4 + \alpha} \\ &= \frac{4\alpha}{1 + 3\alpha} \end{aligned}$$

Therefore the agents of this orientation gather in $\mathcal{O}\left(\frac{n \log n}{\log(4/(3+\alpha'))}\right) = \mathcal{O}\left(\frac{n \log n}{\log(4+12\alpha/(3+13\alpha))}\right)$ rounds. In these many rounds, good agents of each orientation would have gathered. Therefore we now have two groups of good agents. These groups are now gathered by running Algorithm 2 except each group chooses one of the two possible orientations with equal probability in each phase. Then in each phase, the probability that the two groups have the same orientation and same parity is $1/4$ and the probability that one of them is a leader and the other is a follower is $1/2$. Given that these events happen, the probability that the two groups combine is at least $\frac{g/4}{g/4+f} = \frac{1-\alpha}{1+3\alpha}$. Therefore in each phase the two groups combine with probability at least $\frac{1-\alpha}{8(1+3\alpha)}$. Hence the two groups combine in $\mathcal{O}\left(\frac{n \log n}{\log(8+24\alpha/7+25\alpha)}\right)$ rounds with high probability. Therefore the overall algorithm terminates in $\mathcal{O}\left(\frac{n \log n}{\log(8+24\alpha/7+25\alpha)}\right)$ rounds whp.

3 Conclusion

We studied how to exploit randomization to achieve gathering quickly in the presence of strong Byzantine agents and when agents are anonymous. Our main focus was on rings similar to intial research in other fundamental global symmetry breaking problems [19]. Our algorithms and analysis show that these problems are non-trivial and showcase the need to develop this theory further. Our work raises many follow-up questions. For example, how fast can we gather when the number of good agents is asymptotically smaller than the number of Byzantine agents? Consider the case when $k = n$ and $f = n - \mathcal{O}(1)$, can we do better than achieving gathering through coalescing random walks which takes $\tilde{\mathcal{O}}(n^2)$ time? Or is $\tilde{\Omega}(n^2)$ a lower bound for such algorithms?



Earlier works like [13] have focused on deterministic algorithms with labelled agents. While the use of randomization is clear in the anonymous setting, a natural question is whether randomization can improve the time efficiency in the case of labelled agents.

References

- 1 Noa Agmon and David Peleg. Fault-tolerant gathering algorithms for autonomous mobile robots. *SIAM J. Comput.*, 36(1):56–82, July 2006. doi:10.1137/050645221.
- 2 Steve Alpern, V. J. Baston, and Skander Essegaier. Rendezvous search on a graph. *Journal of Applied Probability*, 36(1):223–231, 1999. URL: <http://www.jstor.org/stable/3215416>.
- 3 Steve Alpern and Shmuel Gal. *The theory of search games and rendezvous*, volume 55. Springer Science & Business Media, 2006.
- 4 James Aspnes. Lower bounds for distributed coin-flipping and randomized consensus. *Journal of the ACM (JACM)*, 45(3):415–450, 1998.
- 5 Michael Ben-Or and Nathan Linial. Collective coin flipping, robust voting schemes and minima of banzhaf values. In *26th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 408–416. IEEE, 1985.
- 6 Sébastien Bouchard, Yoann Dieudonné, and Bertrand Ducourthial. Byzantine gathering in networks. *Distrib. Comput.*, 29(6):435–457, November 2016. doi:10.1007/s00446-016-0276-9.
- 7 Sébastien Bouchard, Yoann Dieudonné, and Anissa Lamani. Byzantine Gathering in Polynomial Time. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*, volume 107 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 147:1–147:15, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ICALP.2018.147.
- 8 Colin Cooper, Robert Elsässer, Hirotaka Ono, and Tomasz Radzik. Coalescing random walks and voting on graphs. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing*, PODC '12, pages 47–56, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2332432.2332440.
- 9 Gianlorenzo D’angelo, Gabriele Di Stefano, and Alfredo Navarra. Gathering on rings under the look—compute—move model. *Distrib. Comput.*, 27(4):255–285, August 2014. doi:10.1007/s00446-014-0212-9.
- 10 Gianlorenzo D’angelo, Alfredo Navarra, and Nicolas Nisse. A unified approach for gathering and exclusive searching on rings under weak assumptions. *Distrib. Comput.*, 30(1):17–48, February 2017. doi:10.1007/s00446-016-0274-y.
- 11 Shantanu Das et al. Mobile agents in distributed computing: Network exploration. *Bulletin of EATCS*, 1(109), 2013.
- 12 Xavier Défago, Maria Potop-Butucaru, and Philippe Raipin Parvédy. Self-stabilizing gathering of mobile robots under crash or byzantine faults. *Distributed Comput.*, 33(5):393–421, 2020. doi:10.1007/s00446-019-00359-x.
- 13 Yoann Dieudonné, Andrzej Pelc, and David Peleg. Gathering despite mischief. *ACM Trans. Algorithms*, 11(1), August 2014. doi:10.1145/2629656.
- 14 R. Eguchi, N. Kitamura, and T. Izumi. Fast neighborhood rendezvous. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 168–178, Los Alamitos, CA, USA, December 2020. IEEE Computer Society. doi:10.1109/ICDCS47774.2020.00030.
- 15 Uriel Feige. Noncryptographic selection protocols. In *40th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 142–152. IEEE, 1999.
- 16 Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro. *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.
- 17 Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Giovanni Viglietta. Distributed computing by mobile robots: uniform circle formation. *Distributed Comput.*, 30(6):413–457, 2017. doi:10.1007/s00446-016-0291-x.

- 18 Jion Hirose, Junya Nakamura, Fukuhito Ooshita, and Michiko Inoue. Gathering with a strong team in weakly byzantine environments. In *International Conference on Distributed Computing and Networking 2021, ICDCN '21*, pages 26–35, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3427796.3427799.
- 19 Daniel S. Hirschberg and James B Sinclair. Decentralized extrema-finding in circular configurations of processors. *Communications of the ACM*, 23(11):627–628, 1980.
- 20 Zool Hilmi Ismail and Nohaidda Sariff. A survey and analysis of cooperative multi-agent robot systems: Challenges and directions. In Eflen Gorrostieta Hurtado, editor, *Applications of Mobile Robots*, chapter 1. IntechOpen, Rijeka, 2019. doi:10.5772/intechopen.79337.
- 21 Taisuke Izumi, Tomoko Izumi, Sayaka Kamei, and Fukuhito Ooshita. Feasibility of polynomial-time randomized gathering for oblivious mobile robots. *IEEE Transactions on Parallel and Distributed Systems*, 24(4):716–723, 2013. doi:10.1109/TPDS.2012.212.
- 22 Tomoko Izumi, Taisuke Izumi, Sayaka Kamei, and Fukuhito Ooshita. Mobile robots gathering algorithm with local weak multiplicity in rings. In *Proceedings of the 17th International Conference on Structural Information and Communication Complexity, SIROCCO'10*, pages 101–113, Berlin, Heidelberg, 2010. Springer-Verlag. doi:10.1007/978-3-642-13284-1_9.
- 23 Gangshan Jing, Yuanshi Zheng, and Long Wang. Consensus of multiagent systems with distance-dependent communication networks. *IEEE Transactions on Neural Networks and Learning Systems*, PP, August 2016. doi:10.1109/TNNLS.2016.2598355.
- 24 Ralf Klasing, Adrian Kosowski, and Alfredo Navarra. Taking advantage of symmetries: Gathering of many asynchronous oblivious robots on a ring. *Theoretical Computer Science*, 411(34):3235–3246, 2010. doi:10.1016/j.tcs.2010.05.020.
- 25 Evangelos Kranakis and Danny Krizanc. *Mobile Agents and Exploration*, pages 1338–1341. Springer New York, New York, NY, 2016. doi:10.1007/978-1-4939-2864-4_242.
- 26 Silvio Micali and Tal Rabin. Collective coin tossing without assumptions nor broadcasting. In *Conference on the Theory and Application of Cryptography*, pages 253–266. Springer, 1990.
- 27 Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis*. Cambridge University Press, USA, 2nd edition, 2017.
- 28 Fukuhito Ooshita, Shinji Kawai, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. Randomized gathering of mobile agents in anonymous unidirectional ring networks. *IEEE Trans. Parallel Distributed Syst.*, 25(5):1289–1296, 2014. doi:10.1109/TPDS.2013.259.
- 29 Michael Saks. A robust noncryptographic protocol for collective coin flipping. *SIAM Journal on Discrete Mathematics*, 2(2):240–244, 1989.
- 30 Masahiro Shibata, Toshiya Mega, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. Uniform deployment of mobile agents in asynchronous rings. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC '16*, pages 415–424, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2933057.2933093.
- 31 Yuichi Sudo, Daisuke Baba, Junya Nakamura, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. An agent exploration in unknown undirected graphs with whiteboards. In *Proceedings of the Third International Workshop on Reliability, Availability, and Security, WRAS '10*, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1953563.1953570.
- 32 Ichiro Suzuki and Masafumi Yamashita. Distributed anonymous mobile robots: Formation of geometric patterns. *SIAM Journal on Computing*, 28(4):1347–1363, 1999. doi:10.1137/S009753979628292X.
- 33 Masashi Tsuchida, Fukuhito Ooshita, and Michiko Inoue. Byzantine gathering in networks with authenticated whiteboards. In Sheung-Hung Poon, Md. Saidur Rahman, and Hsu-Chun Yen, editors, *WALCOM: Algorithms and Computation*, pages 106–118, Cham, 2017. Springer International Publishing.
- 34 Xiaoping Yun, Gokhan Alptekin, and Okay Albayrak. Line and circle formation of distributed physical mobile robots. *J. Field Robotics*, 14(2):63–76, 1997.

Gathering of Mobile Robots with Defected Views*

Yonghwan Kim  

Nagoya Institute of Technology, Aichi, Japan

Masahiro Shibata  

Kyushu Institute of Technology, Fukuoka, Japan

Yuichi Sudo  

Hosei University, Tokyo, Japan

Junya Nakamura  

Toyohashi University of Technology, Aichi, Japan

Yoshiaki Katayama  

Nagoya Institute of Technology, Aichi, Japan

Toshimitsu Masuzawa  

Osaka University, Japan

Abstract

An autonomous mobile robot system consisting of many mobile computational entities (called *robots*) attracts much attention of researchers, and it is an emerging issue for a recent couple of decades to clarify the relation between the capabilities of robots and solvability of the problems.

Generally, each robot can observe all other robots as long as there are no restrictions on visibility range or obstructions, regardless of the number of robots. In this paper, we provide a new perspective on the observation by robots; a robot cannot necessarily observe all other robots regardless of distances to them. We call this new computational model the *defected view model*. Under this model, in this paper, we consider the *gathering* problem that requires all the robots to gather at the same non-predetermined point and propose two algorithms to solve the gathering problem in the adversarial $(N, N - 2)$ -defected model for $N \geq 5$ (where each robot observes at most $N - 2$ robots chosen adversarially) and the distance-based $(4, 2)$ -defected model (where each robot observes at most two robots closest to itself), respectively, where N is the number of robots. Moreover, we present an impossibility result showing that there is no (deterministic) gathering algorithm in the adversarial or distance-based $(3, 1)$ -defected model, and we also show an impossibility result for the gathering in a relaxed $(N, N - 2)$ -defected model.

2012 ACM Subject Classification Theory of computation → Self-organization

Keywords and phrases mobile robot, gathering, defected view model

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2022.14

Related Version *Previous Version*: <https://doi.org/10.4230/LIPIcs.DISC.2022.46>

Funding This work was supported in part by JSPS KAKENHI Grant Numbers 18K18031, 19H04085, 19K11823, 20H04140, 20KK0232, 21K17706, 22K11971, and Foundation of Public Interest of Tatematsu.

1 Introduction

An autonomous mobile robot system is a distributed system consisting of many mobile computational entities (called *robots*) with limited capabilities, e.g., robots cannot distinguish other robots, or cannot remember their any past actions. The robots operate autonomously

* A part of this paper was presented in *the 36th International Symposium on Distributed Computing (DISC 2022)* as a brief announcement.



© Yonghwan Kim, Masahiro Shibata, Yuichi Sudo, Junya Nakamura, Yoshiaki Katayama, and Toshimitsu Masuzawa;

licensed under Creative Commons License CC-BY 4.0

26th International Conference on Principles of Distributed Systems (OPODIS 2022).

Editors: Eshcar Hillel, Roberto Palmieri, and Etienne Rivière; Article No. 14; pp. 14:1–14:18



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

and cooperatively; each robot observes the other robots (*Look*), computes the destination (*Compute*), and moves to the destination (*Move*). Each robot autonomously and cyclically performs the above operations to achieve the given common goal. Since an autonomous mobile robot system has been firstly introduced in [21], the literature [17, 18, 19, 21] provides a formal discussion on the capabilities of the robots for the distributed coordination (e.g., gathering, scattering, or pattern formation), and many researchers are interested in clarifying the relationship between the capabilities of the robots and solvability of the problems.

Generally, in *Look* operation, each robot can observe all other robots (within its visibility range if the range is limited). In other words, each robot can take a snapshot consisting of all other robots' (relative) positions in its *Look* operation, i.e., each robot temporarily remembers the positions of up to $N - 1$ robots, where N is the total number of robots. From the practical viewpoint, we claim that a robot with low functionality may not have such large working memory. That is, the main question we address in this paper is “*what occurs if a robot cannot observe some of the other robots?*”. More precisely, “*how many other robots should be observed to achieve the goals of the problems?*”.

Related works. The gathering problem [16], which requires all the robots to move to a common (non-predetermined) position, is a fundamental problem for autonomous mobile robot systems. There are many studies about the gathering of autonomous mobile robots; Cieliebak et al. presented the first algorithm to achieve the gathering from any arbitrary configuration [4], Klasing et al. studied the gathering of mobile robots in one node of an anonymous unoriented ring [13], D’Angelo et al. introduced a gathering algorithm of robots without multiplicity detection on grids and trees [5], and many works for the gathering of robots with dynamic (or inaccurate) compasses are also introduced [9, 10, 11, 20]. The capability of the robots deeply affects the solvability of the gathering problem, thus some investigations about the required capability or impossibility are introduced [16, 17]. However, all of these works assume that each robot can observe all other robots within its visibility range if there is no obstruction (e.g., any opaque robot) between the robots.

The works most related to this paper are those with the limited visibility range [1, 3, 7, 12, 14]. The robots with the limited visibility cannot necessarily observe all robots, which is similar to the defected view model we propose. But visibility is limited by distance in the limited visibility model and thus all robots can be observed when they gather closely enough. On the other hand, the defected view model cannot guarantee such a full view of the robots. As another well-related work, Heriban et al. studied some problems of robots with uncertain visibility sensors [8]: if the distance between two robots is longer than the visibility range, the two robots adversarially observe each other. However, also in this study, every robot can observe all other robots within the visibility range regardless of the number of robots. The works for fault-tolerance [2, 6, 15] are also closely related to this paper. The defected view can be considered as a new type of fault in autonomous mobile robot systems.

Contribution. To provide some answers for the above research questions, we propose a new computational model with restriction on the number of robots that each robot can observe, named the *defected view model*, where each robot observes only k other robots for $1 \leq k < N - 1$. This assumption naturally arises by considering some issues for robots with low functionality such as (1) each robot does not have enough working memory to store the entire observation result, (2) each robot may miss some of observation results due to memory failure, or (3) each robot fails to observe some of other robots by sensing failure. It is obvious that when k becomes the lower, the problem becomes the harder (possibly impossible) to

solve. We consider two different defected view models regarding which k robots are observed: the adversarial (N,k) -defected model and the distance-based (N,k) -defected model. In the former, each robot observes the other k robots determined adversarially, and in the latter, each robot observes the other k robots closest to its current position.

More precisely, the k robots that each robot r can observe are chosen from the robots located at points different from r 's current position. Concerning r 's current position, r can detect only whether another robot exists at the point or not (so called the *weak multiplicity detection*). Such an assumption that the robots at r 's current position are excluded from the candidates of the observed k robots is motivated by the following observation: each robot r observes the robots at remote points and those at r 's current position by different ways usually. Each robot observes the remote robots by, for example, a radar sensor or a vision sensor, but senses the other robots at the same point by, for example, a contact sensor.

As the first step of the gathering in the defected view model, we investigate only the case of $k = N - 2$. The main contributions of this paper are as follows: (1) we propose a gathering algorithm in the adversarial $(N, N - 2)$ -defected model for any $N \geq 5$, (2) we present another algorithm to solve the gathering problem in the distance-based $(4,2)$ -defected model, and (3) we provide the impossibility result showing that there is no (deterministic) algorithm to solve the gathering problem in the adversarial or distance-based $(3,1)$ -defected model. Moreover, we present another impossibility result in a naturally relaxed $(N, N - 2)$ -defected model where the observed k robots can contain the robots at the observer's current position. This impossibility result shows the necessity of the assumption that the observed $N - 2$ robots should be chosen from robots other than those located at the observer's current position.

The rest of this paper is organized as follows: Section 2 presents the system model (including two defected view models) and problem definition; Section 3 introduces an algorithm to solve the gathering problem in the adversarial $(N, N - 2)$ -defected model for any $N \geq 5$; Section 4 gives a gathering algorithm in the distance-based $(4,2)$ -defected model; Section 5 shows two impossibility results showing that there is no algorithm in the adversarial or distance-based $(3,1)$ -defected model and the relaxed adversarial $(N, N - 2)$ -defected model; and Section 6 concludes the paper and provides some open problems.

2 Model and Problem Definition

2.1 Robots

Let $R = \{r_1, r_2, \dots, r_N\}$ be the set of N autonomous mobile robots deployed in a plane. Robots are indistinguishable by their appearance (i.e., identical), execute the same algorithm (i.e., uniform or homogeneous), and have no memory (i.e., oblivious). There is no geometrical agreement; robots do not agree on any axis, the unit distance, or chirality. A point in the plane is called an *occupied point* if there exists a robot at the point. We allow two or more robots to occupy the same point at the same time. We call a robot a *single robot* if the point occupied by the robot has no other robot. Otherwise, we call it an *accompanied robot*.

Each robot cyclically performs the three operations, *Look*, *Compute*, and *Move*: (**Look**) a robot obtains the positions (based on its local coordinate system centered on itself) of all other observed robots, (**Compute**) a robot determines the destination according to the given algorithm based on the result of *Look* operation. Since each robot has no memory, the result of *Compute* is determined only by the result of *Look* operation, and (**Move**) a robot moves to the destination computed in *Compute* operation. We assume *rigid movement* which ensures each robot can reach the destination during its *Move* operation, i.e., a robot never stops before it reaches its destination.

2.2 Schedule and Configuration

We assume a fully-synchronous scheduler (FSYNC): all robots fully-synchronously perform their operations (*Look*, *Compute*, and *Move*). This means that all robots perform the same operation at the same time instant and duration. We call the time duration in which all robots perform the three operations (*Look*, *Compute*, and *Move*) once a *round*.

Let *configuration* C_t be the set of the (global) coordinates of all robots at a given time t : $C_t = \{(r_{1.x}^t, r_{1.y}^t), (r_{2.x}^t, r_{2.y}^t) \dots (r_{N.x}^t, r_{N.y}^t)\}$, where $r_{i.x}^t$ (resp. $r_{i.y}^t$) is the X -coordinate (resp. Y -coordinate) of robot r_i at time t . Note that no robot knows its global coordinate. Configuration C_t is changed into another configuration C_{t+1} after one round (i.e., all robots execute the three operations once).

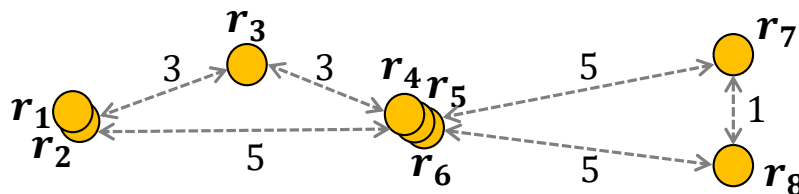
2.3 Observation: Visibility Range and Multiplicity Detection

We basically assume that every robot has unlimited visibility range, i.e., any two robots can observe each other regardless of their distance, while we introduce in Definition 1 the defect in the information obtained by *Look* operation. Moreover, we assume a *weak multiplicity detection*, i.e., each robot cannot get the exact number of robots occupying the same point but can distinguish whether the point is occupied by one robot or by multiple robots. This implies that when each robot observes any point, it can distinguish the three cases: there is *no robot*, *one robot*, or *two or more robots* at the point.

We consider a *defected view* such that each robot may not observe all other robots. We define the (N, k) -defected model, where $1 \leq k < N$ as follows:

► **Definition 1** ((N, k) -defected model). *Each robot r can get from Look operation the set of occupied points (in its coordinate system) where k robots not accompanied with r are located (i.e., the k robots contains no robot located at r 's current point). When the number of robots not accompanied with r is k or less, all such robots are observed. The weak multiplicity detection concerning the k robots is assumed: a point occupied by only one of the k robots can be distinguished from that occupied by two or more of the k robots.*

Note that the $(N, N - 1)$ -defected model is equivalent to the commonly used model (with the weak multiplicity detection) where each robot can observe all robots. The (N, k) -defected



■ **Figure 1** An example configuration by 8 robots.

To help to understand, we explain the model using examples. Figure 1 illustrates an example configuration by 8 robots; $R = \{r_1, r_2, \dots, r_8\}$. Robots r_1 and r_2 (resp. r_4, r_5 and r_6) are accompanied, and the other robots are single. The dotted arrow between robots represents the distance between the points occupied by the robots. Let p_i denote the point occupied by robot r_i . Now we explain the models as follows:

- **The adversarial (8,4)-defected model.** In this model, each robot observes 4 other robots chosen adversarially. Assume that robot r_3 observes 4 robots, r_1, r_2, r_6 , and r_8 . In this case, robot r_3 gets a set of points $P^{r_3} = \{p_1^*, p_3, p_6, p_8\}$ including point p_3 occupied by r_3 itself, where p_i^* denotes that p_i is recognized to be occupied by two or more robots. Robot r_3 knows that two or more robots exist at point p_1 because both robots r_1 and r_2 are chosen, however, r_3 does not know that there is another robot at p_6 because it observes only r_6 at p_6 . Robot r_4 (or r_5, r_6) observes 4 robots among 5 robots, r_1, r_2, r_3, r_7 , and r_8 . If robots r_1, r_3, r_7 , and r_8 are chosen, $P^{r_4} = \{p_1, p_3, p_4^*, p_7, p_8\}$ holds, which means that r_4 observes all points, however, it does not know that another robot exists at p_1 (and the other points except for p_4^*). Robot r_4 can know that point p_4 is occupied by another robot other than itself. If robot r_4 observes robots r_1, r_2, r_7 , and r_8 , $P^{r_4} = \{p_1^*, p_4^*, p_7, p_8\}$ holds, which means that robot r_4 knows there exist two or more robots at p_1 , but it cannot observe point p_3 occupied by robot r_3 . Notice that r_5 and r_6 located at p_4 are allowed to observe the set of points different from those observed by r_4 .
- **The distance-based (8,3)-defected model.** In this model, each robot observes 3 closest robots to itself. Robot r_7 observes 3 robots, r_8 (the closest one) and two robots among three robots at point p_4 , thus $P^{r_7} = \{p_4^*, p_7, p_8\}$ always holds. Robot r_4 observes robot r_3 (the closest one) and two robots among 4 robots, r_1, r_2, r_7 , and r_8 , which are the same distance apart. Note that the observed robots are determined in an arbitrary way, thus in this case, P^{r_4} becomes one among $\{p_1^*, p_3, p_4^*\}$, $\{p_1, p_3, p_4^*, p_7\}$, $\{p_1, p_3, p_4^*, p_8\}$, or $\{p_3, p_4^*, p_7, p_8\}$.

It is obvious that the adversarial (N,k) -defected model is weaker¹ than the distance-based one, that is, any algorithm to achieve the gathering in the adversarial (N,k) -defected model works correctly also in the distance-based (N,k) -defected model.

2.4 Problem Definition: Gathering

We define the gathering problem as follows.

► **Definition 2** (The Gathering Problem). *Given a set of N robots located at arbitrary points. Algorithm A solves the gathering problem if A satisfies all the following conditions:*

- (1) *algorithm A eventually reaches a configuration such that no robot can move, and*
- (2) *when the algorithm A terminates, all the robots are located at the same point.*

¹ Strictly speaking, we do not know the adversarial (N,k) -defected model is properly weaker than the distance-based one yet; it is obvious that the adversarial (N,k) -defected model is NOT stronger than the distance-based one.

3 Algorithm in the Adversarial $(N, N - 2)$ -defected Model where $N \geq 5$

Algorithm 1 presents an algorithm for robot r_i to achieve the gathering in the adversarial $(N, N - 2)$ -defected model where $N \geq 5$. We use two functions defined as follows:

- $\text{OPSET}()$: a function that returns a set of points $\{p \mid p \text{ is occupied by } r_i \text{ or by the robots that } r_i \text{ observed}\}$
- $\text{isMulti}(p)$: a function that returns TRUE if point p is occupied by two or more robots that r_i observed (weak multiplicity), otherwise FALSE.

The algorithm adopts, as the destination of robot r_i , the center of the smallest enclosing circle (SEC) of the occupied points that r_i observed in the *Look* operation. Before proving the correctness of the algorithm, we show some fundamental properties of the SEC of points in a plane.

■ **Algorithm 1** Gathering algorithm in the adversarial $(N, N - 2)$ -defected model where $N \geq 5$.

```

1: if  $\forall p \in \text{OPSET}() : \text{isMulti}(p) = \text{TRUE}$  then
2:   move to the center of the smallest enclosing circle of  $\text{OPSET}()$ 
3: else if  $(r_i \text{ is single}) \wedge (\exists p \in \text{OPSET}() : \text{isMulti}(p) = \text{TRUE})$  then
4:   move to an arbitrary point  $p \in \text{OPSET}()$  such that  $\text{isMulti}(p) = \text{TRUE}$ 
5: else if  $\forall p \in \text{OPSET}() : \text{isMulti}(p) = \text{FALSE}$  then
6:   move to the center of the smallest enclosing circle of  $\text{OPSET}()$ 
7: end if           ▷ No action if  $(r_i \text{ is accompanied}) \wedge (\exists p \in \text{OPSET}() : \text{isMulti}(p) = \text{FALSE})$ 

```

► **Proposition 3.** *Let P be a set of n distinct points in a plane and C be the SEC of P . The following properties hold.*

1. *The SEC of P is unique.*
2. *Let $p \in P$ be any point (if exists) properly inside C , C is the SEC of $P \setminus \{p\}$.*
3. *When there exist three points $p_1, p_2, p_3 \in P$ on the boundary of C that form an acute or right triangle, C is the SEC of $\{p_1, p_2, p_3\}$.*
4. *When three or more points in P are on the boundary of C , there exist three points $p_1, p_2, p_3 \in P$ on the boundary of C that form an acute or right triangle. ◀*

A key property of the $(N, N - 2)$ -defected model used in the following proofs is that *any accompanied robot can observe all the robots* (but only with the weak multiplicity detection).

► **Lemma 4.** *In the adversarial $(N, N - 2)$ -defected model ($N \geq 5$), Algorithm 1 solves the gathering problem in two rounds from any configuration where there exist three or more accompanied robots.*

Proof. When every robot is accompanied, each robot detects all the occupied points in the *Look* operation and recognizes that each of them is occupied by multiple robots. Every robot moves to the center of the SEC of all the occupied points (by lines 1 and 2 in Algorithm 1) and thus the gathering is achieved in one round.

When there exists a single robot r , every accompanied robot observes r and does not move (see line 7 in Algorithm 1). Every single robot misses at most one accompanied robot in its *Look* operation and can detect at least one point occupied by multiple robots: a point occupied by three or more robots (if exists) or one of the points each occupied by two robots. Each single robot moves to one of such points (by lines 3 and 4 in Algorithm 1), which results in the configuration where every robot is accompanied. Thus the gathering is achieved in the next round as shown above. ◀

Notice that Lemma 4 holds for $N \geq 3$.

► **Lemma 5.** *In the adversarial $(N, N - 2)$ -defected model ($N \geq 5$), Algorithm 1 solves the gathering problem in two rounds from any configuration where there exist only two accompanied robots.*

Proof. Let r_1 and r_2 be the two accompanied robots. Robots r_1 and r_2 observe all robots and recognize that single robots exist, which makes r_1 and r_2 stay at the current point.

Now consider actions of single robots. A single robot r misses one robot in its *Look* operation, which implies that r observes (a) both r_1 and r_2 or (b) only one of r_1 and r_2 . In case (a), r moves to the point, say p_a , occupied by r_1 and r_2 . In case (b), r moves to the center, say p_b , of the SEC of all the occupied points. Thus after one round, all the robots are located at p_a or p_b . Note that p_a is occupied by multiple robots including r_1 and r_2 .

When p_b is not occupied by any robot, the gathering is already achieved. When p_b is occupied by multiple robots, the robots at p_b observe all the robots. Thus, all the robots move to the center of the SEC of p_a and p_b (or the midpoint of p_a and p_b) in the next round (by lines 1 and 2 in Algorithm 1), which achieves the gathering. When p_b is occupied by only one robot r , r detects that p_a is occupied by multiple robots and moves to p_a in the next round (by lines 3 and 4 in Algorithm 1) while the robots at p_a recognize that p_b is occupied by only one robot and does not move (see line 7 in Algorithm 1). Thus, the gathering is achieved. ◀

Notice that Lemma 5 holds for $N \geq 4$.

► **Lemma 6.** *In the adversarial $(N, N - 2)$ -defected model ($N \geq 5$), Algorithm 1 solves the gathering problem in three rounds from any configuration where all robots are single.*

Proof. Each robot misses one robot in its *Look* operation. When there exist two robots r_1 and r_2 that miss the same robot, r_1 and r_2 get the same point set $\text{OPSET}()$ and moves to the center of the SEC of $\text{OPSET}()$ (by lines 5 and 6 in Algorithm 1). From Lemmas 4 and 5, two additional rounds are enough to achieve the gathering.

When no two robots miss the same robot, for any pair of two distinct robots r_1 and r_2 , the robot missing r_1 is different from the robot missing r_2 . Let C be the SEC of all the occupied N points. First, consider the case that two (or more) robots r_a and r_b are located properly inside C . The SEC of $R \setminus \{r_a\}$ is equal to the SEC of $R \setminus \{r_b\}$ (that is C from the second property of Proposition 3), which implies that the two robots observing $R \setminus \{r_a\}$ and $R \setminus \{r_b\}$ move to the same point (or the center of the SEC). From Lemmas 4 and 5, two additional rounds are enough to achieve the gathering.

Second, consider the case that $N - 1$ or N robots are on the boundary of C . From the last property of Proposition 3, there exist three robots r_1, r_2, r_3 on the boundary of C that form an acute or right triangle. There exist two robots r_4 and r_5 other than r_1, r_2, r_3 from $N \geq 5$. Both the robots observing $R \setminus \{r_4\}$ and $R \setminus \{r_5\}$ observe all of r_1, r_2, r_3 . The third property of Proposition 3 implies that the two robots find the same SEC (or the SEC of r_1, r_2, r_3), which implies that they move to the same point (or the center of the SEC) (by lines 5 and 6 in Algorithm 1). From Lemmas 4 and 5, two additional rounds are enough to achieve the gathering. ◀

From Lemmas 4, 5 and 6, the following theorem holds.

► **Theorem 7.** *In the adversarial $(N, N - 2)$ -defected model ($N \geq 5$), Algorithm 1 solves the gathering problem in three rounds.* ◀

Algorithm 1 cannot solve the gathering problem for the case of $N = 4$. Assume that four robots, $R = \{r_0, r_1, r_2, r_3\}$. Three robots r_1, r_2 and r_3 are deployed to form an equilateral triangle as Figure 11 and r_0 is located at the center of the triangle (i.e., point p_c in Figure 11). Consider the case that r_i observes $r_{(i+1) \bmod 3}$ and $r_{(i+2) \bmod 3}$ for each i ($0 \leq i \leq 3$). According to Algorithm 1, r_0 moves to the midpoint of r_1 and r_2 , r_1 moves to p_0 , r_2 moves to the midpoint of r_2 and r_3 , and r_3 moves to the midpoint of r_3 and r_1 . In the resultant configuration, r_0, r_2 and r_3 form an equilateral triangle and r_1 is located at the center p_1 of the triangle, which shows by repeating the argument that the gathering is never achieved.

Thus we need another gathering algorithm for the adversarial (4, 2)-defected model, however, we do not know whether the gathering problem in the adversarial (4,2)-defected model is solvable or not yet. In the next section, we present an algorithm to solve the gathering problem in the distance-based (4,2)-defected model.

4 Algorithm in the Distance-based (4,2)-defected Model

In this model, the number of robots is 4 and each robot observes at most two occupied points other than its current location (three points in total including the one occupied by itself). In other words, the observation result of each robot forms a triangle (by three points/robots) when every robot is single. The strategy of the proposed algorithm is to determine one unique point from the formed triangle. Therefore, two robots observing the same three occupied points (including its location) move to the same point according to the proposed algorithm. If two or more robots are accompanied, the gathering can be achieved as the same manner introduced in Algorithm 1. Obviously, in this strategy, we have to consider the case so that all 4 robots observe different triangles. We resolve this problem by the geometrical property (recall that each robot cannot observe the farthest robot from itself in the distance-based defected model).

■ **Algorithm 2** Gathering algorithm for robot r_i in the distance-based (4,2)-defected model.

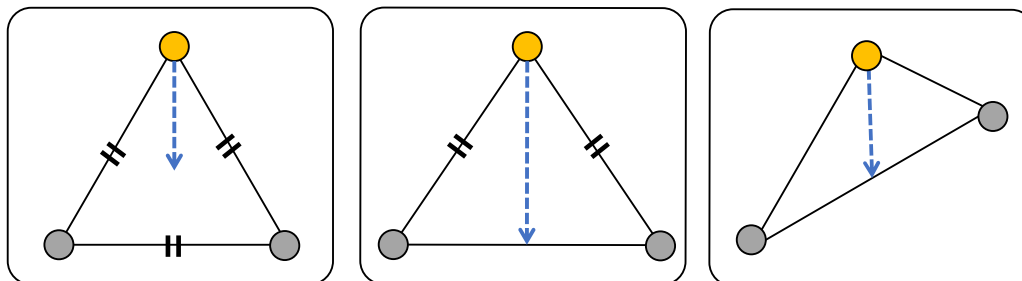
```

1: if  $\forall p \in \text{OPSET}() : \text{isMulti}(p) = \text{TRUE}$  then
2:   move to the center of the smallest enclosing circle of  $\text{OPSET}()$ 
3: else if  $(r_i \text{ is single}) \wedge (\exists p \in \text{OPSET}() : \text{isMulti}(p) = \text{TRUE})$  then
4:   move to an arbitrary point  $p \in \text{OPSET}()$  such that  $\text{isMulti}(p) = \text{TRUE}$ 
5: else if  $\forall p \in \text{OPSET}() : \text{isMulti}(p) = \text{FALSE}$  then
6:   if  $\text{OPSET}()$  forms an equilateral triangle then
7:     move to the center of the triangle (i.e., incenter) ▷ Rule 1
8:   else if  $\text{OPSET}()$  forms an isosceles triangle then
9:     move to the midpoint of the base of the triangle ▷ Rule 2
10:  else ▷ the other triangle or collinear three points
11:    move to the midpoint of the longest line ▷ Rule 3
12:  end if
13: end if ▷ No action if  $(r_i \text{ is accompanied}) \wedge (\exists p \in \text{OPSET}() : \text{isMulti}(p) = \text{FALSE})$ 

```

Algorithm 2 presents an algorithm to achieve the gathering in the distance-based (4,2)-defected model (two functions, $\text{OPSET}()$ and $\text{isMulti}()$, are the same functions described in Section 3). Each robot which does not observe any accompanied robots executes one among three rules (lines from 6 to 11 in Algorithm 2). Figure 2 illustrates these three rules. If a robot observes an equilateral triangle (i.e., the points observed by the robot form an equilateral triangle), it moves to the center of the triangle (Figure 2(a)), and if it observes

an isosceles triangle, it moves to the midpoint of the base of the triangle (Figure 2(b)). In the other case, it moves to the center point of the longest line of the triangle (Figure 2(c)). It is obvious that two robots observing the same set of points (i.e., the same *view*: $r_i.\text{OPSET}() = r_j.\text{OPSET}()$, where $i \neq j$), move to the same point according to Algorithm 2. Hence the following lemma holds.



(a) Case of an equilateral triangle. (b) Case of an isosceles triangle. (c) The other case.

■ **Figure 2** Three rules in Algorithm 2.

► **Lemma 8.** *In any configuration where no robot is accompanied, if two or more robots have the same view, the robots move to the same point in one round by Algorithm 2.* ◀

In Algorithm 2, actions when a robot observes any accompanied robots (including itself) are the exactly same as Algorithm 1 (lines from 1 to 4 in both algorithms). Lemmas 4 and 5 are proved for the adversarial defected model but obviously hold for the distance-based defected model. Remind that Lemmas 4 and 5 hold for $N \geq 3$ and $N \geq 4$, respectively. Moreover, we can see from the proof that the gathering is achieved in one round (not two rounds) in Lemma 4 for $N = 4$. Thus, the following lemma holds.

► **Lemma 9.** *In the distance-based (4,2)-defected model, Algorithm 2 solves the gathering in one round (resp. two rounds) when there exist three or more (resp. only two) accompanied robots.* ◀

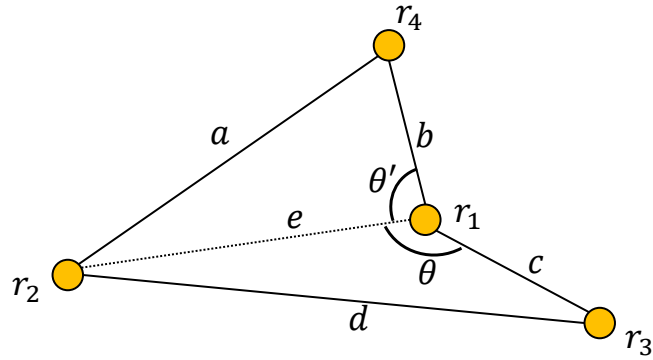
Even when all 4 robots are single, if two or more robots observe the same set of points, the robots move to the same point (by Lemma 8), thus the gathering is achieved by Lemma 9.

Now we show that the gathering is eventually achieved in any configuration where all 4 (single) robots have the different views (i.e., observe the different set of points).

► **Lemma 10.** *In the distance-based (4,2)-defected model, if all robots have the different views, the shape formed by the robots is a convex quadrilateral.*

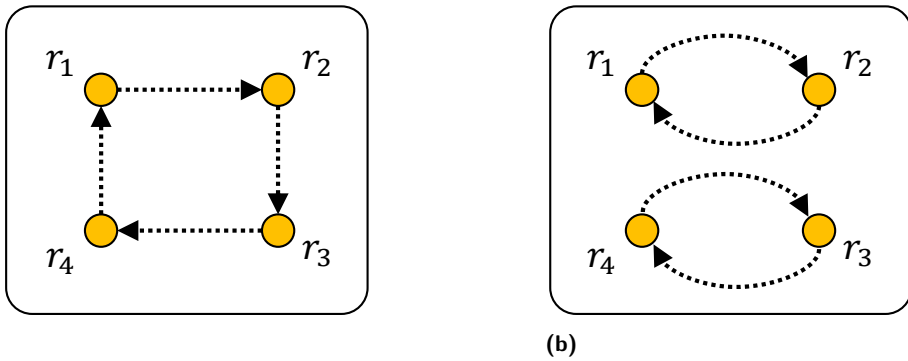
Proof. We prove the contraposition of the lemma: if the robots do not form a convex quadrilateral, there exist two robots having the same view.

Assume that the 4 robots, from r_1 to r_4 , form a concave quadrilateral as Figure 3 (Note that we can also assume that the robots form a triangle (i.e., three robots are collinear), it can be also proved in the same manner). A concave quadrilateral has an interior angle which is larger than 180° , so we assume robot r_1 is located at the point with such an angle as Figure 3. Let e be the line $\overline{r_1 r_2}$, either angle $\angle r_2 r_1 r_4$ or angle $\angle r_2 r_1 r_3$ is an obtuse angle (i.e., angle larger than 90°) because interior angle $\angle r_4 r_1 r_3$ is larger than 180° . Without loss of generality, we assume angle $\angle r_2 r_1 r_3$ is an obtuse angle (denoted by θ). Due to $\theta > 90^\circ$, d is longer than c and e (see Figure 3). This implies that robot r_3 observes r_1 and robot r_2 also



■ **Figure 3** An example of a concave quadrilateral.

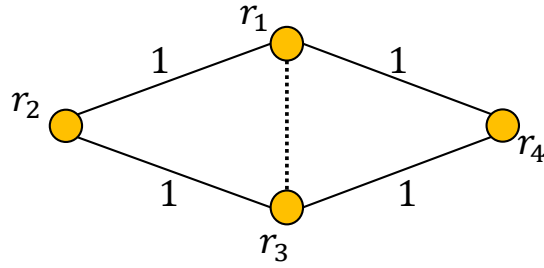
observes r_1 (because the farthest robot is missed in the distance-based defected model). If angle $\angle r_2 r_1 r_4$ (denoted by θ') is also an obtuse angle, robot r_4 also observes r_1 by the same reason. As a result, all robots observe r_1 (including r_1 itself) and the lemma holds because there are two or more robots which have the same view by the pigeonhole principle. If angle $\angle r_2 r_1 r_4$ is an acute angle (i.e., angle smaller than 90°) or a right angle, $\theta + \theta' < 270^\circ$ holds. This means that an exterior angle $\angle r_4 r_1 r_3$ (i.e., $360^\circ - \theta - \theta'$) is an obtuse angle, thus b is



■ **Figure 4** Directed graphs representing unobserved relation.

Assume the case as Figure 4(a): robot r_1 cannot observe r_2 , robot r_2 cannot observe r_3 , and so on. $\overline{r_1 r_4} \leq \overline{r_1 r_2}$ holds because robot r_1 cannot observe r_2 . For the same reason, $\overline{r_1 r_2} \leq \overline{r_2 r_3}$, $\overline{r_2 r_3} \leq \overline{r_3 r_4}$, and $\overline{r_3 r_4} \leq \overline{r_1 r_2}$ also hold. Therefore, $\overline{r_1 r_4} \leq \overline{r_1 r_2} \leq \overline{r_2 r_3} \leq \overline{r_3 r_4} \leq \overline{r_1 r_2}$ holds, thus $\overline{r_1 r_2} = \overline{r_2 r_3} = \overline{r_3 r_4} = \overline{r_1 r_4}$ holds. For simplicity, we assume that the length of $\overline{r_1 r_2}$ is 1.

Now we consider the triangle $\triangle r_1 r_2 r_3$. Due to $\overline{r_1 r_2} = \overline{r_2 r_3}$, triangle $\triangle r_1 r_2 r_3$ is an isosceles triangle (the base is $\overline{r_1 r_3}$). Similarly, triangle $\triangle r_1 r_3 r_4$ is also an isosceles triangle which has line $\overline{r_1 r_3}$ as the base. Line $\overline{r_1 r_3}$ is the common base of these two isosceles triangles, thus the locations of 4 robots are as Figure 5.



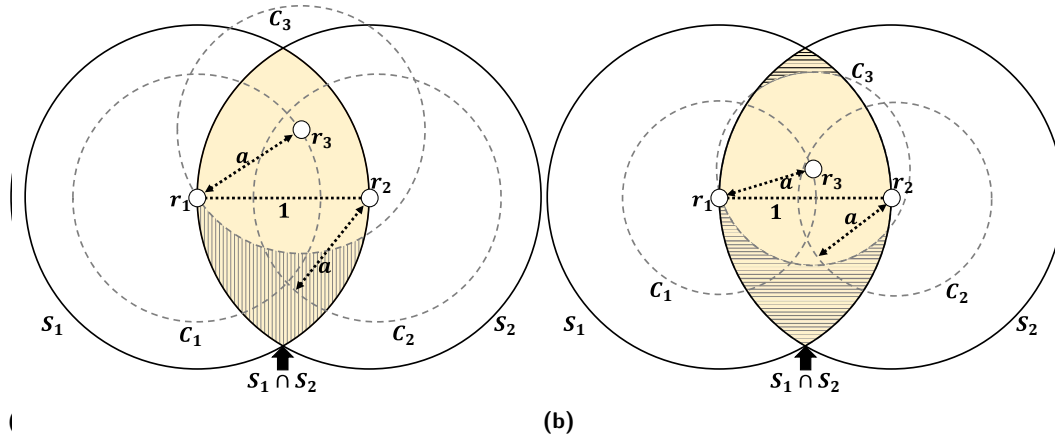
■ **Figure 5** Two isosceles triangles.

In Figure 5, we consider the lengths of two diagonal lines, $\overline{r_1r_3}$ and $\overline{r_2r_4}$. By the assumption, robot r_1 cannot observe r_2 , therefore, $\overline{r_1r_3} \leq 1$ holds because robot r_1 observes r_3 . As the same reason, $\overline{r_2r_4} \leq 1$ also holds. However, both $\overline{r_1r_3} \leq 1$ and $\overline{r_2r_4} \leq 1$ cannot hold in this rhombus, therefore, there is no case as Figure 4(a) and the lemma holds. ◀

By Lemma 11, if all robots have different views, we have two disjoint pairs of robots such that robots in each pair cannot observe each other as in Figure 4(b). Now we discuss the location relations among the robots in this case by the following lemma.

► **Lemma 12.** *If all robots have different views in the distance-based (4,2)-defected model, each of two robots which cannot be observed each other are diagonally located on the formed convex quadrilateral.*

Proof. We already proved that the robots form a convex quadrilateral if all robots have



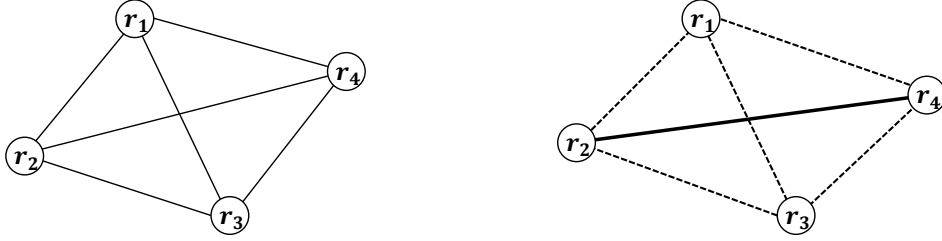
■ **Figure 6** Possible positions of robots r_3 and r_4 .

Figure 6(a) illustrates two circles, called S_1 and S_2 , with radius 1 centered at r_1 and r_2 respectively. Consider the position of robot r_3 : robot r_3 should be located in area $S_1 \cap S_2$, because both r_1 and r_2 observe r_3 (remind that r_1 and r_2 do not observe each other). Locate r_3 in an arbitrary point in area $S_1 \cap S_2$. Let $a = \max(|\overline{r_1r_3}|, |\overline{r_2r_3}|)$, here we assume a is the length of $\overline{r_1r_3}$ without loss of generality. Circles C_1 , C_2 , and C_3 present the circles with radius a centered at r_1 , r_2 , and r_3 respectively. By Lemma 11, robots r_3 and r_4 cannot

14:12 Gathering of Mobile Robots with Defected Views

observe each other, thus $|\overline{r_3 r_4}| \geq a$ holds; robot r_4 should be located outside of C_3 . As a result, robot r_4 should be located in $(C_1 \cap C_2) - C_3$ which is presented as the shaded area in Figure 6. In this case, robots r_1 and r_2 (resp. r_3 and r_4) are diagonally located on a convex quadrilateral, which is a contradiction.

We can consider another case where the shaded area appears on the same side as r_3 (with respect to $\overline{r_1 r_2}$) if a is short enough as Figure 6(b). However, if robot r_4 is located on the same side as r_3 , then robot r_3 is inside the triangle $\triangle r_1 r_2 r_3$. This implies that four robots form a concave quadrilateral, which is a contradiction. ◀



■ **Figure 8** Configuration with one longest line.

Now we show that even when all single robots have different views, two or more robots move to the same point by Algorithm 2. We consider the 6 lines derived by the combination of 4 robots (refer to Figure 7). We focus on the lengths of these 6 lines, and the following corollary holds by Lemma 12.

► **Corollary 13.** *Consider the 6 lines connecting distinct pairs of two robots. If all robots are single and have different views, there is no (side) line which is longer than any diagonal line.*

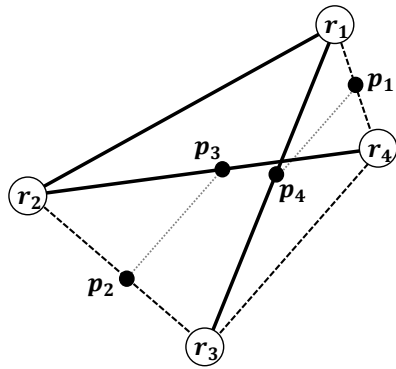
It is worthwhile to mention that there can be at most 4 longest lines among 6 lines. We focus on the number of longest lines and show that the algorithm works correctly in all cases. By Corollary 13, if there exist one or two longest lines, they are diagonal lines. The following lemma holds.

► **Lemma 14.** *Assume that all robots are single and have different views in the distance-based (4,2)-defected model, and consider the 6 lines connecting distinct pairs of two robots. If there exist one or two longest lines, two or more robots become accompanied in one round.*

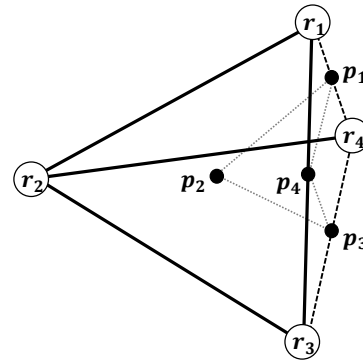
Proof. Figure 8 illustrates an example configuration including the only one longest line (as a diagonal line), where the thick solid line represents the unique longest line. Without loss of generality, we assume that line $\overline{r_2 r_4}$ is the longest one. From the assumption, r_1 and r_3 do not observe each other: r_1 observes triangle $\triangle r_1 r_2 r_4$, and r_3 observes triangle $\triangle r_2 r_3 r_4$. These two triangles are not equilateral triangles because line $\overline{r_2 r_4}$ is the unique longest line. Therefore, robots r_1 and r_3 move to the midpoint of line $\overline{r_2 r_4}$ (by line 9 or 11). If there are two longest lines, the both lines are diagonal lines by Corollary 13 ($\overline{r_1 r_3}$ and $\overline{r_2 r_4}$ in Figure 8). However, this does not affect to the actions of robots r_1 and r_4 ; they move to the midpoint of line $\overline{r_2 r_4}$. Thus the lemma holds. ◀

Now we consider the case that there is a side line whose length is the same as two diagonal lines; there are three or four longest lines.

► **Lemma 15.** *Assume that all robots are single and have different views in the distance-based (4,2)-defected model, and consider the 6 lines connecting distinct pairs of two robots. If there are the three longest lines, two or more robots become accompanied in two rounds.*



■ **Figure 9** Case with three longest lines.



■ **Figure 10** Case with four longest lines.

Proof. Figure 9 illustrates the only configuration including three longest lines. Three thick solid lines are the three longest lines. Remind that robots r_1 and r_3 (or r_2 and r_4) cannot observe each other. By Algorithm 2, all robots move to the different points: robot r_1 (resp. r_2) moves to the midpoint p_1 (resp. p_2) of line $\overline{r_1 r_4}$ (resp. $\overline{r_2 r_3}$) since r_1 (resp. r_2) observes an isosceles triangle $\triangle r_1 r_2 r_4$ (resp. $\triangle r_1 r_2 r_3$). Robot r_3 (resp. r_4) moves to the midpoint p_3 (resp. p_4) of line $\overline{r_2 r_4}$ (resp. $\overline{r_1 r_3}$) that is the longest line of the observed triangle $\triangle r_2 r_3 r_4$ (resp. $\triangle r_1 r_3 r_4$). In this case, triangles $\triangle r_2 r_3 r_4$ and $\triangle r_2 p_2 p_3$ are similar, the length of line $\overline{p_2 p_3}$ is half of the length of line $\overline{r_3 r_4}$, and line $\overline{p_2 p_3}$ and line $\overline{r_3 r_4}$ are parallel. Through the same argument for lines $\overline{p_1 p_4}$ and $\overline{r_4 r_3}$, we can show that the lengths of lines $\overline{p_1 p_4}$ and $\overline{p_2 p_3}$ are the same and these two lines are parallel. This means that the quadrilateral formed in the next round is a parallelogram: even if all robots have different views in this configuration, two or more robots become accompanied in the next round because diagonal line $\overline{p_1 p_2}$ is the unique longest line (by Lemma 14). ◀

► **Lemma 16.** *Assume that all robots are single and have different views in the distance-based $(4, 2)$ -defected model, and consider the 6 lines connecting distinct pairs of two robots. If there are four longest lines, two or more robots become accompanied in two rounds.*

Proof. Figure 10 illustrates the only possible configuration including four longest lines. Four thick solid lines are the four longest lines. By Algorithm 2, all robots move to the different points: robot r_1 (resp. r_3) moves to the midpoint p_1 (resp. p_3) of line $\overline{r_1 r_4}$ (resp. $\overline{r_3 r_4}$) since r_1 (resp. r_3) observes an isosceles triangle $\triangle r_1 r_2 r_4$ (resp. $\triangle r_2 r_3 r_4$). Robot r_2 moves to the center p_2 of the equilateral triangle $\triangle r_1 r_2 r_3$ it observes, and r_4 moves to the midpoint p_4 of the unique longest line $\overline{r_1 r_3}$ it observes (note that if $|\overline{r_1 r_4}| = |\overline{r_3 r_4}|$, triangle $\triangle r_1 r_3 r_4$ is an isosceles triangle, however robot r_4 moves to the midpoint p_4 of the base line also in this case). As a result, the four points, from p_1 to p_4 , form a concave quadrilateral. Hence, two or more robots become accompanied in the next round by Lemma 10. ◀

From Lemmas 4, 9, 14, 15 and 16, the following theorem holds.

► **Theorem 17.** *In the distance-based $(4, 2)$ -defected model, Algorithm 2 solves the gathering problem in at most four rounds.* ◀

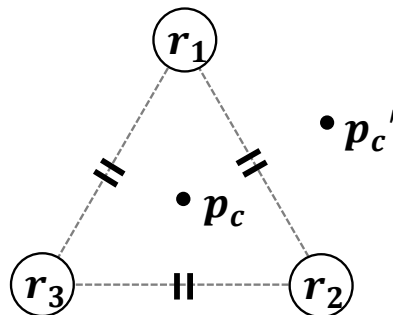
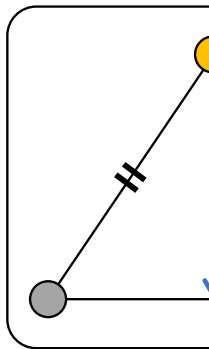
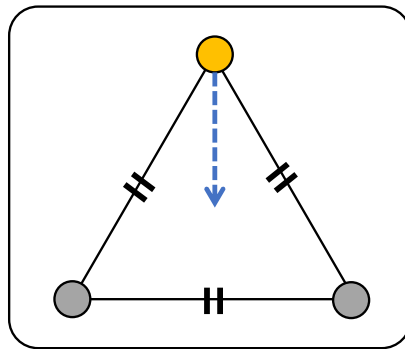
5 II

In this s
view mc
model, ε
defected

5.1 I

By the t
achieved
for $N \geq$
to find ε
(3,1)-def

► **Theo**
distance



■ **Figure 11** Example for an unsolvable configuration in the distance-based (3,1)-defected model.

Proof. We prove this theorem by showing that there is no (deterministic) algorithm even in the distance-based (3,1)-defected model. Note that the distance-based (3,1)-defected model is stronger than the adversarial one, this result implies that the gathering is also unsolvable in the adversarial one. Assume that three robots, $R = \{r_1, r_2, r_3\}$, are arranged in an equilateral triangle as Figure 11, and robot r_1 (resp. r_2 and r_3) observes r_2 (resp. r_3 and r_1). All robots do not agree on any geometrical agreement (e.g., direction, orientation, chirality, or unit distance), thus we can assume that every robot r_i considers the direction from itself to the center of the triangle (p_c) (i.e., $\overrightarrow{r_i p_c}$) as the positive direction of X -axis in its local coordinate system. Moreover, we also assume that all robots have the same chirality (e.g., clockwise) and the same unit distance. This means that all robots obtain the exactly same view from of *Look* operation.

Let \mathcal{A} be an algorithm for gathering in the distance-based (3,1)-defected model. In the above configuration, all robots have the same views, thus they execute the same behavior according to \mathcal{A} (i.e., all robots move to the same x and y coordinates in their local coordinate systems). This causes another configuration forming a different equilateral triangle, which shows by repeating the argument that the robots cannot gather at the same point forever. The only way to prevent the robots from forming another equilateral triangle is to move to point p_c , i.e., each robot moves to the point located at $|\overrightarrow{r_i r_j}|/\sqrt{3}$ distance in the 30°

clockwise direction of the observed robot r_j . However, if all robots agree on the opposite direction of chirality (counter-clockwise in this case), they move to the outside of triangle $\triangle r_1 r_2 r_3$ (i.e., robot r_1 moves to point p'_c instead of p_c). As a result, the robots form another equilateral triangle. ◀

5.2 Impossibility in the relaxed adversarial $(N, N - 2)$ -defected model

The (N, k) -defected model assumes that k robots observed by robot r are chosen from the robots that are located at points other than r 's current position and that r can detect whether it is single or accompanied. Natural relaxation of the model is to choose the k robots other than r (i.e., robots at r 's current position can be chosen) and assume the weak multiplicity detection for the k robots and r itself. We call the model with the relaxation the *relaxed adversarial* (N, k) -defected model. Notice that the key property of the $(N, N - 2)$ -defected model such that any accompanied robot can observe all the robots does not hold in the relaxed model.

The following theorem shows that the gathering is impossible (from some configuration) in the relaxed adversarial $(N, N - 2)$ -defected model.

► **Theorem 19.** *There is no (deterministic) algorithm to solve the gathering problem in the relaxed adversarial $(N, N - 2)$ -defected model.*

Proof. Let \mathcal{A} be a gathering algorithm in the relaxed adversarial $(N, N - 2)$ -defected model. We consider only initial configurations where all robots are located at two points p_1 and p_2 .

First, consider the initial configuration where $N - 1$ robots are located at p_1 and one robot, say r_1 , is located at p_2 . When the robots at p_1 do not observe r_1 , they misunderstand that the gathering is already achieved and terminate. To achieve the gathering, r_1 has to move to p_1 . This implies that \mathcal{A} has the following action (**Action 1**): when a single robot r observes only one occupied point other than r 's current point and recognizes that the point is occupied by multiple robots, r has to move to the point.

Notice that **Action 1** is sufficient to show that \mathcal{A} cannot solve the gathering in the relaxed adversarial $(4, 2)$ -defected model. Consider the initial configuration where two robots exist at both of p_1 and p_2 (four robots in total). When the robots at p_1 (resp. p_2) observe only the two robots at p_2 (resp. p_1), the robots at p_1 (resp. p_2) move to p_2 (resp. p_1) by **Action 1**. At the resultant configuration, two robots exist at both of p_1 and p_2 , which shows by repeating the argument that algorithm \mathcal{A} cannot solve the gathering problem.

Second, consider the initial configuration where $N \geq 5$ and all robots recognize that both p_1 and p_2 are occupied by multiple robots, which can occur when a point is occupied by three or more robots and the other is occupied by two or more robots. When all the robots at the same point observe the same set of robots (but still they recognize that both the points are occupied by multiple robots), the robots at the same point execute the same action (i.e., move to the same point). Since algorithm \mathcal{A} solves the gathering problem, all robots eventually have to move to the same point (precisely the midpoint of the two points occupied by robots) to achieve the gathering. This implies that \mathcal{A} has the following action (**Action 2**): when an accompanied robot r observes only one occupied point other than r 's current point and recognizes that the point is occupied by multiple robots, r has to move to the midpoint of the two points.

Finally, consider the initial configuration of $N (\geq 5)$ robots where two robots exist at p_1 and $N - 2$ robots exist at p_2 . When each robot r_1 at p_1 observes only $N - 2$ robots at p_2 (and recognizes itself as a single robot), r_1 moves to p_2 by **Action 1**. On the other hand, when each robot r_2 at p_2 observes the two robots at p_1 and $N - 4$ robots (other than r_2)

at p_2 , r_2 moves to the midpoint of p_1 and p_2 by **Action 2**. At the resultant configuration, two robots exist at p_2 and $N - 2$ robots exist at the midpoint of p_1 and p_2 . By repeating the argument, we can show that algorithm \mathcal{A} cannot solve the gathering problem although all robots converge at the same point (i.e., the distance between the two groups of robots becomes smaller and smaller but does not become zero).

Consequently, there is no gathering algorithm in the relaxed adversarial $(N, N - 2)$ -defected model. ◀

6 Conclusion and Open Problems

In this paper, we introduced a new computational model, the (N, k) -defected model, where each robot cannot necessarily observe all other robots: i.e., each robot observes at most k other robots not located at its current position (where $k < N - 1$). We addressed the gathering problem, which is one of the basic problem in autonomous mobile robot systems, in the $(N, N - 2)$ -defected model. We proposed two gathering algorithms: (1) an algorithm in the adversarial $(N, N - 2)$ -defected model that achieves the gathering within three rounds, and (2) an algorithm in the distance-based $(4, 2)$ -defected model that achieves the gathering within four rounds. Moreover, we showed that there is no (deterministic) algorithm in either the adversarial or distance-based $(3, 1)$ -defected model. In the proposed model, we assume that each robot r observes k other robots among the robots located at the different points than the point occupied by r itself. The relaxation of this assumption, where k robots are chosen among all other robots other than r , can be considered, however, we proved that the gathering is unsolvable in this relaxed model.

The remaining problem we are most interested in is to clarify the solvability of the gathering problem in the adversarial $(4, 2)$ -defected model. Remind that the basic strategy of the proposed algorithm in the distance-based $(4, 2)$ -defected model is to determine one unique point from the triangle formed by the observed set of points. We call the algorithm using this strategy the *set-based algorithm*, where each robot determines the destination referring to only the set of observed points: for example, when a robot observes an isosceles triangle, it always moves to the midpoint of the base, regardless of whether it is adjacent to the base or not, i.e., we do not use the information of the (relative) position of the observing robot. It can be easily proved that *there is no (deterministic) set-based algorithm to solve the gathering problem in adversarial $(4, 2)$ -defected model*. This means that if a gathering algorithm exists in the adversarial $(4, 2)$ -defected model, each robot has to use its relative position in the set of observed points, e.g., when a robot observes an isosceles triangle, the destination point changes depending on whether the robot is at a point incident to the base of the triangle or not.

An important future work is to find the minimum k that allows a solution for the gathering problem in the adversarial or distance-based (N, k) -defected model. In this paper, we considered only the gathering problem, therefore, to challenge other problems under the (N, k) -defected model is another future work.

References

- 1 Hideki Ando, Yoshinobu Oasa, Ichiro Suzuki, and Masafumi Yamashita. Distributed memory-less point convergence algorithm for mobile robots with limited visibility. *IEEE Trans. Robotics Autom.*, 15(5):818–828, 1999. doi:10.1109/70.795787.
- 2 Zohir Bouzid, Shantanu Das, and Sébastien Tixeuil. Gathering of mobile robots tolerating multiple crash faults. In *IEEE 33rd International Conference on Distributed Computing Systems, ICDCS*, pages 337–346. IEEE Computer Society, 2013. doi:10.1109/ICDCS.2013.27.

- 3 Avik Chatterjee, Sruti Gan Chaudhuri, and Krishnendu Mukhopadhyaya. Gathering asynchronous swarm robots under nonuniform limited visibility. In *Distributed Computing and Internet Technology – 11th International Conference, ICDCIT*, volume 8956 of *Lecture Notes in Computer Science*, pages 174–180. Springer, 2015. doi:10.1007/978-3-319-14977-6_11.
- 4 Mark Cieliebak, Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro. Solving the Robots Gathering Problem. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming, ICALP*, pages 1181–1196. Springer, 2003. doi:10.1007/3-540-45061-0_90.
- 5 Gianlorenzo D’Angelo, Gabriele Di Stefano, Ralf Klasing, and Alfredo Navarra. Gathering of robots on anonymous grids and trees without multiplicity detection. *Theor. Comput. Sci.*, 610:158–168, 2016. doi:10.1016/j.tcs.2014.06.045.
- 6 Xavier Défago, Maria Gradinariu, Stéphane Messika, and Philippe Raipin Parvédy. Fault-Tolerant and Self-stabilizing Mobile Robots Gathering. In *Proceedings of the 20th International Symposium on Distributed Computing, DISC*, pages 46–60. Springer, 2006. doi:10.1007/11864219_4.
- 7 Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Peter Widmayer. Gathering of Asynchronous Oblivious Robots with Limited Visibility. In *Proceedings of the 18th Annual Symposium on Theoretical Aspects of Computer Science, STACS*, pages 247–258. Springer, 2001. doi:10.1007/3-540-44693-1_22.
- 8 Adam Heriban and Sébastien Tixeuil. Mobile robots with uncertain visibility sensors: Possibility results and lower bounds. *Parallel Process. Lett.*, 31(1):2150002:1–2150002:21, 2021. doi:10.1142/S012962642150002X.
- 9 Nobuhiro Inuzuka, Yuichi Tomida, Taisuke Izumi, Yoshiaki Katayama, and Koichi Wada. Gathering Problem of Two Asynchronous Mobile Robots with Semi-dynamic Compasses. In *Proceedings of the 15th International Colloquium on Structural Information and Communication Complexity, SIROCCO*, pages 5–19. Springer, 2008. doi:10.1007/978-3-540-69355-0_3.
- 10 Taisuke Izumi, Yoshiaki Katayama, Nobuhiro Inuzuka, and Koichi Wada. Gathering Autonomous Mobile Robots with Dynamic Compasses: An Optimal Result. In *Proceedings of the 21st International Symposium on Distributed Computing, DISC*, pages 298–312. Springer, 2007. doi:10.1007/978-3-540-75142-7_24.
- 11 Yoshiaki Katayama, Yuichi Tomida, Hiroyuki Imazu, Nobuhiro Inuzuka, and Koichi Wada. Dynamic Compass Models and Gathering Algorithms for Autonomous Mobile Robots. In *Proceedings of the 14th International Colloquium on Structural Information and Communication Complexity, SIROCCO*, pages 274–288. Springer, 2007. doi:10.1007/978-3-540-72951-8_22.
- 12 David G. Kirkpatrick, Irina Kostitsyna, Alfredo Navarra, Giuseppe Prencipe, and Nicola Santoro. Separating bounded and unbounded asynchrony for autonomous robots: Point convergence with limited visibility. In *PODC ’21: ACM Symposium on Principles of Distributed Computing*, pages 9–19. ACM, 2021. doi:10.1145/3465084.3467910.
- 13 Ralf Klasing, Euripides Markou, and Andrzej Pelc. Gathering asynchronous oblivious mobile robots in a ring. *Theor. Comput. Sci.*, 390(1):27–39, 2008. doi:10.1016/j.tcs.2007.09.032.
- 14 Giuseppe Antonio Di Luna, Ryuhei Uehara, Giovanni Viglietta, and Yukiko Yamauchi. Gathering on a circle with limited visibility by anonymous oblivious robots. In *34th International Symposium on Distributed Computing, DISC*, volume 179 of *LIPICs*, pages 12:1–12:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICs.DISC.2020.12.
- 15 Debasish Pattanayak, Kaushik Mondal, H. Ramesh, and Partha Sarathi Mandal. Fault-tolerant gathering of mobile robots with weak multiplicity detection. In *Proceedings of the 18th International Conference on Distributed Computing and Networking*, page 7. ACM, 2017. URL: <http://dl.acm.org/citation.cfm?id=3007786>.
- 16 Giuseppe Prencipe. On the feasibility of gathering by autonomous mobile robots. In *Proceedings of the 12th International Colloquium on Structural Information and Communication Complexity, SIROCCO*, pages 246–261. Springer, 2005. doi:10.1007/11429647_20.

14:18 Gathering of Mobile Robots with Defected Views

- 17 Giuseppe Prencipe. Impossibility of gathering by a set of autonomous mobile robots. *Theor. Comput. Sci.*, 384(2-3):222–231, 2007. doi:10.1016/j.tcs.2007.04.023.
- 18 Giuseppe Prencipe. Autonomous Mobile Robots: A Distributed Computing Perspective. In *Proceedings of the 9th International Symposium on Algorithms and Experiments for Sensor Systems, Wireless Networks and Distributed Robotics, ALGOSENSORS*, pages 6–21. Springer, 2013. doi:10.1007/978-3-642-45346-5_2.
- 19 Giuseppe Prencipe and Nicola Santoro. Distributed Algorithms for Autonomous Mobile Robots. In *Proceedings of Fourth IFIP International Conference on Theoretical Computer Science, TCS*, pages 47–62. Springer, 2006. doi:10.1007/978-0-387-34735-6_8.
- 20 Samia Souissi, Xavier Défago, and Masafumi Yamashita. Gathering Asynchronous Mobile Robots with Inaccurate Compasses. In *Proceedings of the 10th International Conference on Principles of Distributed Systems, OPODIS*, pages 333–349. Springer, 2006. doi:10.1007/11945529_24.
- 21 Ichiro Suzuki and Masafumi Yamashita. Distributed Anonymous Mobile Robots: Formation of Geometric Patterns. *SIAM J. Comput.*, 28(4):1347–1363, 1999. doi:10.1137/S009753979628292X.

A Unifying Approach to Efficient (Near)-Gathering of Disoriented Robots with Limited Visibility

Jannik Castenow ✉ 

Heinz Nixdorf Institute & Computer Science Department, Paderborn University, Germany

Jonas Harbig ✉ 

Heinz Nixdorf Institute & Computer Science Department, Paderborn University, Germany

Daniel Jung ✉ 

Heinz Nixdorf Institute & Computer Science Department, Paderborn University, Germany

Peter Kling ✉ 

Department of Informatics, Universität Hamburg, Germany

Till Knollmann ✉ 

Heinz Nixdorf Institute & Computer Science Department, Paderborn University, Germany

Friedhelm Meyer auf der Heide ✉

Heinz Nixdorf Institute & Computer Science Department, Paderborn University, Germany

Abstract

We consider a swarm of n robots in a d -dimensional Euclidean space. The robots are oblivious (no persistent memory), disoriented (no common coordinate system/compass), and have limited visibility (observe other robots up to a constant distance). The basic formation task GATHERING requires that all robots reach the same, not predefined position. In the related NEAR-GATHERING task, they must reach distinct positions in close proximity such that every robot sees the entire swarm. In the considered setting, GATHERING can be solved in $\mathcal{O}(n + \Delta^2)$ synchronous rounds both in two and three dimensions, where Δ denotes the initial maximal distance of two robots [3, 13, 24].

In this work, we formalize a key property of efficient GATHERING protocols and use it to define λ -contracting protocols. Any such protocol gathers n robots in the d -dimensional space in $\mathcal{O}(\Delta^2)$ synchronous rounds, for $d \geq 2$. For $d = 1$, any λ -contracting protocol gathers in optimal time $\mathcal{O}(\Delta)$. Moreover, we prove a corresponding lower bound stating that any protocol in which robots move to target points inside the local convex hulls of their neighborhoods – λ -contracting protocols have this property – requires $\Omega(\Delta^2)$ rounds to gather all robots ($d > 1$). Among others, we prove that the d -dimensional generalization of the GTC-protocol [3] is λ -contracting. Remarkably, our improved and generalized runtime bound is independent of n and d .

We also introduce an approach to make any λ -contracting protocol collision-free (robots never occupy the same position) to solve NEAR-GATHERING. The resulting protocols maintain the runtime of $\Theta(\Delta^2)$ and work even in the semi-synchronous model. This yields the first NEAR-GATHERING protocols for disoriented robots and the first proven runtime bound. In particular, combined with results from [28] for robots with global visibility, we obtain the first protocol to solve UNIFORM CIRCLE FORMATION (arrange the robots on the vertices of a regular n -gon) for oblivious, disoriented robots with limited visibility.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases mobile robots, gathering, limited visibility, runtime, collision avoidance, near-gathering

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2022.15

Related Version *Full Version:* <https://arxiv.org/abs/2206.07567> [14]

Funding This work was partially supported by the German Research Foundation (DFG) under the project number ME 872/14-1.



© Jannik Castenow, Jonas Harbig, Daniel Jung, Peter Kling, Till Knollmann, and Friedhelm Meyer auf der Heide;

licensed under Creative Commons License CC-BY 4.0

26th International Conference on Principles of Distributed Systems (OPODIS 2022).

Editors: Eshcar Hillel, Roberto Palmieri, and Etienne Rivière; Article No. 15; pp. 15:1–15:25

Leibniz International Proceedings in Informatics



Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Envision a huge swarm of n robots spread in a d -dimensional Euclidean space that must solve a *formation task* like GATHERING (moving all robots to a single, not pre-determined point) or UNIFORM-CIRCLE (distributing the robots over the vertices of a regular n -gon). Whether and how efficiently a given task is solvable varies largely with the robots' capabilities (local vs. global visibility, memory vs. memory-less, communication capabilities, common orientation vs. disorientation). While classical results study what capabilities the robots need *at least* to solve a given task, our focus lies on *how fast* a given formation task can be solved assuming simple robots. Specifically, we consider the GATHERING problem and the related NEAR-GATHERING problem for oblivious, disoriented robots with a limited viewing range.

GATHERING is the most basic formation task and a standard benchmark to compare robot models [27]. The robots must gather at the same, not predefined, position. Whether or not GATHERING is solvable depends on various robot capabilities. It is easy to see that robots can solve GATHERING in case they have unlimited visibility (can observe all other robots) and operate fully synchronously [17]. However, as soon as the robots operate asynchronously, have only limited visibility, or do not agree on common coordinate systems, the problem gets much harder or even impossible to solve (see Section 1.1 for a comprehensive discussion). A well-known protocol to solve GATHERING of robots with limited visibility is the GO-TO-THE-CENTER (GTC) protocol that moves each robot towards the center of the smallest enclosing circle of all observable robots [3]. GTC gathers all robots in $\mathcal{O}(n + \Delta^2)$ synchronous rounds, where the *diameter* Δ denotes the initial maximal distance of two robots [24]. The term n upper bounds the number of rounds in which robots *collide* (move to the same position), while Δ^2 results from how quickly the global smallest enclosing circle shrinks. Hence, GTC not only forces the robots to collide in the final configuration but also incurs several collisions during GATHERING. Such collisions are fine for point robots in theoretical models but a serious problem for physical robots that cannot occupy the same position. This leads us to NEAR-GATHERING, which requires the robots to move *collision-free* to *distinct* locations such that every robot can observe the entire swarm despite its limited visibility [42]. Requiring additionally that, eventually, robots *simultaneously* (within one round/epoch) *terminate*, turns NEAR-GATHERING into a powerful subroutine for more complex formation tasks like UNIFORM CIRCLE. Once all robots see the entire swarm and are simultaneously aware of that, they can switch to the protocol of [28] to build a uniform circle. Although that protocol is designed for robots with a global view, we can use it here since solving NEAR-GATHERING grants the robots *de facto* a global view. Note the importance of *simultaneous* termination, as otherwise, some robots might build the new formation while others are still gathering, possibly disconnecting some robots from the swarm.

Robot Model. We assume the standard *OBLOT* model [27] for oblivious, point-shaped robots in \mathbb{R}^d . The robots are *anonymous* (no identifiers), *homogeneous* (all robots execute the same protocol), *identical* (same appearance), *autonomous* (no central control) and *deterministic*. Moreover, we consider *disoriented* robots with *limited visibility*. Disorientation means that a robot observes itself at the origin of its local coordinate system, which can be arbitrarily rotated and inverted compared to other robots. The disorientation is *variable*, i.e., the local coordinate system might differ from round to round. Limited visibility implies that each robot can observe other robots only up to a constant distance. The robots do not have *multiplicity detection*, i.e., robots observe only a single robot in case multiple robots are located at the same position. Furthermore, time is divided into discrete LCM-cycles (*rounds*)

consisting of the operations LOOK, COMPUTE and MOVE. During its LOOK operation, a robot takes a snapshot of all visible robots, which is used in the following COMPUTE operation to compute a *target point*, to which the robot moves in the MOVE operation. Moves are *rigid* (a robot always reaches its target point) and depend solely on observations from the last LOOK operation (robots are oblivious). The time model can be fully synchronous ($\mathcal{F}\text{SYNC}$; all robots are active each round and operations are executed synchronously), semi-synchronous ($\mathcal{S}\text{SYNC}$; a subset of robots is active each round and operations are executed synchronously), or completely asynchronous ($\mathcal{A}\text{SYNC}$). The $\mathcal{S}\text{SYNC}$ and $\mathcal{A}\text{SYNC}$ schedules of the robots are *fair*, i.e., each robot is activated infinitely often. Time is measured in rounds in $\mathcal{F}\text{SYNC}$ and *epochs* (the smallest number of rounds such that all robots finish one LCM-cycle) in $\mathcal{S}\text{SYNC}$ or $\mathcal{A}\text{SYNC}$.

Results in a Nutshell. For GATHERING of oblivious, disoriented robots with limited visibility in \mathbb{R}^d , we introduce the class of λ -contracting protocols for a constant $\lambda \in (0, 1]$. For instance, the well-known GTC [3] and several other GATHERING protocols are λ -contracting. We prove that for $d > 1$, every λ -contracting protocol gathers a swarm of diameter Δ in $\mathcal{O}(\Delta^2)$ rounds. The case $d = 1$ leads to an optimal time bound of $\Theta(\Delta)$ rounds. We also prove a matching lower bound for any protocol in which robots always move to points inside the convex hull of their neighbors, including themselves. While our results for GATHERING assume the $\mathcal{F}\text{SYNC}$ scheduler¹, for NEAR-GATHERING we also consider $\mathcal{S}\text{SYNC}$. We show how to transform any λ -contracting protocol into a *collision-free* λ -contracting protocol to solve NEAR-GATHERING while maintaining a runtime of $\mathcal{O}(\Delta^2)$.

1.1 Related Work

One important topic of the research area of distributed computing by mobile robots is *pattern formation* problems, i.e., the question of which patterns can be formed by a swarm of robots and which capabilities are required. For instance, the ARBITRARY PATTERN FORMATION problem requires the robots to form an arbitrary pattern specified in the input [21, 25, 30, 46, 47, 48]. The patterns *point* and *uniform circle* play an important role since these are the only two patterns that can be formed starting from *any* input configuration due to their high symmetry [46]. In the following, we focus on the pattern *point*, more precisely on the GATHERING, CONVERGENCE and NEAR-GATHERING problems. While GATHERING requires that all robots move to a single (not predefined) point in finite time, CONVERGENCE demands that for all $\varepsilon > 0$, there is a point in time such that the maximum distance of any pair of robots is at most ε and this property is maintained (the robots *converge* to a single point). NEAR-GATHERING is closely related to the CONVERGENCE problem by robots with limited visibility. Instead of converging to a single point, NEAR-GATHERING is solved as soon as all robots are located at *distinct* locations within a small area. For a more comprehensive overview of other patterns and models, we refer to [26].

Possibilities & Impossibilities. In the context of robots with *unlimited visibility*, GATHERING can be solved under the $\mathcal{F}\text{SYNC}$ scheduler by disoriented and oblivious robots without multiplicity detection [17]. Under the same assumptions, GATHERING is impossible under the $\mathcal{S}\text{SYNC}$ and $\mathcal{A}\text{SYNC}$ schedulers [45]. Multiplicity detection plays a crucial role: at least 3 disoriented robots with multiplicity detection can be gathered in $\mathcal{A}\text{SYNC}$ (and thus also

¹ With the considered robot capabilities, GATHERING is impossible in $\mathcal{S}\text{SYNC}$ or $\mathcal{A}\text{SYNC}$ [45].

$\mathcal{S}\text{SYNC}$) [16]. The case of 2 robots remains impossible [46]. Besides multiplicity detection, an agreement on one axis of the local coordinate systems also allows the robots to solve GATHERING in $\mathcal{A}\text{SYNC}$ [5]. CONVERGENCE requires less assumptions than GATHERING. No multiplicity detection is needed for the $\mathcal{A}\text{SYNC}$ scheduler [17].

Under the assumption of *limited visibility*, disoriented robots without multiplicity detection can be gathered in $\mathcal{F}\text{SYNC}$ [3] with the GTC protocol that moves every robot towards the center of the smallest circle enclosing its neighborhood. GTC has also been generalized to three dimensions [13]. In $\mathcal{A}\text{SYNC}$, current solutions require more capabilities: GATHERING can be achieved by robots with limited visibility that agree additionally on the axes and orientation of their local coordinate systems [29]. It is open whether fewer assumptions are sufficient to solve GATHERING of robots with limited visibility in $\mathcal{S}\text{SYNC}$ or $\mathcal{A}\text{SYNC}$. In $\mathcal{S}\text{SYNC}$, CONVERGENCE can be solved even by disoriented robots with limited visibility without multiplicity detection [3]. However, similar to GATHERING, it is still open whether disoriented robots with limited visibility can solve CONVERGENCE under the $\mathcal{A}\text{SYNC}$ scheduler. Recently, it could be shown that multiplicity detection suffices to solve CONVERGENCE under the more restricted k - $\mathcal{A}\text{SYNC}$ scheduler. The constant k bounds how often other robots can be activated within one LCM cycle of a single robot [36, 37].

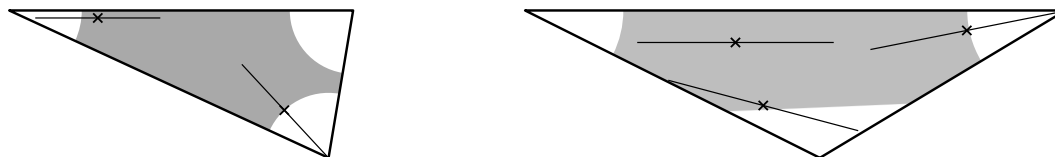
The NEAR-GATHERING problem has been introduced in [41, 42] together with a protocol to solve NEAR-GATHERING by robots with limited visibility and agreement on one axis of their local coordinate systems under the $\mathcal{A}\text{SYNC}$ scheduler. An important tool to prevent collisions is a well-connected initial configuration, i.e., the initial configuration is connected concerning the *connectivity range* which is by an additive constant smaller than the viewing range [41, 42]. In earlier work, NEAR-GATHERING has been used as a subroutine to solve ARBITRARY PATTERN FORMATION by robots with limited visibility [49]. The solution, however, uses infinite persistent memory for each robot. Further research directions study GATHERING and CONVERGENCE under crash faults or Byzantine faults [2, 4, 6, 7, 8, 9, 10, 11, 22, 32, 43] or inaccurate measurement and movement sensors of the robots [18, 31, 33, 37].

Runtime. Considering disoriented robots with *unlimited* visibility, it is known that CONVERGENCE can be solved in $\mathcal{O}(n \cdot \log \Delta/\varepsilon)$ epochs under the $\mathcal{A}\text{SYNC}$ scheduler, where the diameter Δ denotes the initial maximum distance of two robots [19] (initially a bound of $\mathcal{O}(n^2 \cdot \log \Delta/\varepsilon)$ has been proven in [17]). When considering disoriented robots with *limited* visibility and the $\mathcal{F}\text{SYNC}$ scheduler, the GTC protocol solves GATHERING both in two and three dimensions in $\Theta(n + \Delta^2)$ rounds [13, 24]. It is conjectured that the runtime is optimal in worst-case instances, where $\Delta \in \Omega(n)$ [13, 15]. There is some work achieving a faster runtime for slightly different models: robots on a grid in combination with the \mathcal{LUMI} model (constant sized local communication via lights) [1, 20], predefined neighborhoods in a closed chain [1, 15] or agreement on one axis of the local coordinate systems [44]. Also, a different time model – the *continuous* time model, where the movement of robots is defined for each *real* point in time by a bounded velocity vector – leads to a faster runtime: There are protocols with a runtime of $\mathcal{O}(n)$ [12, 23]. In [38], a more general class of continuous protocols has been introduced, the *contracting* protocols. Contracting protocols demand that each robot part of the global convex hull of all robots' positions moves at full speed towards the inside. Any contracting protocol gathers all robots in time $\mathcal{O}(n \cdot \Delta)$. One such protocol also needs a runtime of $\Omega(n \cdot \Delta)$ in a specific configuration. For instance, the continuous variant of GTC is contracting [38] but also the protocols of [12, 23]. The class of contracting protocols also generalizes to three dimensions with an upper time bound of $\mathcal{O}(n^{3/2} \cdot \Delta)$ [13].

1.2 Our Contribution & Outline

In the following, we provide a detailed discussion of our results and put them into context concerning the related results discussed in Section 1.1. Our results assume robots located in \mathbb{R}^d and the *OBLLOT* model for deterministic, disoriented robots with limited visibility.

Gathering. Our first main contribution is introducing a large class of GATHERING protocols in $\mathcal{F}_{\text{SYNC}}$ that contains several natural protocols such as GTC. We prove that *every* protocol from this class gathers in $\mathcal{O}(\Delta^2)$ ($d > 1$) or $\mathcal{O}(\Delta)$ ($d = 1$) rounds, where the diameter Δ denotes the initial maximal distance between two robots. Note that, the bound of $\mathcal{O}(\Delta^2)$ not only reflects how far a given initial swarm is from a gathering but also improves the GTC bound from $\mathcal{O}(n + \Delta^2)$ to $\mathcal{O}(\Delta^2)$. We call this class λ -contracting protocols. Such protocols restrict the allowed target points to a specific subset of a robot’s local convex hull (formed by the positions of all visible robots, including itself) in the following way. Let $diam$ denote the diameter of a robot’s local convex hull. Then, a target point p is an allowed target point if it is the center of a line segment of length $\lambda \cdot diam$, completely contained in the local convex hull. This guarantees that the target point lies far enough inside the local convex hull (at least along one dimension) to decrease the swarm’s diameter sufficiently. See Figure 1 for an illustration.



■ **Figure 1** Two local convex hulls, each formed by 3 robots. The gray area marks valid target points of λ -contracting protocols. The exemplary line segments all have length $\lambda \cdot diam$, where $diam$ is the diameter of the respecting convex hull. On the left $\lambda = 4/7$, on the right $\lambda = 4/11$.

We believe these λ -contracting protocols encapsulate the core property of fast GATHERING protocols. Their analysis is comparatively clean, simple, and holds for any dimension d . Thus, by proving that (the generalization of) GTC is λ -contracting for arbitrary dimensions, we give the first protocol that provably gathers in $\mathcal{O}(\Delta^2)$ rounds for any dimension. As a strong indicator that our protocol class might be asymptotically optimal, we prove that every GATHERING protocol for deterministic, disoriented robots whose target points lie *always inside* the robots’ local convex hulls requires $\Omega(\Delta^2)$ rounds. Staying in the convex hull of visible robots is a natural property for any known protocol designed for oblivious, disoriented robots with limited visibility. Thus, reaching a sub-quadratic runtime – if at all possible – would require the robots to compute target points outside of their local convex hulls sufficiently often.

Near-Gathering. Our second main contribution proves that any λ -contracting protocol for GATHERING can be transformed into a collision-free protocol that solves NEAR-GATHERING in $\mathcal{O}(\Delta^2)$ rounds ($\mathcal{F}_{\text{SYNC}}$) or epochs ($\mathcal{S}_{\text{SYNC}}$). As in previous work [41, 42], our transformed protocols require that the initial swarm is *well-connected*, i.e., the swarm is connected concerning the *connectivity range* of 1 and the robots have a viewing range of $1 + \tau$, for a constant $\tau > 0$. The adapted protocols ensure that the swarm stays connected concerning the connectivity range.

The well-connectedness serves two purposes. First, it allows a robot to compute its target point under the given λ -contracting protocol and the target points of nearby robots to prevent collisions. Its second purpose is to enable termination: Once there is a robot whose local convex hull has a diameter at most τ , *all* robots must have distance at most τ , as otherwise, the swarm would not be connected concerning the connectivity range 1. Thus, all robots can *simultaneously* decide (in the same round in \mathcal{F} SYNC and within one epoch in \mathcal{S} SYNC) whether NEAR-GATHERING is solved. If the swarm is not well-connected, it is easy to see that such a simultaneous decision is impossible². The simultaneous termination also allows us to derive the first protocol to solve UNIFORM-CIRCLE for disoriented robots with limited visibility. Once the robots' local diameter (and hence also the global diameter) is less than τ , they essentially have a global view. As the UNIFORM CIRCLE protocol from [28] maintains a small diameter, it can be used after the termination of our NEAR-GATHERING protocol without any modification.

Outline. Section 2 introduces various notations. λ -contracting protocols are introduced in Section 3.1. Upper and lower runtime bounds are provided in Section 3.2. The section is concluded with three exemplary λ -contracting protocols, including GTC (Section 3.3). Section 4 discusses the general approach to transform any λ -contracting protocol (in any dimension) into a collision-free protocol to solve NEAR-GATHERING. Finally, the paper is concluded, and future research questions are addressed in Section 5. Due to space constraints, some proofs and additional information are moved to the full version of this paper [14].

2 Notation

We consider a swarm of n robots $R = \{r_1, \dots, r_n\}$ moving in a d -dimensional Euclidean space \mathbb{R}^d . Initially, the robots are located at pairwise distinct locations. We denote by $p_i(t)$ the position of robot r_i in a global coordinate system (not known to the robots) in round t . Robots have a *limited visibility*, i.e., they can observe other robots only up to a constant distance. We distinguish the terms *viewing* range and *connectivity* range and normalize all distances such that the connectivity range is 1. The initial configuration is connected concerning the connectivity range. More formally, $\text{UBG}(t) = (R, E(t))$ the Unit Ball Graph, where $\{r_i, r_j\} \in E(t)$ if and only if $|p_i(t) - p_j(t)| \leq 1$, where $|\cdot|$ represents the Euclidean norm. The initial Unit Ball Graph $\text{UBG}(0)$ is always connected. The connectivity and viewing ranges are equal when we study the GATHERING problem. In the context of NEAR-GATHERING, the viewing range is larger than the connectivity range. More formally, the viewing range is $1 + \tau$, for a constant $0 < \tau \leq 2/3$. Thus, the robots can observe other robots at a distance of at most $1 + \tau$. Two robots are neighbors at round t if their distance is at most the viewing range (1 for GATHERING and $1 + \tau$ for NEAR-GATHERING). Due to their viewing range, all robots have a common understanding of 1 and $1 + \tau$ (1 and τ are known to the robots). The set $N_i(t)$ contains all neighbors of r_i in round t , including r_i . Additionally, hull_i^t denotes the *local convex hull* of all neighbors of r_i , i.e., the smallest convex polytope that encloses the positions of all robots in $N_i(t)$, including r_i . We define $\text{diam}(t)$ as the maximum distance of any pair of robots at time t . Moreover, $\Delta := \text{diam}(0)$, i.e., the maximum distance of any pair of robots in the initial configuration. Lastly, $\text{diam}_i(t)$ denotes the maximum distance of any two neighbors of r_i in round t .

² Consider a protocol that solves NEAR-GATHERING for a swarm of two robots and terminates in the \mathcal{F} SYNC model. Fix the last round before termination and add a new robot visible to only one robot (the resulting swarm is not connected concerning). One of the original two robots still sees the same situation as before and will terminate, although NEAR-GATHERING is not solved.

Discrete Protocols. A discrete robot formation protocol \mathcal{P} specifies for every round $t \in \mathbb{N}_0$ how each robot determines its target point, i.e., it is an algorithm that computes the target point $\text{target}_i^{\mathcal{P}}(t)$ of each robot in the COMPUTE operation based upon its snapshot taken during LOOK. To simplify the notation, $\text{target}_i^{\mathcal{P}}(t)$ might express the target point of r_i either in the local coordinate system of r_i or in a global coordinate system (not known to r_i) – the concrete meaning is always clear based on the context. Finally, during MOVE, each robot moves to the position computed by \mathcal{P} , i.e., $p_i(t+1) = \text{target}_i^{\mathcal{P}}(t)$ for all robots r_i .

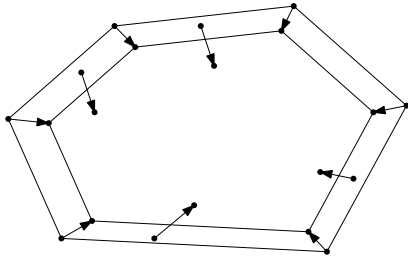
Problem Statements. GATHERING requires all robots to gather at a single, not predefined point. While the GATHERING problem clearly demands that more than one robot occupies the same position, this is prohibited in the NEAR-GATHERING problem. Two robots r_i and r_j *collide* in round t if $p_i(t) = p_j(t)$. A discrete robot formation protocol is *collision-free*, if there is no round $t' \in \mathbb{N}_0$ with a collision. NEAR-GATHERING requires all robots to maintain distinct locations, become mutually visible, and be aware of this fact in the same round/epoch. More formally, NEAR-GATHERING is solved if there is a time $t' \in \mathbb{N}_0$ and a constant $0 \leq c_{ng} \leq 1$ such that $\text{diam}(t') \leq c_{ng}$, $p_i(t'') = p_i(t')$ for all robots r_i and all rounds $t'' \geq t'$ and $p_i(t) \neq p_j(t)$ for all robots r_i and r_j and rounds t . Moreover, all robots terminate simultaneously, i.e., know in the same round or within one epoch that $\text{diam}(t) < c_{ng}$. In our protocols, $c_{ng} = \tau$. Our protocols keep $\text{UBG}(t)$ always connected and hence, robots can detect termination as soon as $\text{diam}(t) < \tau$ (due to their viewing range of $1 + \tau$). Due to the disorientation and obliviousness of the robots, any protocol to solve NEAR-GATHERING must be collision-free. As soon as two robots would move to the same location, their neighborhoods are identical and their local coordinate systems could have the same orientation such that the two robots would always compute the same movement. Hence, the robots cannot deterministically move to different locations. As a consequence, collisions must be avoided.

3 A Class of Gathering Protocols

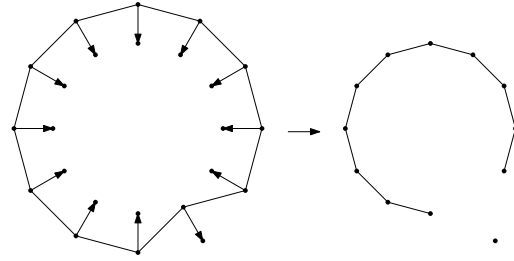
In this section, we describe the class of λ -contracting (gathering) protocols – a class of protocols which solve GATHERING in $\Theta(\Delta^2)$ rounds and serves as a basis for collision-free protocols to solve NEAR-GATHERING (see Section 4). Moreover, we derive a subclass of λ -contracting protocols, called (α, β) -contracting protocols. The class of (α, β) -contracting protocols is a powerful tool to determine whether a given gathering protocol (such as GTC) fulfills the property of being λ -contracting.

The first intuition to define a class of protocols to solve GATHERING would be to transfer the class of continuous contracting protocols (cf. Section 1.1) to the discrete LCM case. A continuous robot formation protocol is called *contracting* if robots that are part of the global convex hull move with constant speed towards the inside or along the boundary of the global convex hull. A translation to the discrete (LCM) case might be to demand that each robot moves a constant distance inwards (away from the boundary) of the global convex hull, cf. Figure 2.

However, such a protocol cannot exist in the discrete LCM setting. Consider n robots positioned on the vertices of a regular polygon with side length 1. Now take one robot and mirror its position along the line segment connecting its two neighbors (cf. Figure 3). Next, we assume that all robots would move a constant distance along the angle bisector between their direct neighbors in the given gathering protocol. Other movements would lead to the same effect since the robots are disoriented. In the given configuration, $n - 1$ robots would



■ **Figure 2** Ideally, every robot that is close to the boundary of the global convex hull would move a constant distance inwards.



■ **Figure 3** Visualization of the example to emphasize that continuous protocols cannot be directly translated to the LCM case.

move a constant distance inside the global convex hull while one robot even leaves the global convex hull. Not only that the global convex hull does not decrease as desired, but also the connectivity of $UBG(t)$ is not maintained as the robot moving outside loses connectivity to its direct neighbors. Consequently, discrete gathering protocols have to move the robots more carefully to maintain the connectivity of $UBG(t)$.

3.1 λ -contracting Protocols

Initially, we emphasize two core features of the protocols. A discrete protocol is *connectivity preserving* if it always maintains the connectivity of $UBG(t)$. Due to the limited visibility and disorientation, every protocol to solve GATHERING and NEAR-GATHERING must be connectivity preserving since it is deterministically impossible to reconnect lost robots to the remaining swarm. Moreover, we study protocols that are *invariant*, i.e., the movement of a robot does not change no matter how its local coordinate system is oriented³. This is a natural assumption since the robots have variable disorientation and thus cannot rely on their local coordinate system to synchronize their movement with nearby robots. Moreover, many known protocols under the given robot capabilities are invariant, e.g., [3, 13, 39, 40].

► **Definition 1.** Let Q be a convex polytope with diameter $diam$ and $0 < \lambda \leq 1$ a constant. A point $p \in Q$ is called to be **λ -centered** if it is the midpoint of a line segment that is completely contained in Q and has a length of $\lambda \cdot diam$.

► **Definition 2.** A connectivity preserving and invariant discrete robot formation protocol \mathcal{P} is called **λ -contracting** if $target_i^{\mathcal{P}}(t)$ is a λ -centered point of $hull_i^t$ for every robot r_i and every $t \in \mathbb{N}_0$.

Two examples of λ -centered points are depicted in Figure 1 (contained in Section 1.2). Observe that Definition 2 does not necessarily enforce a final gathering of the protocols. Consider, for instance, two robots. A protocol that demands the two robots to move halfway towards the midpoint between themselves would be $1/4$ -contracting, but the robots would only *converge* towards the same position. However, for GATHERING, the robots must compute the same target point eventually. We demand this by requiring that there is a constant $c < 1$, such that $N_i(t) = N_j(t)$ and $diam_i(t) = diam_j(t) \leq c$ implies that the robots compute the same target point. Protocols that have this property are called *collapsing*. Observe

³ Note that the protocols do not need to be invariant to ensure GATHERING. Nevertheless, being invariant becomes important when we study the NEAR-GATHERING problem. For ease of description, we consider invariant protocols in general.

that being collapsing is reasonable since λ -contracting demands that robots compute target points inside their local convex hulls and hence, the robots' local diameters are monotonically decreasing in case no further robot enters their neighborhood. Hence, demanding a threshold to enforce moving to the same point is necessary to ensure a final gathering. For ease of description, we fix $c = 1/2$ in this work. However, c could be chosen as an arbitrary constant by scaling the obtained runtime bounds with a factor of $1/c$.

► **Definition 3.** *A discrete robot formation protocol \mathcal{P} is a λ -contracting gathering protocol if \mathcal{P} is λ -contracting and collapsing.*

3.2 Analysis of λ -contracting Gathering Protocols

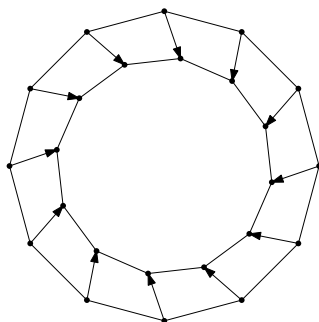
In the following, we state upper and lower bounds about λ -contracting gathering protocols. When considering $d = 1$, λ -contracting gathering protocols are optimal.

► **Theorem 4.** *Consider a swarm of robots in \mathbb{R} . Every λ -contracting gathering protocol gathers all robots in $\Theta(\Delta)$ rounds.*

The proof of Theorem 4 can be found in the full version of this paper [14]. For larger dimensions, we start with a lower bound that holds for a larger class of protocols but is especially valid for λ -contracting gathering protocols. The lower bound holds for all discrete gathering protocols that compute robot target points always inside local convex hulls. The proof of the lower bound is in most parts identical to the lower bound of the GTC protocol [24]. Essentially, we prove that, in the configuration where all robots are located on the vertices of a regular polygon with side length 1, GTC is the best possible of all protocols that compute target points inside of local convex hulls.

► **Theorem 5.** *For a swarm of n robots in \mathbb{R}^d with $d \geq 2$ and diameter Δ there exists an initial configuration such that every discrete gathering protocol \mathcal{P} that ensures $\text{target}_i^{\mathcal{P}}(t) \in \text{hull}_i^t$ for all robots r_i and all rounds $t \in \mathbb{N}_0$, requires $\Omega(\Delta^2)$ rounds to gather all robots.*

Proof. In the following, we assume $n \geq 5$. Consider n robots that are located on the vertices of a regular polygon with side length 1. Observe first that due to the disorientation and because the protocols are deterministic, the local coordinate systems of the robots could be chosen such that the configuration remains a regular polygon forever (see Figure 4 for an example).



■ **Figure 4** Initially, the robots are located on the surrounding regular polygon. The local coordinate systems of the robots can be chosen such that all robots execute the same movement in a rotated fashion such that the configuration remains a regular polygon (depicted by the inner regular polygon).

15:10 Efficient (Near)-Gathering of Disoriented Robots with Limited Visibility

Henceforth, we assume in the following that the robots remain on the vertices of a regular polygon. Let C be the surrounding circle and r_C its radius. For large n , the circumference p_C of C is $\approx n$ and $r_C \approx \frac{n}{2\pi}$. Hence, $\Delta \approx \frac{n}{\pi}$. We show that any λ -contracting protocol (not only gathering protocols) requires $\Omega(\Delta^2)$ rounds until $p_C \leq \frac{2}{3}n$. As long as $p_C \geq \frac{2}{3}n$, each robot can observe exactly two neighbors at distance $\frac{2}{3} \leq s \leq 1$.

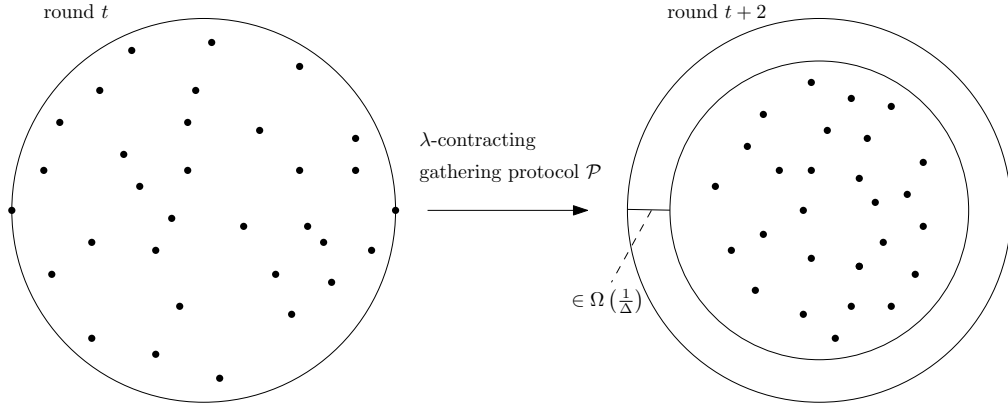
The internal angles of a regular polygon have a size of $\gamma = \frac{(n-2) \cdot \pi}{n}$. Fix any robot r_i and assume that $p_i(t) = (0, 0)$ and the two neighbors are at $p_{i-1}(t) = (-s \cdot \sin(\frac{\gamma}{2}), s \cdot \cos(\frac{\gamma}{2}))$ and $p_{i+1}(t) = p_{i-1}(t) = (s \cdot \sin(\frac{\gamma}{2}), s \cdot \cos(\frac{\gamma}{2}))$. Now, consider the target point $\text{target}_i^{\mathcal{P}}(t) = (x_{\text{target}_i^{\mathcal{P}}(t)}, y_{\text{target}_i^{\mathcal{P}}(t)})$. Observe that the radius r_C decreases by exactly $y_{\text{target}_i^{\mathcal{P}}(t)}$. Next, we derive an upper bound on $y_{\text{target}_i^{\mathcal{P}}(t)}$: $y_{\text{target}_i^{\mathcal{P}}(t)} = s \cdot \cos(\frac{\gamma}{2}) \leq \cos(\frac{\gamma}{2}) = \cos(\frac{(n-2) \cdot \pi}{2n})$.

Now, we use $\cos(x) \leq -x + \frac{\pi}{2}$ for $0 \leq x \leq \frac{\pi}{2}$. Hence, we obtain $\cos(\frac{(n-2) \cdot \pi}{2n}) \leq -\frac{(n-2) \cdot \pi}{2n} + \frac{\pi}{2} = -\frac{\pi}{2} + \frac{\pi}{n} + \frac{\pi}{2} = \frac{\pi}{n}$. Therefore, it takes at least $\frac{n^2}{3}$ rounds until r_C has decreased by at least $\frac{n}{3}$. The same holds for the perimeter. All in all, it takes at least $\frac{n^2}{3} \in \Omega(\Delta^2)$ rounds until the r_C decreases by at least $\frac{n}{3}$. ◀

We state a matching upper bound for λ -contracting protocols in two dimensions (later for any $d \geq 2$). We first focus on robots in the Euclidean plane to make the core ideas visualizable.

► **Theorem 6.** *Consider a swarm of robots in \mathbb{R}^2 with diameter Δ . Every λ -contracting gathering protocol gathers all robots in $\frac{171 \cdot \pi \cdot \Delta^2}{\lambda^3} + 1 \in \mathcal{O}(\Delta^2)$ rounds.*

High-Level Description. The proof is inspired by the proof of the GTC protocol [24]. The proof aims to show that the radius of the global smallest enclosing circle (SEC), i.e., the SEC that encloses all robots' positions in a global coordinate system, decreases by $\Omega(1/\Delta)$ every two rounds. Since the initial radius is upper bounded by Δ , the runtime of $\mathcal{O}(\Delta^2)$ follows. See Figure 5 for a visualization.



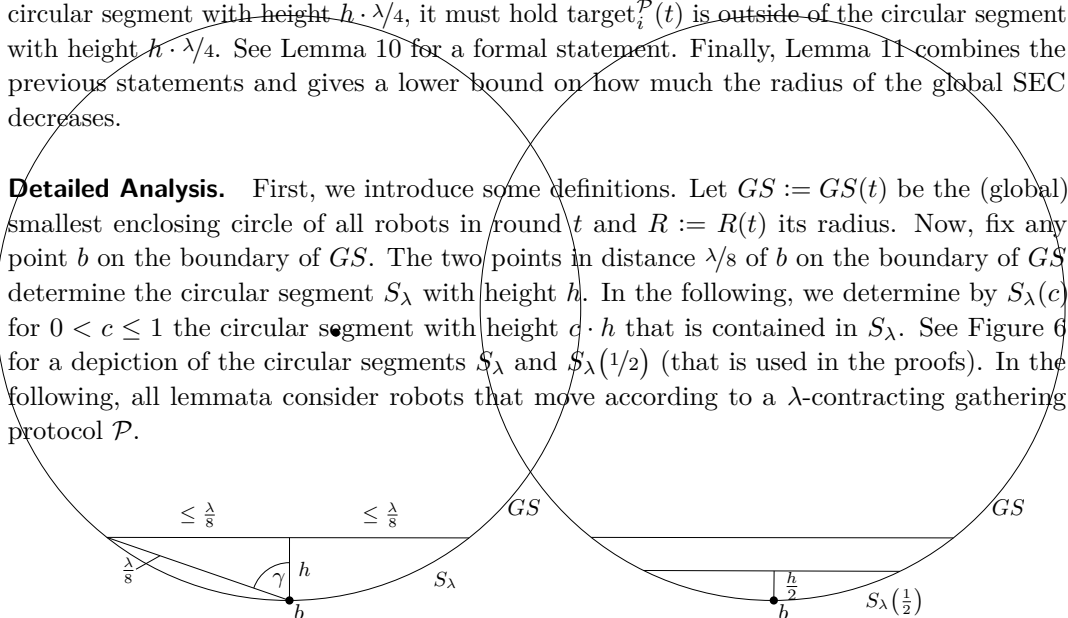
■ **Figure 5** We show that the radius of the global SEC decreases by $\Omega(1/\Delta)$ every two rounds.

We consider the fixed circular segment S_λ of the global SEC and analyze how the inside robots behave. A circular segment is a region of a circle “cut off” by a chord. The circular segment S_λ has a chord length of at most $\lambda/4$ (for a formal definition, see below) and we can prove a height h of S_λ in the order of $\Omega(1/\Delta)$ (Lemma 8). Observe that in any circular segment, the chord’s endpoints are the points that have a maximum distance within the circular segment, and hence, the maximum distance between any pair of points in S_λ is

at most $\lambda/4$. Now, we split the robots inside of S_λ into two classes: the robots r_i with $\text{diam}_i(t) > 1/4$ and the others with $\text{diam}_i(t) \leq 1/4$. Recall that every robot r_i moves to the λ -centered point $\text{target}_i^{\mathcal{P}}(t)$. Moreover, $\text{target}_i^{\mathcal{P}}(t)$ is the midpoint of a line segment ℓ of length $\lambda \cdot \text{diam}_i(t)$ that is completely contained in the local convex hull of r_i . For robots with $\text{diam}_i(t) > 1/4$ we have that ℓ is larger than $\lambda/4$ and thus, ℓ cannot be completely contained in S_λ . Hence, ℓ either connects two points outside of S_λ or one point inside and another outside. In the former case, $\text{target}_i^{\mathcal{P}}(t)$ is outside of S_λ , and in the latter case, $\text{target}_i^{\mathcal{P}}(t)$ is outside of a circular segment with half the height h of S_λ . See Lemma 9 for a formal statement of the first case.

It remains to argue about robots with $\text{diam}_i(t) < 1/4$. Here, we consider a circular segment with an even smaller height, namely $h \cdot \lambda/4$. We will see that all robots which compute a target point inside this circular segment (which can only be robots with $\text{diam}_i(t) < \lambda/4$) will move exactly to the same position. Hence, in round $t + 1$ there is only one position in the circular segment with height $h \cdot \lambda/4$ occupied by robots. All other robots are located outside of the circular segment with height $h/2$. As a consequence, for all robots r_i in the circular segment with height $h \cdot \lambda/4$, it must hold $\text{target}_i^{\mathcal{P}}(t)$ is outside of the circular segment with height $h \cdot \lambda/4$. See Lemma 10 for a formal statement. Finally, Lemma 11 combines the previous statements and gives a lower bound on how much the radius of the global SEC decreases.

Detailed Analysis. First, we introduce some definitions. Let $GS := GS(t)$ be the (global) smallest enclosing circle of all robots in round t and $R := R(t)$ its radius. Now, fix any point b on the boundary of GS . The two points in distance $\lambda/8$ of b on the boundary of GS determine the circular segment S_λ with height h . In the following, we determine by $S_\lambda(c)$ for $0 < c \leq 1$ the circular segment with height $c \cdot h$ that is contained in S_λ . See Figure 6 for a depiction of the circular segments S_λ and $S_\lambda(1/2)$ (that is used in the proofs). In the following, all lemmata consider robots that move according to a λ -contracting gathering protocol \mathcal{P} .



■ **Figure 6** The circular segments S_λ (to the left) and $S_\lambda(1/2)$ of the global SEC GS are depicted.

In the following, we prove that all robots leave the circular segment $S_\lambda(\lambda/4)$ every two rounds. As a consequence, the radius of GS decreases by at least $\lambda/4 \cdot h$. Initially, we give a bound on h . We use Jung’s Theorem (Theorem 7) to obtain a bound on R and also on h .

► **Theorem 7** (Jung’s Theorem [34, 35]). *The smallest enclosing hypersphere of a point set $K \subset \mathbb{R}^d$ with diameter diam has a radius of at most $\text{diam} \cdot \sqrt{\frac{d}{2 \cdot (d+1)}}$.*

► **Lemma 8.** $h \geq \frac{\sqrt{3} \cdot \lambda^2}{64\pi\Delta}$.

Proof. Initially, we give an upper bound on the angle γ , see Figure 6 for its definition. The circumference of GS is $2\pi R$. We can position at most $\frac{16}{\lambda}\pi R$ points on the boundary of GS that are at distance $\frac{\lambda}{8}$ from the points closest to them and form a regular convex polygon. The internal angle of this regular polygon is 2γ . Hence, the sum of all internal angles is

15:12 Efficient (Near)-Gathering of Disoriented Robots with Limited Visibility

$(\frac{16}{\lambda}\pi R - 2) \cdot \pi$. Thus, each individual angle has a size of at most $\frac{(\frac{16}{\lambda}\pi R - 2) \cdot \pi}{\frac{16}{\lambda}\pi R} = \pi - \frac{2\pi}{\frac{16}{\lambda}\pi R} = \pi - \frac{\lambda}{8R}$. Hence, $\gamma \leq \frac{\pi}{2} - \frac{\lambda}{16R}$. Now, we are able to bound h . First of all, we derive a relation between h and γ : $\cos(\gamma) = \frac{h}{\frac{\lambda}{8}} = \frac{8h}{\lambda} \iff h = \frac{\lambda \cdot \cos(\gamma)}{8}$. In the following upper bound, we make use of the fact that $\cos(x) \geq -\frac{2}{\pi}x + 1$ for $x \in [0, \frac{\pi}{2}]$.

$$h = \frac{\lambda \cdot \cos(\gamma)}{8} \geq \frac{\lambda \cdot \cos\left(\frac{\pi}{2} - \frac{\lambda}{16R}\right)}{8} \geq \frac{\lambda \cdot \left(-\frac{2}{\pi} \cdot \left(\frac{\pi}{2} - \frac{\lambda}{16R}\right) + 1\right)}{8} = \frac{\lambda \cdot \frac{\lambda}{8\pi R}}{8} = \frac{\lambda^2}{64\pi R}$$

Applying Theorem 7 with $d = 2$ yields $h \geq \frac{\sqrt{3} \cdot \lambda^2}{64\pi\Delta}$. \blacktriangleleft

We continue to prove that all robots leave $S_\lambda(\lambda/4)$ every two rounds. First of all, we analyze robots for which $\text{diam}_i(t) > 1/4$. These robots even leave the larger circular segment $S_\lambda(1/2)$.

► **Lemma 9.** *For any robot r_i with $\text{diam}_i(t) > 1/4$: $\text{target}_i^{\mathcal{P}}(t) \in GS \setminus S_\lambda(1/2)$.*

Proof. Since $\text{diam}_i(t) > 1/4$ and \mathcal{P} is λ -contracting, $\text{target}_i^{\mathcal{P}}(t)$ is the midpoint of a line segment $\ell_i^{\mathcal{P}}(t)$ of length at least $\lambda \cdot \text{diam}_i(t) > \lambda/4$. As the maximum distance between any pair of points inside of S_λ is $\frac{\lambda}{4}$, it follows that $\ell_i^{\mathcal{P}}(t)$ either connects two points outside of S_λ or one point inside and another point outside. In the first case, $\text{target}_i^{\mathcal{P}}(t)$ lies outside of S_λ (since the maximum distance between any pair of points inside of S_λ is $\frac{\lambda}{4} \leq 1/4 < \text{diam}_i(t)$). In the second case, $\text{target}_i^{\mathcal{P}}(t)$ lies outside of $S_\lambda(1/2)$ since, in the worst case, one endpoint of $\ell_i^{\mathcal{P}}(t)$ is the point b used in the definition of GS (see the beginning of Section 3.2) and the second point lies very close above of $S_\lambda(1/2)$. Since $\text{target}_i^{\mathcal{P}}(t)$ is the midpoint of $\ell_i^{\mathcal{P}}(t)$, it lies closely above of $S_\lambda(1/2)$. Every other position of the two endpoints of $\ell_i^{\mathcal{P}}(t)$ would result in a point $\text{target}_i^{\mathcal{P}}(t)$ that lies even farther above of $S_\lambda(1/2)$. \blacktriangleleft

Now, we consider the case of a single robot in $S_\lambda(\lambda/4)$, and its neighbors are located outside of $S_\lambda(1/2)$. We prove that this robot leaves $S_\lambda(\lambda/4)$. Additionally, we prove that none of the robots outside of $S_\lambda(1/2)$ that see the single robot in $S_\lambda(\lambda/4)$ enters $S_\lambda(\lambda/4)$.

► **Lemma 10.** *Consider a robot r_i located in $S_\lambda(\lambda/4)$. If all its neighbors are located outside of $S_\lambda(1/2)$, $\text{target}_i^{\mathcal{P}}(t) \in GS \setminus S_\lambda(\lambda/4)$. Similarly, for a robot r_i that is located outside of $S_\lambda(1/2)$ and that has only one neighbor located in $S_\lambda(\lambda/4)$, $\text{target}_i^{\mathcal{P}}(t) \in GS \setminus S_\lambda(\lambda/4)$.*

Proof. First, we consider a robot r_i that is located in $S_\lambda(\lambda/4)$ and all its neighbors are above of $S_\lambda(1/2)$. Let p_1 and p_2 be the two points of hull_i^t closest to the intersection points of hull_i^t and the boundary of $S_\lambda(1/2)$ (p_1 and p_2 are infinitesimally above of $S_\lambda(1/2)$). In case hull_i^t consists of only two robots, define p_1 to be the intersection point of hull_i^t and $S_\lambda(1/2)$ and $p_2 = p_i(t)$. The maximum distance d_{\max} between any pair of points in $\text{hull}_i^t \cap S_\lambda(1/2)$ is less than $\max\{|p_1 - p_2|, |p_1 - p_i(t)|, |p_2 - p_i(t)|\}$, since p_1 and p_2 are slightly above of $S_\lambda(1/2)$. Clearly, $\text{diam}_i(t) \geq d_{\max}$. Thus, the maximum distance between any pair of points in $\text{hull}_i^t \cap S_\lambda(\frac{\lambda}{2})$ is less than $\lambda \cdot d_{\max}$. We conclude that $\text{target}_i^{\mathcal{P}}(t)$ must be located above of $S_\lambda(\lambda/4)$ since $\text{target}_i^{\mathcal{P}}(t)$ is the midpoint of a line segment of length $\lambda \cdot \text{diam}_i(t) \geq \lambda \cdot d_{\max}$ either connecting two robots above of $S_\lambda(\lambda/4)$ or one robot inside of $S_\lambda(\lambda/4)$ and one robot outside of $S_\lambda(\lambda/4)$. The arguments for the opposite case – r_i is located in $S_\lambda(1/2)$, one neighbor of r_i is located in $S_\lambda(\lambda/4)$ and all others are also outside of $S_\lambda(1/2)$ – are analogous. \blacktriangleleft

Next, we derive with help of Lemmas 9 and 10 that the circular segment $S_\lambda(\lambda/4)$ is empty after two rounds. Additionally, we analyze how much $R(t)$ decreases.

► **Lemma 11.** *For any round t with $\text{diam}(t) \geq 1/2$, $R(t+2) \leq R(t) - \frac{\lambda^3 \cdot \sqrt{3}}{256 \cdot \pi \cdot \Delta}$.*

Proof. Fix any circular S_λ and consider the set of robots R_S that are located in $S_\lambda(\lambda/4)$ or compute a target point in $S_\lambda(\lambda/4)$. Initially, we argue that all robots in R_S can see each other. Via Lemma 9, we obtain that for every robot $r_i \in R_S$ that computes a target point in $S_\lambda(\lambda/4)$, $\text{diam}_i(t) \leq 1/4$. Since the maximum distance between any pair of points in $S_\lambda(\lambda/4)$ is less than $1/4$ (as the maximum distance of any points in the larger circular segment S_λ is $\lambda/4$), we conclude that, a robot which is not located in $S_\lambda(\lambda/4)$ but computes its target point inside, is at distance at most $1/4$ from $S_\lambda(\lambda/4)$. Hence, via the triangle inequality, it is located at distance at most $1/2$ from any other robot in R_S . Thus, all robots in R_S can see each other. Now consider the robot $r_{\min} \in R_S$ which is one of the robots of R_S with the minimal number of visible neighbors. Furthermore, A_{\min} is the set of robots that have exactly the same neighborhood as r_{\min} . For all robots $r_j \in R_S \setminus A_{\min}$, we have that r_j can see r_{\min} and at least one robot that r_{\min} cannot see. Thus, $\text{diam}_j(t) > 1$. We can conclude with help of Lemma 9 that all robots in $R_S \setminus A_{\min}$ compute a target point outside of $S_\lambda(1/2)$. Since all robots $r_i \in A_{\min}$ have the same neighborhood and $\text{diam}_i(t) < 1/4$, they also compute the same target point (λ -contracting gathering protocols are collapsing). Thus, at the beginning of round $t+1$, at most one position in $S_\lambda(\lambda/4)$ is occupied. In round $t+1$ we have the picture that one position in $S_\lambda(\lambda/4)$ is occupied and all neighbors are located above of $S_\lambda(1/2)$. Lemma 10 yields that the robots in $S_\lambda(\lambda/4)$ compute a target point outside. Moreover, Lemma 10 yields as well that no robot outside of $S_\lambda(\lambda/4)$ computes a target point inside and thus, $S_\lambda(\lambda/4)$ is empty in round $t+2$. Since the circular segment S_λ has been chosen arbitrarily, the arguments hold for the entire circle GS and thus, $R(t+2) \leq R(t) - \lambda/4 \cdot h \leq R(t) - \frac{\lambda^3 \cdot \sqrt{3}}{256 \cdot \pi \cdot \Delta}$. ◀

Finally, we can conclude with help of Lemma 11 the main Theorem 6.

Proof of Theorem 6. First, we bound the initial radius of GS : $R(0) \leq \Delta/\sqrt{3}$ (Theorem 7). Lemma 11 yields that $R(t)$ decreases every two rounds by at least $\frac{\lambda^3 \cdot \sqrt{3}}{256 \cdot \pi \cdot \Delta}$. Thus, it requires $2 \cdot \frac{256 \cdot \pi \cdot \Delta}{\lambda^3}$ rounds until $R(t)$ decreases by at least $\sqrt{3}$. Next, we bound how often this can happen until $R(t) \leq \frac{1}{4}$ and thus $\text{diam}(t) \leq \frac{1}{2}$: $\frac{\Delta}{\sqrt{3}} - x \cdot \sqrt{3} \leq \frac{1}{4} \iff \frac{\Delta}{3} - \frac{1}{4 \cdot \sqrt{3}} \leq x$.

All in all, it requires $x \cdot \frac{512 \cdot \pi \cdot \Delta}{\lambda^3} = \left(\frac{\Delta}{3} - \frac{1}{4 \cdot \sqrt{3}}\right) \cdot \frac{512 \cdot \pi \cdot \Delta}{\lambda^3} \leq \frac{171 \cdot \pi \cdot \Delta^2}{\lambda^3}$ rounds until $\text{diam}(t) \leq \frac{1}{2}$. As soon as $\text{diam}(t) \leq \frac{1}{2}$, all robots can see each other, compute the same target point and will reach it in the next round. ◀

Upper Bound in d -dimensions. The upper bound we derived for two dimensions can also be generalized to every dimension d . Only the constants in the runtime increase slightly.

► **Theorem 12.** *Consider a team of n robots located in \mathbb{R}^d . Every λ -contracting gathering protocol gathers all robots in $\frac{256 \cdot \pi \cdot \Delta^2}{\lambda^3} + 1 \in \mathcal{O}(\Delta^2)$ rounds.*

3.3 Examples of λ -contracting Gathering Protocols

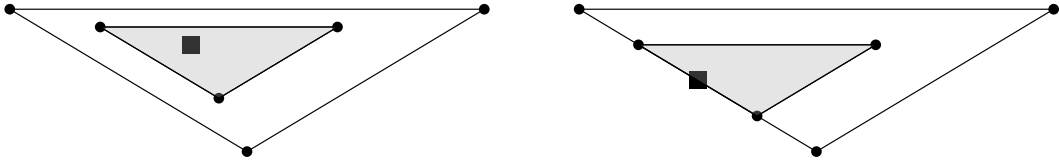
Next, we present examples of λ -contracting gathering protocols. Before introducing the concrete protocols, we describe an important subclass of λ -contracting protocols, denoted as (α, β) -contracting protocols, a powerful tool to decide whether a given protocol is λ -contracting. Afterward, we introduce the known protocol GTC [3] and prove it to be λ -contracting. Additionally, we introduce two further two-dimensional λ -contracting gathering protocols: GTMD and GTCDMB.

(α, β) -contracting Protocols. While the definition of λ -contracting gathering protocols describes the core properties of efficient protocols to solve GATHERING, it might be practically challenging to determine whether a given protocol is λ -contracting. Concrete protocols often are designed as follows: robots compute a *desired* target point and move as close as possible towards it without losing connectivity [3, 13, 39]. The GTC protocol, for instance, uses this rule. Since the robots do not necessarily reach the desired target point, it is hard to determine whether the resulting point is λ -centered. Therefore, we introduce a two-stage definition: (α, β) -contracting protocols. The parameter α represents an α -centered point (Definition 1) and β describes how close the robots move towards the point.

► **Definition 13.** Let c_1, \dots, c_k with $c_i \in \mathbb{R}^d$ be the vertices of a convex polytope Q , $p \in Q$ and $0 < \beta \leq 1$ a constant. $Q(p, \beta)$ is the convex polytope with vertices $p + (1 - \beta) \cdot (c_i - p)$.

Now, we are ready to define the class of (α, β) -contracting protocols. It uses a combination of Definitions 1 and 13: the target points of the robots must be inside of the β -scaled local convex hull around an α -centered point. See also Figure 7 for a visualization of valid target points in (α, β) -contracting protocols. Recall that hull_i^t defines the convex hull of all neighbors of r_i including r_i in round t and $\text{hull}_i^t(p, \beta)$ is the scaled convex hull around p (Definition 13).

► **Definition 14.** A connectivity preserving and invariant discrete robot formation protocol \mathcal{P} is called to be **(α, β) -contracting**, if there exists an α -centered point $\alpha\text{-center}_i^{\mathcal{P}}(t)$ s.t. $\text{target}_i^{\mathcal{P}}(t) \in \text{hull}_i^t(\alpha\text{-center}_i^{\mathcal{P}}(t), \beta)$ for every robot r_i and every $t \in \mathbb{N}_0$. Moreover, \mathcal{P} is called an **(α, β) -contracting gathering protocol** if \mathcal{P} is (α, β) -contracting and collapsing.



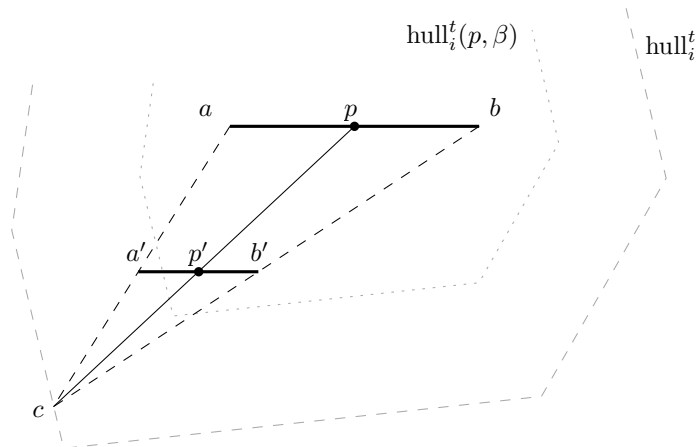
■ **Figure 7** Two examples of valid target points of (α, β) -contracting protocols. The small gray triangle represents the $\frac{1}{2}$ -scaled convex hull around an $\frac{1}{4}$ -centered point marked with a square.

Next, we state the relation between (α, β) -contracting and λ -contracting protocols.

► **Theorem 15.** Every (α, β) -contracting protocol \mathcal{P} is λ -contracting with $\lambda = \alpha \cdot \beta$.

Proof. From the definition of (α, β) -contracting protocols, we know that for a target point $\text{target}_i^{\mathcal{P}}(t)$, there exists a point $\alpha\text{-center}_i^{\mathcal{P}}(t)$ such that $\text{target}_i^{\mathcal{P}}(t) \in \text{hull}_i^t(\alpha\text{-center}_i^{\mathcal{P}}(t), \beta)$. We do the following geometric construction in Figure 8. Let $p = \alpha\text{-center}_i^{\mathcal{P}}(t)$ and $p' = \text{target}_i^{\mathcal{P}}(t)$. We draw a line segment from $\alpha\text{-center}_i^{\mathcal{P}}(t)$ through $\text{target}_i^{\mathcal{P}}(t)$ to the boundary of hull_i^t . Let c be the endpoint of this line segment. Because p is α -centered, there exists a line segment with length $\text{diam}_i(t) \cdot \alpha$ through p , let this be the line segment \overline{ab} . The line segment $\overline{a'b'}$ is a parallel to \overline{ab} inside the triangle Δ_{abc} . We know that $p' \in \text{hull}_i^t(p, \beta)$, therefore $|\overline{cp'}| \geq \beta|\overline{cp}|$. By the intercept theorem, it follows that $|\overline{a'b'}| \geq \beta|\overline{ab}| = \beta \cdot \alpha \cdot \text{diam}_i(t)$. Because the points a, b and c are all inside hull_i^t , the entire triangle Δ_{abc} and $\overline{a'b'}$ are inside hull_i^t as well. Therefore, $\text{target}_i^{\mathcal{P}}(t)$ is a λ -centered point with $\lambda = \alpha \cdot \beta$. ◀

Go-To-The-Center. As a first example, we study the two-dimensional GTC protocol [3]. It is already known that it gathers all robots in $\mathcal{O}(n + \Delta^2)$ rounds [24]. We show that GTC is (α, β) -contracting (hence also λ -contracting) and thus, obtain an improved upper runtime



■ **Figure 8** The construction used in the proof of Theorem 15.

bound of $\mathcal{O}(\Delta^2)$. The formal description of the GTC protocol can be found in [14]. Robots always move towards the center of the smallest enclosing circle of their neighborhood. To maintain connectivity, *limit circles* are used. Each robot r_i always stays within the circle of radius $1/2$ centered in the midpoint m_j of every visible robot r_j . Since each robot r_j does the same, it is ensured that two visible robots always stay within a circle of radius $1/2$ and thus, they remain connected. Consequently, robots move only that far towards the center of the smallest enclosing circle such that no limit circle is left.

► **Theorem 16.** *GTC is $(\sqrt{3}/8, 1/2)$ -contracting.*

GTC can be generalized to d -dimensions by moving robots toward the center of the smallest enclosing hypersphere of their neighborhood. We denote the resulting protocol by d -GTC, a complete description is deferred to [14].

► **Theorem 17.** *d -GTC is $(\sqrt{2}/8, 1/2)$ -contracting.*

Go-To-The-Middle-Of-The-Diameter (GtMD). Next, we describe a second two-dimensional protocol that is also (α, β) -contracting. The intuition is quite simple: a robot r_i moves towards the midpoint of the two robots defining $\text{diam}_i(t)$. Similar to the GTC protocol, connectivity is maintained with the help of limit circles. A robot only moves that far towards the midpoint of the diameter such that no limit circle (a circle with radius $1/2$ around the midpoint of r_i and each visible robot r_j) is left. Observe further that the midpoint of the diameter is not necessarily unique. To make GtMD in cases where the midpoint of the diameter is not unique deterministic, robots move according to GTC. The formal description can be found in [14]. We prove the following property about GtMD.

► **Theorem 18.** *In rounds, where the local diameter of all robots is unique, GtMD is $(1, 1/10)$ -contracting $((\sqrt{3}/8, 1/2)$ -contracting otherwise).*

Go-To-The-Center-Of-The-Diameter-MinBox (GtCDBM). Lastly, we derive a third protocol for robots in \mathbb{R}^2 that is also (α, β) -contracting. It is based on the local *diameter minbox* defined as follows. The local coordinate system is adjusted such that the two robots that define the diameter are located on the y -axis, and the midpoint of the diameter coincides with the origin. Afterwards, the maximal and minimal x -coordinates x_{\max} and x_{\min} of other

visible robots are determined. Finally, the robot moves towards $(\frac{1}{2} \cdot (x_{\min} + x_{\max}), 0)$. The box boundaries with x -coordinates x_{\min}, x_{\max} and y -coordinates $-\text{diam}_i(t)/2$ and $\text{diam}_i(t)/2$ is called the *diameter minbox* of r_i . Note that, similar to GTMD, the diameter minbox of r_i might not be unique. In this case, a fallback to GTC is used. The complete description of GTCDMB is contained in [14]. Also, GTCDMB is (α, β) -contracting.

► **Theorem 19.** *In rounds, where the local diameter of all robots is unique, GTCDMB is $(\sqrt{3}/8, 1/10)$ -contracting ($(\sqrt{3}/8, 1/2)$ -contracting otherwise).*

4 Collision-free Near-Gathering Protocols

In this section, we study the NEAR-GATHERING problem for robots located in \mathbb{R}^d under the \mathcal{SSYNC} scheduler. The main difference to GATHERING is that robots may never collide (move to the same position). We introduce a very general approach to NEAR-GATHERING that builds upon λ -contracting gathering protocols (Section 3). We show how to transform *any* λ -contracting gathering protocol into a *collision-free* λ -contracting protocol that solves NEAR-GATHERING in $\mathcal{O}(\Delta^2)$ epochs under the \mathcal{SSYNC} scheduler. The only difference in the robot model (compared to GATHERING in Section 3) is that we need a slightly stronger assumption on the connectivity: the connectivity range must be by an additive constant smaller than the viewing range. More formally, the connectivity range is 1 while robots have a viewing range of $1 + \tau$ for a constant $0 < \tau \leq 2/3$. Note that the upper bound on τ is only required because $\tau/2$ also represents the maximum movement distance of a robot (see below). In general, the viewing range could also be chosen larger than $1 + \tau$ without any drawbacks while keeping the maximum movement distance at $\tau/2$.

The main idea of our approach can be summarized as follows: first, robots compute a *potential* target point based on a λ -contracting gathering protocol \mathcal{P} that considers only robots at a distance at most 1. Afterward, a robot r_i uses the viewing range of $1 + \tau$ to determine whether its potential target point collides with any potential target point of a nearby neighbor. If there might be a collision, r_i does not move to its potential target point. Instead, it only moves to a point between itself and the potential target point where no other robot moves to. At the same time, it is also ensured that r_i moves sufficiently far towards the potential target point to maintain the time bound of $\mathcal{O}(\Delta^2)$ epochs. To realize the ideas with a viewing range of $1 + \tau$, we restrict the maximum movement distance of any robot to $\tau/2$. More precisely, if the potential target point of any robot given by \mathcal{P} is at a distance of more than $\frac{\tau}{2}$, the robot moves at most $\frac{\tau}{2}$ towards it. With this restriction, each robot could only collide with other robots at a distance of at most τ . The viewing range of $1 + \tau$ allows computing the potential target point based on \mathcal{P} of all neighbors at a distance at most τ . By knowing all these potential target points, the own target point of the collision-free protocol can be chosen. While this only summarizes the key ideas, we give a more technical intuition and a summary of the proof in Section 4.2.

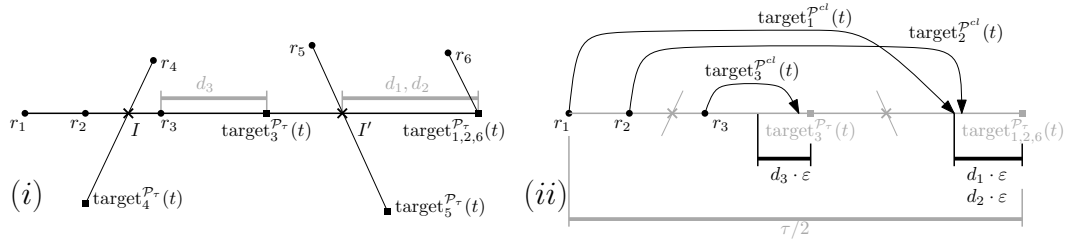
► **Theorem 20.** *For every λ -contracting gathering protocol \mathcal{P} , there exists a collision-free λ -contracting protocol \mathcal{P}^{cl} which solves NEAR-GATHERING in $\mathcal{O}(\Delta^2)$ epochs under the \mathcal{SSYNC} scheduler. Let 1 be the viewing and connectivity range of \mathcal{P} . \mathcal{P}^{cl} has a connectivity range of 1 and viewing range of $1 + \tau$ for a constant $0 < \tau \leq 2/3$.*

4.1 Collision-free Protocol

The construction of the collision-free protocol $\mathcal{P}^{cl}(\mathcal{P}, \tau, \varepsilon)$ depends on several parameters that we briefly define. \mathcal{P} is a λ -contracting gathering protocol (designed for robots with a viewing range of 1). The constant τ has two purposes. The robots have a viewing range

of $1 + \tau$ and $\tau/2$ is the maximum movement distance of any robot, $0 < \tau \leq 2/3$. Lastly, the constant $\varepsilon \in (0, 1/2)$ determines how close each robot moves towards its target point based on \mathcal{P} . To simplify the notation, we usually write \mathcal{P}^{cl} instead of $\mathcal{P}^{cl}(\mathcal{P}, \tau, \varepsilon)$. Subsequently, we formally define $\mathcal{P}^{cl}(\mathcal{P}, \tau, \varepsilon)$. The description is split into three parts that can be found in Algorithms 1–3. The main routine is contained in Algorithm 1. The other two Algorithms 2 and 3 are used as subroutines.

The computation of $\text{target}_i^{\mathcal{P}^{cl}}(t)$ is based on the movement r_i would do in a slightly modified version of \mathcal{P} , denoted as \mathcal{P}_τ . The protocol \mathcal{P}_τ is defined in Algorithm 3 and a detailed intuition of why it is needed can be found in Section 4.2. The position of $\text{target}_i^{\mathcal{P}^{cl}}(t)$ lies on the *collision vector* $\text{colvec}_i^{\mathcal{P}_\tau}(t)$, the vector from $p_i(t)$ to $\text{target}_i^{\mathcal{P}_\tau}(t)$. On $\text{colvec}_i^{\mathcal{P}_\tau}(t)$, there may be several *collision points*. These are either current positions, potential target points ($\text{target}_k^{\mathcal{P}_\tau}(t)$) of other robots r_k or single intersection points between $\text{colvec}_i^{\mathcal{P}_\tau}(t)$ and another collision vector $\text{colvec}_k^{\mathcal{P}_\tau}(t)$. The computation of collision points is defined in Algorithm 2. Moreover, $d_i > 0$ is the minimal distance between a collision point and $\text{target}_i^{\mathcal{P}_\tau}(t)$. The final target point $\text{target}_i^{\mathcal{P}^{cl}}(t)$ is exactly at distance $d_i \cdot \varepsilon \cdot 2/\tau \cdot |\text{colvec}_i^{\mathcal{P}_\tau}(t)|$ from $\text{target}_i^{\mathcal{P}_\tau}(t)$. Figure 9 gives an example of collision points and target points of \mathcal{P}^{cl} .



■ **Figure 9** Example of $\text{target}_i^{\mathcal{P}^{cl}}(t)$ with $\tau = 2/3$ and $\varepsilon = 0.49$. (i) shows the collision points and computation of d_1, d_2 and d_3 (line 3 in Algorithm 1). (ii) shows the positions where r_1, r_2 and r_3 will move to in protocol \mathcal{P}^{cl} as returned by Algorithm 1.

■ **Algorithm 1** $\text{target}_i^{\mathcal{P}^{cl}(\mathcal{P}, \tau, \varepsilon)}(t)$.

-
- 1: $R_i \leftarrow \{r_k : |p_k(t) - p_i(t)| \leq \tau\}$ ▷ Robots in radius τ around r_i (including r_i)
 - 2: $C_i \leftarrow \text{collisionPoints}_i^{\mathcal{P}_\tau}(R_i, t)$ ▷ Collision points on $\text{colvec}_i^{\mathcal{P}_\tau}(t)$, see Algorithm 2
 - 3: $d_i \leftarrow \min \left(\left\{ |c - \text{target}_i^{\mathcal{P}_\tau}(t)| : c \in C_i \setminus \{\text{target}_i^{\mathcal{P}_\tau}(t)\} \right\} \right)$ ▷ min. dist. to collision point
 - 4: **return** point on $\text{colvec}_i^{\mathcal{P}_\tau}(t)$ with distance $d_i \cdot \varepsilon \cdot 2/\tau \cdot |\text{colvec}_i^{\mathcal{P}_\tau}(t)|$ to $\text{target}_i^{\mathcal{P}_\tau}(t)$
-

4.2 Proof Summary and Intuition

In the following, we describe the technical intuitions behind the protocol \mathcal{P}^{cl} . Since the intuition is closely interconnected with the formal analysis, we also give a proof outline here. The proofs of all stated lemmas and theorems can be found in [14]. The entire protocol \mathcal{P}^{cl} is described in Section 4.1. The idea for \mathcal{P}^{cl} is straightforward: robots compute a potential target point based on a λ -contracting gathering protocol \mathcal{P} (that uses a viewing range of 1), restrict the maximum movement distance to $\tau/2$ and use the viewing range of $1 + \tau$ to avoid collisions with robots in the distance at most τ . However, there are several technical details we want to emphasize in this section.

Algorithm 2 collisionPoints $_i^{\mathcal{P}}(R_i, t)$.

```

1:  $C_i \leftarrow$  empty set
2: for all  $r_k \in R_i$  do
3:   compute target $_k^{\mathcal{P}}(t)$  and collvec $_k^{\mathcal{P}}(t)$  in local coordinate system of  $r_i$ 
4:   if  $p_k(t) \in \text{collvec}_k^{\mathcal{P}}(t)$  then
5:     add  $p_k(t)$  to  $C_i$   $\triangleright$  position of  $r_k$ 
6:   if target $_k^{\mathcal{P}}(t) \in \text{collvec}_i^{\mathcal{P}}(t)$  then
7:     add target $_k^{\mathcal{P}}(t)$  to  $C_i$ 
8:   if collvec $_k^{\mathcal{P}}(t)$  intersects collvec $_i^{\mathcal{P}}(t)$  and is not collinear to collvec $_i^{\mathcal{P}}(t)$  then
9:     add intersection point between collvec $_k^{\mathcal{P}}(t)$  and collvec $_i^{\mathcal{P}}(t)$  to  $C_i$ 
10: return  $C_i$ 

```

Algorithm 3 target $_i^{\mathcal{P}_\tau}(t)$.

```

1: if robots in range 1 have pairwise distance  $\leq \tau/2$  then
2:    $\mathcal{P}^{1+\tau/2} \leftarrow$  protocol  $\mathcal{P}$  scaled to viewing range  $1 + \tau/2$ 
3:    $P_i \leftarrow$  target $_i^{\mathcal{P}^{1+\tau/2}}(t)$ 
4: else
5:    $P_i \leftarrow$  target $_i^{\mathcal{P}}(t)$ 
6: if distance  $p_i(t)$  to  $P_i > \tau/2$  then
7:   return point with distance  $\tau/2$  to  $p_i(t)$  between  $p_i(t)$  and  $P_i$ 
8: else
9:   return  $P_i$ 

```

For the correctness and the runtime analysis of the protocol \mathcal{P}^{cl} , we would like to use the insights into λ -contracting protocols derived in Section 3. However, since the robots compute their potential target point based on a λ -contracting gathering protocol \mathcal{P} with viewing range 1, this point must not necessarily be λ -centered concerning the viewing range of $1 + \tau$. We discuss this problem in more detail in Section 4.3 and motivate the *intermediate* protocol \mathcal{P}_τ that is λ -contracting with respect to the viewing range of $1 + \tau$. \mathcal{P}_τ is only an intermediate protocol since robots still may collide. Afterward in Section 4.4, we explain how to implement collision avoidance and transform the intermediate protocol \mathcal{P}_τ into the collision-free protocol \mathcal{P}^{cl} . Lastly, in Section 4.5 we argue that our transformed and collision-free protocol is still λ -contracting and derive the running time from this property.

4.3 The protocol \mathcal{P}_τ

Recall that the main goal is to compute potential target points based on a λ -contracting gathering protocol \mathcal{P} with viewing range 1. Unfortunately, a direct translation of the protocol loses the λ -contracting property in general. Consider the following example which is also depicted in Figure 10. Assume there are the robots r_1, r_2, r_3 and r_4 in one line with respective distances of $1/n, 1 + 1/n$ and $1 + \tau$ to r_1 . It can easily be seen, that the target point target $_1^{\mathcal{P}}(t)$ (protocol \mathcal{P} has only a viewing range of 1) is between r_1 and r_2 . Such a target point can never be λ -centered with $\lambda > 2/n$ for \mathcal{P}^{cl} (with viewing range $1 + \tau$).

Next, we argue how to transform the protocol \mathcal{P} with viewing range 1 into a protocol \mathcal{P}_τ with viewing range $1 + \tau$ such that \mathcal{P}_τ is λ -contracting gathering protocol. The example above already emphasizes the main problem: robots can have very small local diameters $\text{diam}_i(t)$. Instead of moving according to \mathcal{P} , those robots compute a target point based

on $\mathcal{P}^{1+\tau/2}$, which is a λ -contracting gathering protocol concerning the viewing range of $1 + \tau/2$. Protocol $\mathcal{P}^{1+\tau/2}$ is obtained by scaling \mathcal{P} to the larger viewing range of $1 + \tau/2$. More precisely, robots r_i with $\text{diam}_i(t) \leq \tau/2$ compute their target points based on $\mathcal{P}^{1+\tau/2}$ and all others according to \mathcal{P} . In addition, \mathcal{P}_τ ensures that no robot moves more than a distance of $\tau/2$ towards the target points computed in \mathcal{P} and $\mathcal{P}^{1+\tau/2}$. The first reason is to maintain the connectivity of $\text{UBG}(t)$. While the protocol \mathcal{P} maintains connectivity by definition, the protocol $\mathcal{P}^{1+\tau/2}$ could violate the connectivity of $\text{UBG}(t)$. Restricting the movement distance to $\tau/2$ and upper bounding τ by $2/3$ resolves this issue since for all robots r_i that move according to $\mathcal{P}^{1+\tau/2}$, $\text{diam}_i(t) \leq \tau/2$. Hence, after moving according to $\mathcal{P}^{1+\tau/2}$, the distance to any neighbor is at most $3 \cdot \tau/2$. Since τ is upper bounded by $2/3$, the distance is at most 1 afterward.

► **Lemma 21.** *Let \mathcal{P} be a λ -contracting gathering protocol with a viewing range of 1. $\text{UBG}(t)$ stays connected while executing \mathcal{P}_τ .*

The second reason is that moving at most $\tau/2$ makes sure that collisions are only possible within a range of τ . This is crucial for our collision avoidance which is addressed in the following section. While \mathcal{P}_τ has a viewing range of $1 + \tau$, it never uses its full viewing range for computing a target point. Either, it simulates \mathcal{P} with a viewing range of 1, or $\mathcal{P}^{1+\tau/2}$ with one of $1 + \tau/2$. It is observable that the $1 + \tau/2$ surrounding must always have a diameter $\geq \tau/2$ (see Section 4.5 for more details). Hence, the diameter of robots used for the simulation of \mathcal{P} or $\mathcal{P}^{1+\tau/2}$ cannot be less than $\Omega(\tau)$. The constant λ can be chosen accordingly.

► **Lemma 22.** *Let \mathcal{P} be a λ -contracting gathering protocol. \mathcal{P}_τ is a λ' -contracting gathering protocol with $\lambda' = \lambda \cdot \frac{\tau}{4(1+\tau)}$.*

To conclude, the protocol \mathcal{P}_τ has two main properties: it restricts the movement distance of any robot to at most $\tau/2$ and robots r_i with $\text{diam}_i(t) \leq \tau/2$ compute their target points based on protocol $\mathcal{P}^{1+\tau/2}$ with viewing range $1 + \tau/2$.

4.4 Collision Avoidance

Next, we argue how to transform the protocol \mathcal{P}_τ into the collision-free protocol \mathcal{P}^{cl} . The viewing range of $1 + \tau$ in \mathcal{P}^{cl} allows a robot r_i to compute $\text{target}_k^{\mathcal{P}_\tau}(t)$ (the target point in protocol \mathcal{P}_τ) for all robots r_k within distance at most τ . Since the maximum movement distance of a robot in \mathcal{P}_τ is $\tau/2$, this enables r_i to know the movement directions of all robots r_k which can collide with r_i . We will ensure that each robot r_i moves to some position on $\text{colvec}_i^{\mathcal{P}_\tau}(t)$ and avoids positions of all other $\text{colvec}_k^{\mathcal{P}_\tau}(t)$. Henceforth, no collision can happen. While this is the basic idea of our collision avoidance, there are some details to add.

First of all, \mathcal{P}_τ has the same viewing range as \mathcal{P}^{cl} of $1 + \tau$. However, it never uses the full viewing range to compute the target position $\text{target}_i^{\mathcal{P}_\tau}(t)$. We consider two robots r_i and r_k with distance $\leq \tau$. If r_k simulates \mathcal{P} to compute $\text{target}_k^{\mathcal{P}_\tau}(t)$, r_i can compute $\text{target}_k^{\mathcal{P}_\tau}(t)$ as well since r_i is able to observe all robots in distance 1 around r_k . If r_k simulates $\mathcal{P}^{1+\tau/2}$, the condition in \mathcal{P}_τ makes sure that r_i and r_k have a distance of $\leq \tau/2$. Similarly, r_i is able to observe all robot in distance $1 + \tau/2$ around r_k and can compute $\text{target}_k^{\mathcal{P}_\tau}(t)$ as well.

► **Lemma 23.** *Let \mathcal{P} be a λ -contracting gathering protocol with a viewing range of 1. A viewing range of $1 + \tau$ is sufficient to compute $\text{target}_k^{\mathcal{P}_\tau}(t)$ for all robots r_k within a radius of τ .*

15:20 Efficient (Near)-Gathering of Disoriented Robots with Limited Visibility

Secondly, r_i cannot avoid positions on all other $\text{colvec}_k^{\mathcal{P}_\tau}(t)$ in some cases. For instance, $\text{colvec}_i^{\mathcal{P}_\tau}(t)$ may be completely contained in $\text{colvec}_k^{\mathcal{P}_\tau}(t)$ (e.g., $\text{colvec}_2^{\mathcal{P}_\tau}(t) \in \text{colvec}_1^{\mathcal{P}_\tau}(t)$ in the example depicted in Figure 9). In case $\text{colvec}_i^{\mathcal{P}_\tau}(t)$ and $\text{colvec}_k^{\mathcal{P}_\tau}(t)$ are not collinear and intersect in a single point, both robots simply avoid the intersection point (e.g. r_1 and r_4 in the example).

► **Lemma 24.** *No robot moves to a point that is the intersection of two collision vectors that are not collinear.*

If $\text{colvec}_i^{\mathcal{P}_\tau}(t)$ and $\text{colvec}_k^{\mathcal{P}_\tau}(t)$ are collinear, both robots move to a point closer to their target point than to the other one (e.g., r_1 and r_3 in the example).

► **Lemma 25.** *If the target points of robots are different in \mathcal{P}_τ they are different in \mathcal{P}^{cl} .*

But there are cases, in which robots have the same target point in \mathcal{P}_τ (e.g. r_1, r_2 and r_6 in the example). Because robots stay in the same direction towards the target point, collisions can only happen if one robot is currently on the collision vector of another one (e.g., r_2 is on $\text{colvec}_1^{\mathcal{P}_\tau}(t)$). Their movement is scaled by the distance to the target point, which must be different. Therefore, their target points in \mathcal{P}^{cl} must be different as well.

► **Lemma 26.** *If the target points of robots are the same in \mathcal{P}_τ they are different in \mathcal{P}^{cl} .*

In \mathcal{SSYNC} robots may be inactive in one round. Nevertheless, in the same way, single intersection points between collision vectors and the positions of other robots are avoided as well.

► **Lemma 27.** *No robot moves to the position of an inactive robot.*

The following lemma follows immediately from Lemma 25, 26 and 27.

► **Lemma 28.** *The protocol \mathcal{P}^{cl} is collision-free.*

4.5 Time Bound

Previously, we have addressed the intermediate protocol \mathcal{P}_τ that is λ -contracting gathering protocol concerning the viewing range of $1 + \tau$ and also keeps $\text{UBG}(t)$ always connected. The same holds for \mathcal{P}^{cl} . Keeping $\text{UBG}(t)$ connected is important for the termination of a NEAR-GATHERING protocol. Suppose that $\text{UBG}(t)$ is connected and the robots only have a viewing range of 1. Then, the robots can never decide if they can see all the other robots. However, with a viewing range of $1 + \tau$, it becomes possible if the swarm is brought close together ($\text{diam}(t) < \tau$). For any configuration where the viewing range is $1 + \tau$ and $\text{UBG}(t)$ is connected, we state an important observation.

► **Lemma 29.** *Let \mathcal{P} be a λ -contracting protocol with viewing range $1 + \tau$ for a constant $\tau > 0$ and let $\text{UBG}(t)$ be connected. If $\text{diam}(t) > \tau$, then $\text{diam}_i(t) > \tau$, for every robot r_i .*

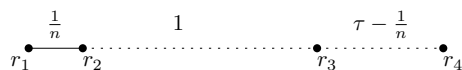
Due to the λ -contracting property, robots close to the boundary of the global smallest enclosing hypersphere (SEH) move upon activation at least $\Omega\left(\frac{\text{diam}_i(t)}{\Delta}\right)$ inwards. With $\text{diam}_i(t) > \tau$, it follows that the radius of the SEH decreases by $\Omega(\tau/\Delta)$ after each robot was active at least once (see Lemma 30). Consequently, $\text{diam}(t) \leq \tau$ after $\mathcal{O}(\Delta^2)$ epochs.

► **Lemma 30.** *Let \mathcal{P} be a λ -contracting protocol with a viewing range of $1 + \tau$ while $\text{UBG}(t)$ is always connected. After at most $\frac{32 \cdot \pi \cdot \Delta^2}{\lambda^2 \cdot \tau} \in \mathcal{O}(\Delta^2)$ epochs executing \mathcal{P} , $\text{diam}(t) \leq \tau$.*

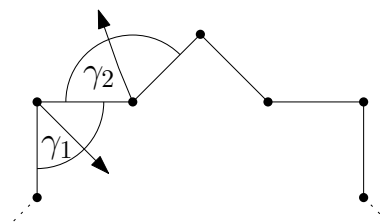
Because \mathcal{P}^{cl} has, regarding λ -contracting, connectivity and connectivity range, the same properties as \mathcal{P}_τ , this lemma can directly be applied to show the runtime of Theorem 20.

5 Conclusion & Future Work

In this work, we introduced the class of λ -contracting protocols and their collision-free extensions that solve GATHERING and NEAR-GATHERING of n robots located in \mathbb{R}^d in $\Theta(\Delta^2)$ epochs. While these results already provide several improvements over previous work, there are open questions that could be addressed by future research. First of all, we did not aim to optimize the constants in the runtime. Thus, the upper runtime bound of $\frac{256 \cdot \pi \cdot \Delta^2}{\lambda^3}$ seems to be improvable. Moreover, one major open question remains unanswered: is it possible to solve GATHERING or NEAR-GATHERING of oblivious and disoriented robots with limited visibility in $\mathcal{O}(\Delta)$ rounds? We could get closer to the answer: If there is such a protocol, it must compute target points regularly outside of the convex hulls of robots' neighborhoods. All λ -contracting protocols are slow in the configuration where the positions of the robots form a regular polygon with side length equal to the viewing range. In [15], it has been shown that this configuration can be gathered in time $\mathcal{O}(\Delta)$ by a protocol where each robot moves as far as possible along the angle bisector between its neighbors (leaving the local convex hull). However, this protocol cannot perform well in general. See Figure 11 for the *alternating star*, a configuration where this protocol is always worse compared to any protocol that computes target points inside of local convex hulls. Figure 11 gives a hint that every protocol that performs well for the regular polygon cannot perform equally well in the alternating star. Thus, we conjecture that $\Omega(\Delta^2)$ is a lower bound for every protocol that considers oblivious and disoriented robots with limited visibility.



■ **Figure 10** Example where target $P_i^P(t)$ is not λ -centered with respect to the viewing range $1 + \tau$.



■ **Figure 11** The robots at γ_1 observe a regular square, the robots at γ_2 a regular octagon. Given that each robot moves along the angle bisector between its neighbors and leaves its local convex hull, the radius of the global SEC decreases slower than in any λ -contracting protocol.

References

- 1 Sebastian Abshoff, Andreas Cord-Landwehr, Matthias Fischer, Daniel Jung, and Friedhelm Meyer auf der Heide. Gathering a closed chain of robots on a grid. In *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*, pages 689–699. IEEE Computer Society, 2016. doi:10.1109/IPDPS.2016.51.
- 2 Noa Agmon and David Peleg. Fault-Tolerant Gathering Algorithms for Autonomous Mobile Robots. *SIAM Journal on Computing*, 36(1):56–82, January 2006. doi:10.1137/050645221.
- 3 Hideki Ando, Yoshinobu Oasa, Ichiro Suzuki, and Masafumi Yamashita. Distributed memoryless point convergence algorithm for mobile robots with limited visibility. *IEEE Trans. Robotics Autom.*, 15(5):818–828, 1999. doi:10.1109/70.795787.
- 4 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*, Lecture Notes in Computer Science, pages 178–190, Cham, 2013. Springer International Publishing. doi:10.1007/978-3-319-03089-0_13.

- 5 Subhash Bhagat, Sruti Gan Chaudhuri, and Krishnendu Mukhopadhyaya. Fault-tolerant gathering of asynchronous oblivious mobile robots under one-axis agreement. *J. Discrete Algorithms*, 36:50–62, 2016. doi:10.1016/j.jda.2015.10.005.
- 6 Subhash Bhagat, Sruti Gan Chaudhuri, and Krishnendu Mukhopadhyaya. Fault-Tolerant Gathering of Asynchronous Oblivious Mobile Robots under One-Axis Agreement. In M. Sohel Rahman and Etsuji Tomita, editors, *WALCOM: Algorithms and Computation*, Lecture Notes in Computer Science, pages 149–160, Cham, 2015. Springer International Publishing. doi:10.1007/978-3-319-15612-5_14.
- 7 Subhash Bhagat and Krishnendu Mukhopadhyaya. Fault-tolerant Gathering of Semi-synchronous Robots. In *Proceedings of the 18th International Conference on Distributed Computing and Networking*, pages 1–10, Hyderabad India, January 2017. ACM. doi:10.1145/3007748.3007781.
- 8 Zohir Bouzid, Shantanu Das, and Sébastien Tixeuil. Gathering of Mobile Robots Tolerating Multiple Crash Faults. In *2013 IEEE 33rd International Conference on Distributed Computing Systems*, pages 337–346, July 2013. doi:10.1109/ICDCS.2013.27.
- 9 Zohir Bouzid, Maria Gradinariu Potop-Butucaru, and Sébastien Tixeuil. Byzantine Convergence in Robot Networks: The Price of Asynchrony. In Tarek Abdelzaher, Michel Raynal, and Nicola Santoro, editors, *Principles of Distributed Systems*, Lecture Notes in Computer Science, pages 54–70, Berlin, Heidelberg, 2009. Springer. doi:10.1007/978-3-642-10877-8_7.
- 10 Zohir Bouzid, Maria Gradinariu Potop-Butucaru, and Sébastien Tixeuil. Optimal Byzantine-resilient Convergence in Unidimensional Robot Networks. *Theoretical Computer Science*, 411(34-36):3154–3168, July 2010. doi:10.1016/j.tcs.2010.05.006.
- 11 Quentin Bramas and Sébastien Tixeuil. Wait-Free Gathering Without Chirality. In Christian Scheideler, editor, *Structural Information and Communication Complexity*, Lecture Notes in Computer Science, pages 313–327, Cham, 2015. Springer International Publishing. doi:10.1007/978-3-319-25258-2_22.
- 12 Philipp Brandes, Bastian Degener, Barbara Kempkes, and Friedhelm Meyer auf der Heide. Energy-efficient strategies for building short chains of mobile robots locally. *Theor. Comput. Sci.*, 509:97–112, 2013. doi:10.1016/j.tcs.2012.10.056.
- 13 Michael Braun, Jannik Castenow, and Friedhelm Meyer auf der Heide. Local gathering of mobile robots in three dimensions. 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 63–79. Springer, 2020. doi:10.1007/978-3-030-54921-3_4.
- 14 Jannik Castenow, Jonas Harbig, Daniel Jung, Peter Kling, Till Knollmann, and Friedhelm Meyer auf der Heide. A unifying approach to efficient (near)-gathering of disoriented robots with limited visibility. *CoRR*, abs/2206.07567, 2022.
- 15 Jannik Castenow, Jonas Harbig, Daniel Jung, Till Knollmann, and Friedhelm Meyer auf der Heide. Gathering a euclidean closed chain of robots in linear time. In Leszek Gasieniec, Ralf Klasing, and Tomasz Radzik, editors, *Algorithms for Sensor Systems - 17th International Symposium on Algorithms and Experiments for Wireless Sensor Networks, ALGOSENSORS 2021, Lisbon, Portugal, September 9-10, 2021, Proceedings*, volume 12961 of *Lecture Notes in Computer Science*, pages 29–44. Springer, 2021. doi:10.1007/978-3-030-89240-1_3.
- 16 Mark Cieliebak, Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro. Distributed Computing by Mobile Robots: Gathering. *SIAM J. Comput.*, 41(4):829–879, 2012. doi:10.1137/100796534.
- 17 Reuven Cohen and David Peleg. Convergence Properties of the Gravitational Algorithm in Asynchronous Robot Systems. *SIAM J. Comput.*, 34(6):1516–1528, 2005. doi:10.1137/S0097539704446475.
- 18 Reuven Cohen and David Peleg. Convergence of Autonomous Mobile Robots with Inaccurate Sensors and Movements. *SIAM J. Comput.*, 38(1):276–302, 2008. doi:10.1137/060665257.

- 19 Andreas Cord-Landwehr, Bastian Degener, Matthias Fischer, Martina Hüllmann, Barbara Kempkes, Alexander Klaas, Peter Kling, Sven Kurras, Marcus Märten, Friedhelm Meyer auf der Heide, Christoph Raupach, Kamil Swierkot, Daniel Warner, Christoph Weddemann, and Daniel Wonisch. A New Approach for Analyzing Convergence Algorithms for Mobile Robots. In Luca Aceto, Monika Henzinger, and Jirí Sgall, editors, *Automata, Languages and Programming*, Lecture Notes in Computer Science, pages 650–661, Berlin, Heidelberg, 2011. Springer. doi:10.1007/978-3-642-22012-8_52.
- 20 Andreas Cord-Landwehr, Matthias Fischer, Daniel Jung, and Friedhelm Meyer auf der Heide. Asymptotically optimal gathering on a grid. In Christian Scheideler and Seth Gilbert, editors, *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*, pages 301–312. ACM, 2016. doi:10.1145/2935764.2935789.
- 21 Shantanu Das, Paola Flocchini, Nicola Santoro, and Masafumi Yamashita. On the computational power of oblivious robots: forming a series of geometric patterns. In Andréa W. Richa and Rachid Guerraoui, editors, *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010*, pages 267–276. ACM, 2010. doi:10.1145/1835698.1835761.
- 22 Xavier Défago, Maria Gradinariu, Stéphane Messika, and Philippe Raipin-Parvédy. Fault-Tolerant and Self-stabilizing Mobile Robots Gathering. In Shlomi Dolev, editor, *Distributed Computing*, Lecture Notes in Computer Science, pages 46–60, Berlin, Heidelberg, 2006. Springer. doi:10.1007/11864219_4.
- 23 Bastian Degener, Barbara Kempkes, Peter Kling, and Friedhelm Meyer auf der Heide. Linear and competitive strategies for continuous robot formation problems. *ACM Trans. Parallel Comput.*, 2(1):2:1–2:18, 2015. doi:10.1145/2742341.
- 24 Bastian Degener, Barbara Kempkes, Tobias Langner, Friedhelm Meyer auf der Heide, Peter Pietrzyk, and Roger Wattenhofer. A tight runtime bound for synchronous gathering of autonomous robots with limited visibility. In Rajmohan Rajaraman and Friedhelm Meyer auf der Heide, editors, *SPAA 2011: Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Jose, CA, USA, June 4-6, 2011 (Co-located with FCRC 2011)*, pages 139–148. ACM, 2011. doi:10.1145/1989493.1989515.
- 25 Yoann Dieudonné, Franck Petit, and Vincent Villain. Leader election problem versus pattern formation problem. In Nancy A. Lynch and Alexander A. Shvartsman, editors, *Distributed Computing, 24th International Symposium, DISC 2010, Cambridge, MA, USA, September 13-15, 2010. Proceedings*, volume 6343 of *Lecture Notes in Computer Science*, pages 267–281. Springer, 2010. doi:10.1007/978-3-642-15763-9_26.
- 26 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.
- 27 Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro. Moving and computing models: Robots. 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 3–14. Springer, 2019. doi:10.1007/978-3-030-11072-7_1.
- 28 Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Giovanni Viglietta. Distributed computing by mobile robots: uniform circle formation. *Distributed Comput.*, 30(6):413–457, 2017. doi:10.1007/s00446-016-0291-x.
- 29 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.
- 30 Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Peter Widmayer. Arbitrary pattern formation by asynchronous, anonymous, oblivious robots. *Theor. Comput. Sci.*, 407(1-3):412–447, 2008. doi:10.1016/j.tcs.2008.07.026.

- 31 Taisuke Izumi, Zohir Bouzid, Sébastien Tixeuil, and Koichi Wada. The BG-simulation for Byzantine Mobile Robots, June 2011. doi:10.48550/arXiv.1106.0113.
- 32 Taisuke Izumi, Zohir Bouzid, Sébastien Tixeuil, and Koichi Wada. Brief Announcement: The BG-Simulation for Byzantine Mobile Robots. In David Peleg, editor, *Distributed Computing*, Lecture Notes in Computer Science, pages 330–331, Berlin, Heidelberg, 2011. Springer. doi:10.1007/978-3-642-24100-0_32.
- 33 Taisuke Izumi, Samia Souissi, Yoshiaki Katayama, Nobuhiro Inuzuka, Xavier Défago, Koichi Wada, and Masafumi Yamashita. The Gathering Problem for Two Oblivious Robots with Unreliable Compasses. *SIAM J. Comput.*, 41(1):26–46, 2012. doi:10.1137/100797916.
- 34 Heinrich Jung. Ueber die kleinste kugel, die eine räumliche figur einschliesst. *Journal für die reine und angewandte Mathematik*, 123:241–257, 1901. URL: <http://eudml.org/doc/149122>.
- 35 Heinrich Jung. Über den kleinsten kreis, der eine ebene figur einschließt. *Journal für die reine und angewandte Mathematik*, 137:310–313, 1910. URL: <http://eudml.org/doc/149324>.
- 36 Branislav Katreniak. Convergence with Limited Visibility by Asynchronous Mobile Robots. In Adrian Kosowski and Masafumi Yamashita, editors, *Structural Information and Communication Complexity - 18th International Colloquium, SIROCCO 2011, Gdansk, Poland, June 26-29, 2011. Proceedings*, volume 6796 of *Lecture Notes in Computer Science*, pages 125–137. Springer, 2011. doi:10.1007/978-3-642-22212-2_12.
- 37 David G. Kirkpatrick, Irina Kostitsyna, Alfredo Navarra, Giuseppe Prencipe, and Nicola Santoro. Separating Bounded and Unbounded Asynchrony for Autonomous Robots: Point Convergence with Limited Visibility. 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 9–19. ACM, 2021. doi:10.1145/3465084.3467910.
- 38 Shouwei Li, Christine Markarian, Friedhelm Meyer auf der Heide, and Pavel Podlipyan. A continuous strategy for collisionless gathering. *Theor. Comput. Sci.*, 852:41–60, 2021. doi:10.1016/j.tcs.2020.10.037.
- 39 Ji Lin, A. Stephen Morse, and Brian D. O. Anderson. The multi-agent rendezvous problem. part 1: The synchronous case. *SIAM J. Control. Optim.*, 46(6):2096–2119, 2007. doi:10.1137/040620552.
- 40 Ji Lin, A. Stephen Morse, and Brian D. O. Anderson. The multi-agent rendezvous problem. part 2: The asynchronous case. *SIAM J. Control. Optim.*, 46(6):2120–2147, 2007. doi:10.1137/040620564.
- 41 Linda Pagli, Giuseppe Prencipe, and Giovanni Viglietta. Getting close without touching. In Guy Even and Magnús M. Halldórsson, editors, *Structural Information and Communication Complexity - 19th International Colloquium, SIROCCO 2012, Reykjavik, Iceland, June 30-July 2, 2012, Revised Selected Papers*, volume 7355 of *Lecture Notes in Computer Science*, pages 315–326. Springer, 2012. doi:10.1007/978-3-642-31104-8_27.
- 42 Linda Pagli, Giuseppe Prencipe, and Giovanni Viglietta. Getting close without touching: near-gathering for autonomous mobile robots. *Distributed Comput.*, 28(5):333–349, 2015. doi:10.1007/s00446-015-0248-5.
- 43 Debasish Pattanayak, Kaushik Mondal, H. Ramesh, and Partha Sarathi Mandal. Fault-Tolerant Gathering of Mobile Robots with Weak Multiplicity Detection. In *Proceedings of the 18th International Conference on Distributed Computing and Networking*, pages 1–4, Hyderabad India, January 2017. ACM. doi:10.1145/3007748.3007786.
- 44 Pavan Poudel and Gokarna Sharma. Time-optimal gathering under limited visibility with one-axis agreement. *Inf.*, 12(11):448, 2021. doi:10.3390/info12110448.
- 45 Giuseppe Prencipe. Impossibility of gathering by a set of autonomous mobile robots. *Theor. Comput. Sci.*, 384(2-3):222–231, 2007. doi:10.1016/j.tcs.2007.04.023.
- 46 Ichiro Suzuki and Masafumi Yamashita. Distributed Anonymous Mobile Robots: Formation of Geometric Patterns. *SIAM J. Comput.*, 28(4):1347–1363, 1999. doi:10.1137/S009753979628292X.

- 47 Masafumi Yamashita and Ichiro Suzuki. Characterizing geometric patterns formable by oblivious anonymous mobile robots. *Theor. Comput. Sci.*, 411(26-28):2433–2453, 2010. doi:10.1016/j.tcs.2010.01.037.
- 48 Yukiko Yamauchi, Taichi Uehara, and Masafumi Yamashita. Brief announcement: Pattern formation problem for synchronous mobile robots in the three dimensional euclidean space. 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 447–449. ACM, 2016. doi:10.1145/2933057.2933063.
- 49 Yukiko Yamauchi and Masafumi Yamashita. Pattern Formation by Mobile Robots with Limited Visibility. In Thomas Moscibroda and Adele A. Rescigno, editors, *Structural Information and Communication Complexity*, Lecture Notes in Computer Science, pages 201–212, Cham, 2013. Springer International Publishing. doi:10.1007/978-3-319-03578-9_17.

New Dolev-Reischuk Lower Bounds Meet Blockchain Eclipse Attacks

Ittai Abraham ✉

VMWare Research, Herzliya, Israel

Gilad Stern ✉

The Hebrew University of Jerusalem, Israel

Abstract

In 1985, Dolev and Reischuk proved a fundamental communication lower bounds on protocols achieving fault tolerant synchronous broadcast and consensus: any deterministic protocol solving those tasks (even against omission faults) requires at least a quadratic number of messages to be sent by nonfaulty parties. In contrast, many blockchain systems achieve consensus with seemingly linear communication per instance against Byzantine faults. We explore this dissonance in three main ways. First, we extend the Dolev-Reischuk family of lower bounds and prove a new lower bound for Crusader Broadcast protocols. Our lower bound for crusader broadcast requires non-trivial extensions and a much stronger Byzantine adversary with the ability to simulate honest parties. Secondly, we extend our lower bounds to all-but- m Crusader Broadcast, in which up to m parties are allowed to output a different value. Finally, we discuss the ways in which these lower bounds relate to the security of blockchain systems. We show how *Eclipse-style attacks* in such systems can be viewed as specific instances of the attacks used in our lower bound for Crusader Broadcast. This connection suggests a more systematic way of analyzing and reasoning about Eclipse-style attacks through the lens of the Dolev-Reischuk family of attacks.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases consensus, crusader broadcast, Byzantine fault tolerance, blockchain, synchrony, lower bounds

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2022.16

Related Version *Full Version*: <https://eprint.iacr.org/2022/730>

Funding *Gilad Stern*: This work was supported by the HUJI Federmann Cyber Security Research Center in conjunction with the Israel National Cyber Directorate (INCD) in the Prime Minister's Office.

1 Introduction

Two of the foundational and highly related tasks in the world of distributed systems are *consensus*, and *broadcast*. In a consensus protocol, all parties have some input and they must agree on an output. On the other hand, in a broadcast protocol, a designated sender attempts to send a specific message to all parties, and all parties must output the same message sent by the sender. These tasks have been widely researched both in theoretic settings and practical settings. Ideally, we would like to be able to design efficient protocols for solving these tasks in the presence of faults. A foundational limit on the efficiency of such protocols is the work of Dolev and Reischuk in 1985 [9]. They prove that any deterministic protocol solving fault tolerant broadcast must send at least $\Omega(n \cdot f)$ messages, where n is the number of parties overall and f is the number of omission-faulty parties, whose incoming and outgoing messages can be dropped¹. Since broadcast and consensus reduce to each other [6], the lower

¹ the lower bound in [9] mentioned malicious adversaries, the extension to omission failures appears in [3].



bound also provides a lower bound on consensus. Hadzilacos and Halpern [14] show that a similar lower bound also holds only when considering fault-free runs of broadcast protocols if they are designed to be resilient to faults. Abraham, Chun, Dolev, Nayak, Pass, Ren, and Shi [1] generalized this work to probabilistic protocols, showing that with f Byzantine faults, a broadcast protocol with a $\frac{3}{4} + \epsilon$ probability of success requires $\Omega(\epsilon n f)$ messages to be sent in expectation (assuming a strongly adaptive adversary).

1.1 Dolev-Reischuk does not hold for Crusader Broadcast

A slightly relaxed task related to that of broadcast is the task of *Crusader Broadcast* [8], in which parties are allowed to remain undecided when the sender is faulty. This is formalized by allowing parties to output a special non-value, \perp . Two important restrictions, in this case, are that no two nonfaulty parties may output different non- \perp values and that all nonfaulty parties must output the sender's input if it is nonfaulty. The known Dolev-Reischuk style attacks heavily rely on the fact that parties have to output some value from the protocol, regardless of what they see. This is utilized by completely isolating a party, forcing it to communicate only with omission faulty parties. The adversary then simply blocks all communication with the isolated party, forcing it to output some value without hearing anything throughout the protocol. All that is left to do is make sure that other parties output the other value, successfully attacking any protocol with low communication complexity. However, in Crusader Broadcast, a nonfaulty party is allowed to output \perp if it hears nothing throughout the protocol. Since a nonfaulty sender may send messages to any party without reaching quadratic communication complexity, it is entirely possible that in any run in which the sender is nonfaulty, no party can be completely isolated from nonfaulty parties in the protocol. This implies that the Dolev-Reischuk lower bound attack does not hold as is for Crusader broadcast protocols.

1.2 A New Lower Bound for Crusader Broadcast

The main contribution of this paper is a new lower bound for crusader broadcast. It differs substantially from the classic Dolev-Reischuk lower bound in that the adversary is required to **actively** corrupt parties. Using Byzantine corruption also raises a new challenge that is similar to that of the lower bound proven by Fischer, Lynch and Meritt [12]: the corrupted parties need to be able to **simulate** honest parties.

Intuitively, while classic Dolev-Reischuk isolates one party and makes it hear no message at all, while other parties hear a sender sending say v , in our lower bound we isolate one party and make it think it is living in an alternative world where the sender is sending $v' \neq v$. Again intuitively, building an alternative universe is harder since it requires active simulation of other parties, and thus requires more malice than just causing the isolated party to hear nothing.

This type of attack (isolating a node and making it think it's living in an alternative world) is not just theoretical, we discuss in Section 5 how Eclipse-type attacks can be viewed as types of this attack. We find this connection between a theoretical lower bound and Eclipse-style blockchain attacks to be a conceptual contribution of its own and expand on this in section 1.4.

1.3 Extending to the all-but-m model

Yet another problem that the classic Dolev-Reischuk lower bound does not cover is **almost everywhere agreement** [10, 17]. This notion is closely related to that of classic agreement protocols but allows a small number of nonfaulty parties to output the wrong value, as long

as the percentage of those dissenting parties tends towards 0 as n tends towards infinity. In particular, classic Dolev-Reischuk just shows a safety violation of one party. What if we allow some fraction of the parties to dissent and output differing values? Does some variant of Dolev-Reischuk hold in this case?

In this work, we prove that if the number of messages sent is significantly smaller than quadratic, then a large number of parties can be made to disagree. Concretely, if the number of messages sent is $O(nf^{1-c})$ for some $c \in [0, 1]$, the number of parties that can be made to output a differing value is $O(f^c)$. Crucially, as the number of messages in the protocol approaches $O(n)$, the adversary can make $O(f)$ nonfaulty parties disagree, which is often taken to be a constant fraction of the total number of parties.

1.4 Why does Crusader Broadcast matter for Blockchains? Connections to Eclipse Style Attacks

Blockchain systems are designed to solve the task of consensus [21, 24], or more precisely, state machine replication. Many of these systems claim to achieve consensus in a **linear** number of messages per block. This seems to be in direct conflict with the lower bounds of Dolev-Reischuk, suggesting that at least **quadratic** ($\Omega(nf)$) messages are required. One way of making sense of this contradiction is by looking at the details of the lower bounds. As discussed above, Dolev-Reischuk prove such lower bounds for protocols in which parties are required to output some value eventually, even without hearing any message. In Bitcoin and Ethereum (Nakamoto consensus based systems) not hearing new blocks simply causes parties to not decide anything. Formally, we can think of this as if such a party outputs a special value \perp , signifying not knowing. In this sense, it is natural to actually view such blockchain protocols, when focusing on a single block, as **solving Crusader Broadcast rather than consensus**.

In Section 5, we explore the connections between our new lower bounds for crusader broadcast and the eclipse style attacks in proof-of-work blockchain systems. We observe the following similarity:

- In our lower bound for a single shot of crusader broadcast, in the deterministic case the adversary can be static to isolate a party and needs to be able to simulate other parties to trick the isolated party to believe an alternative world.
- Similarly, in Eclipse-style attacks, the systems had insufficient randomness at the network layer, meaning that the communication graph induced by the protocol was either deterministic, or very easy to influence. This allowed a relatively static adversary to isolate a party, or even worse. Similarly, by the nature of Nakamoto consensus, the adversary can simulate other parties (often for a limited time) and trick the isolated party to believe an alternative world.

Note that in both cases, the isolated party believes in an alternative world and a double spend is executed. This is unlike the classic Dolev-Reischuk lower bound where the isolated node just sees silence (so there is just one spend and one party that does not observe the spend). Finally, we discuss in Section 6 the consequence of our lower bound for the study of upper bounds for crusader broadcast. In particular, we show how a subquadratic protocol for crusader broadcast takes advantage of randomization and cryptography in order to circumvent the $\Omega(n^2)$ lower bound.

Our contributions

To summarize, our work makes three main contributions:

1. A new Lower Bound for crusader broadcast. While it is definitely part of the enhanced Dolev-Reischuk family, it requires new non-trivial extensions. In particular, Byzantine adversaries and the ability to simulate.
 - **Theorem 4.** *Let there be a deterministic protocol solving Binary Crusader Broadcast in lockstep synchrony. If the protocol is resilient to f static Byzantine corruptions, then there must be at least one run of the protocol in which at least $\frac{1}{4}(n-1)f$ messages are sent for $n \geq f+2$.*
2. We extend the Dolev-Reischuk style lower bounds to the **all but m model** showing that near linear protocols may actually suffer a near linear number of isolated parties. Similar to how Abraham *et al.* [1] extend the Dolev-Reischuk lower bound to the randomized setting given a strongly adaptive adversary, we also extended our lower bound on Crusader Broadcast to the randomized setting given a strongly adaptive adversary in Theorem 5.
 - **Theorem 6.** *Let there be a probabilistic $(\frac{2}{3} + \epsilon)$ -correct protocol solving all-but $(f^c - 1)$ Binary Crusader Broadcast in lockstep synchrony for some $c \in [0, 1]$ and $\epsilon \in (0, \frac{1}{3}]$. If the protocol is resilient to f strongly adaptive Byzantine corruptions, then the expected number of messages sent in the protocol is at least $\frac{\epsilon}{8}(n-1)f^{1-c}$ for $n \geq 3f$.*
3. Make the conceptual connection between Eclipse style attacks and our new Crusader broadcast lower bound. We believe that by highlighting this connection, protocol designers may be able to more rigorously design blockchain protocols that are more secure against Eclipse-style attacks.

2 Communication and Adversary Model

We consider a fully-connected network of n parties with synchronous communication: there is a commonly known bound Δ on message delay. The adversary can choose exactly how long each message is delayed within the range $[0, \Delta]$. The lower bounds hold in an even stronger synchrony assumption: lockstep communication where communication proceeds in lockstep rounds.

We assume a Byzantine adversary that can corrupt up to f parties. We consider both static and strongly adaptive adversaries. A static Byzantine adversary must choose which parties to corrupt at the beginning of the protocol. A strongly adaptive adversary can choose which parties to corrupt at any given time. Furthermore, it can even choose to corrupt parties after they send messages, but before they are delivered. If it chooses to do so, it can delete those messages and send different messages instead. We assume a computationally unbounded adversary that can simulate other parties if required. When the adversary is computationally limited, we explicitly mention this.

3 Definitions

The Binary Crusader Broadcast task is very similar to the Binary Broadcast task, except parties are also allowed to output \perp if the sender is faulty. Formally, such a protocol is defined as follows:

► **Definition 1.** *A Binary Crusader Broadcast protocol has a designated sender s with some input $x \in \{0, 1\}$. Every party outputs some value $y_i \in \{0, 1, \perp\}$. A protocol solving Binary Crusader Broadcast has the following properties:*

- **Validity.** *If the sender is nonfaulty, then every nonfaulty party outputs x .*
- **Correctness.** *If two nonfaulty parties i, j output $y_i, y_j \in \{0, 1\}$, then $y_i = y_j$.*
- **Termination.** *If all nonfaulty parties participate in the protocol, they all complete it.*

A weaker version of the Binary Crusader Broadcast task is the almost-everywhere Binary Crusader Broadcast protocol, similar to the almost-everywhere agreement problem [10, 23]. Whereas in a regular Binary Crusader Broadcast protocol all parties that don't output \perp must output the same value even when the sender is faulty, in an almost-everywhere Binary Crusader Broadcast protocol a small number of parties are allowed to output a different value when the sender is faulty. A protocol solving Binary Crusader allowing m parties to disagree is called an all-but m Binary Crusader protocol, and is defined as follows:

► **Definition 2.** *An all-but m Binary Crusader Broadcast protocol has a designated sender s with some input $x \in \{0, 1\}$. Every party outputs some value $y_i \in \{0, 1, \perp\}$. A protocol solving all-but m Binary Crusader Broadcast has the following properties:*

- **Validity.** *If the sender is nonfaulty, then every nonfaulty party outputs x .*
- **Correctness.** *There exists some $y \in \{0, 1\}$ such that at most m nonfaulty parties i output $y_i \notin \{y, \perp\}$.*
- **Termination.** *If all nonfaulty parties participate in the protocol, they all complete it.*

Note that an all-but 0 Binary Crusader Broadcast protocol is simply a Binary Crusader Broadcast protocol. A protocol is said to be deterministic if all nonfaulty parties' actions are chosen as a deterministic function of their input and the messages they receive, and probabilistic otherwise. A protocol is said to be p -correct if for any adversary all of its properties hold with probability p or greater.

3.1 Relationship to Crusader Consensus

The notion of Crusader Broadcast is highly related to that of Crusader Consensus. We define the task of Crusader Consensus as follows:

► **Definition 3.** *In a Binary Crusader Consensus protocol, every party i has an input $x_i \in \{0, 1\}$. Every party outputs some value $y_i \in \{0, 1, \perp\}$. A protocol solving Binary Crusader Consensus has the following properties:*

- **Validity.** *If all nonfaulty parties have the same input x , then they all output x .*
- **Correctness.** *If two nonfaulty parties i, j output $y_i, y_j \in \{0, 1\}$, then $y_i = y_j$.*
- **Termination.** *If all nonfaulty parties participate in the protocol, they all complete it.*

These tasks reduce to each other in the same way regular consensus and broadcast reduce to each other when $n \geq 2f + 1$ with f Byzantine parties [6]. In short, assume we have a Crusader Broadcast protocol. In order to achieve Crusader Consensus, every party broadcasts its input x_i and waits to complete all broadcasts. After completing all n broadcasts, if there exists some value y which was received in at least $n - f$ broadcasts, output y . Otherwise output \perp . If all nonfaulty parties have the same input x , then from the Validity property they will receive that value in at least the $n - f$ broadcasts with nonfaulty senders and output it. On the other hand, if some nonfaulty party outputs a value $y \neq \perp$, then it received it in at least $n - f$ broadcasts. From the Correctness property, every other party either outputs y or \perp in those broadcasts, and thus can output some y' such that $y' \notin \{y, \perp\}$ only in the f remaining broadcasts. We know that $n \geq 2f + 1$, and thus $f \leq n - (f + 1) < n - f$, which means that it won't output $y' \notin \{y, \perp\}$, as required.

In the other direction, assume that there exists a Crusader Consensus protocol. In order to implement broadcast, the sender sends its input to all parties. Each party that doesn't receive a value within Δ time chooses a default value, e.g. 0. They then all participate in the Crusader Consensus protocol with their received value as input, and output their output from the Crusader Consensus protocol. If the sender is nonfaulty with the input x , then all nonfaulty parties will receive that value in Δ time. They then all participate the Crusader Consensus protocol with the input x , and therefore from the Validity property output x as well. In addition, if two nonfaulty parties output $y, y' \in \{0, 1\}$, then from the Correctness property $y = y'$, as required.

4 Lower Bounds

This section provides several communication lower bounds on protocols solving Binary Crusader Broadcast. The lower bounds use ideas from the Dolev-Reischuk lower bound [9], and from the subsequent work of Abraham *et al.* [1]. The lower bounds presented in Theorem 4 and Theorem 5 isolate a party in a similar way to the ones described in [9] and [1] respectively. However, unlike previous works, the lower bounds presented in this paper require the adversary to act in a Byzantine manner and actively simulate other parties. Theorem 6 shows how previously known techniques can be used to isolate a large number of parties, and cause wide disagreement in the network. All of the following lower bounds are stated as lower bounds on the number of messages sent, as done in previous works. However, the lower bounds actually only use the number of messages as a bound on the number edges in the communication graph. That is, if fewer than m messages are sent in the network, then there are fewer than m pairs of parties that communicate with each other. These lower bounds cannot be avoided by increasing the number of messages without increasing the number of edges in the communication graph of the protocol. This means that the lower bounds might be more accurately stated in terms of edges in the communication graph instead of messages sent.

The first lower bound uses the fact that few messages are sent in order to isolate a single party i and cause it to communicate only with faulty parties. The faulty parties then simulate a run with the input 1 for party i when a nonfaulty sender has the input 0, causing it to output 1. The faulty parties also make sure that the rest of the network doesn't notice that i was isolated by simulating its messages in a run with the sender's input being 0 and having the faulty parties respond accordingly when communicating with other parties. Note that in order for the theorem to hold, the content of the messages actually can be probabilistic, as long as parties always communicate with the same parties throughout the protocol. All of the following bounds also include an upper bound on the number of faulty parties. This is done in order to make sure that the required number of nonfaulty parties remain in order to reach a contradiction. Clearly, if the adversary can actually corrupt a larger number of parties, it can choose not to do so and achieve the same lower bounds, slightly adjusting the exact number of messages. In all cases however, f is allowed to be a constant fraction of n .

► **Theorem 4.** *Let there be a deterministic protocol solving Binary Crusader Broadcast in lockstep synchrony. If the protocol is resilient to f static Byzantine corruptions, then there must be at least one run of the protocol in which at least $\frac{1}{4}(n-1)f$ messages are sent for $n \geq f+2$.*

Proof. Assume by way of contradiction that fewer than $\frac{1}{4}(n-1)f$ messages are sent overall throughout any run of the protocol. Let W_0 be a run in which the adversary does not corrupt any party and the sender has input 0. Similarly, let W_1 be a run in which the adversary does

not corrupt any party and the sender has input 1. From the Validity property of the protocol all parties output 0 in W_0 and 1 in W_1 . By assumption, the total number of messages sent in either run is less than $\frac{1}{4}(n-1)f$, and thus the total number of messages in both runs is less than $\frac{1}{2}(n-1)f$. Now assume by way of contradiction that at least $n-1$ parties send or receive at least f messages in total in both runs. When summing over the messages sent or received by all parties, each message is counted twice: once when it is sent and once when it is received. Therefore, the total number of messages sent in both W_0 and W_1 is at least $\frac{1}{2}(n-1)f$, reaching a contradiction to the stated above. This means that at least 2 parties send or receive no more than f messages in total in both runs. Let i be one of those parties such that i is not the sender s . Let P_0, P_1 be the sets of parties with which i communicated in W_0 and W_1 respectively. By the stated above, $|P_0 \cup P_1| \leq f$.

Now observe the run W_{hybrid} in which s has the input 0 and the adversary acts according to the following strategy: the adversary corrupts all parties in $P_0 \cup P_1$, all of those parties communicate with all parties that aren't i as nonfaulty parties would in the protocol, and communicate with i as nonfaulty parties would if s had the input 1. More precisely, parties in $P_0 \cup P_1$ simulate all of i 's messages internally when communicating with all parties other than i and act as if they received those messages, but don't send resulting messages to i . On the other hand, when communicating with i they simulate all of the messages from all other parties with a nonfaulty s having the input 1 and act accordingly, but only send resulting messages to party i . Note that both in W_0 and in W_1 , all parties not in $P_0 \cup P_1 \cup \{i\}$ don't communicate directly with party i . All nonfaulty parties see communication that is identical to the one in W_0 and since they are not in P_0 , they don't send any messages to i in W_{hybrid} as well. Similarly, i sees communication that is identical to the one in W_1 and thus doesn't send any messages to parties other than those in P_1 in W_{hybrid} . Therefore the view of all parties not in $P_0 \cup P_1 \cup \{i\}$ is identical to their view in W_0 , and thus as stated above, they all output 0. On the other hand, i 's view is identical to its view in W_1 , and thus it outputs 1. Note that $n \geq f + 2$, so there are at least two nonfaulty parties. Party i and all parties not in $P_0 \cup P_1 \cup \{i\}$ are nonfaulty, so this is a violation of the Correctness property of the protocol, reaching a contradiction and completing the proof. ◀

The second lower bound uses ideas from [1] and generalizes them to the task of probabilistic Crusader Broadcast. The first part of the lower bound shows that if no more than $\frac{\epsilon}{4}f^2$ messages are sent in expectation in the protocol, then there is at least one non-sender party that communicates with a small number of parties with probability ϵ . Using this insight, an adversary can isolate that party and perform a similar attack to the one described in the previous theorem. The last part of the theorem shows that the probability that if the original protocol is purported to be $(\frac{2}{3} + \epsilon)$ -correct, then the isolated party and the rest of the nonfaulty parties output different values with at least $\frac{1}{3}$ probability, reaching a contradiction. Recall that as defined in Section 3, a protocol is said to be p -correct if its properties hold with probability p or greater.

► **Theorem 5.** *Let there be a probabilistic $(\frac{2}{3} + \epsilon)$ -correct protocol solving Binary Crusader Broadcast in lockstep synchrony for some $\epsilon \in (0, \frac{1}{3}]$. If the protocol is resilient to f strongly adaptive Byzantine corruptions, then the expected number of messages sent in the protocol is at least $\frac{\epsilon}{4}(n-1)f$ for $n \geq f + 2$.*

Proof. Assume that is not the case. This means that there exists a $(\frac{2}{3} + \epsilon)$ -correct Binary Crusader Broadcast protocol with expected message complexity smaller than $\frac{\epsilon}{4}(n-1)f$. Similarly to the previous theorem, we will define W_0 and W_1 as runs in which the adversary does not corrupt any party and the sender has inputs 0 and 1 respectively. In both of these

runs, the probability that all parties terminate and output the sender's input must be at least $\frac{2}{3} + \epsilon$. Define M_0 and M_1 to be random variables indicating the number of messages sent by nonfaulty parties in W_0 and W_1 respectively. In addition, define $M = M_0 + M_1$ to be the number of messages sent in both runs. By assumption, $\mathbb{E}[M] = \mathbb{E}[M_0] + \mathbb{E}[M_1] < \frac{\epsilon}{2}(n-1)f$. For every $i \in [n]$, let X_i be a random variable indicating the total number of messages sent or received by party i in total both in W_0 and in W_1 . Assume by way of contradiction that for at least $n-1$ parties $i \in [n]$, $\mathbb{E}[X_i] > \epsilon f$. First note that $M = \frac{1}{2} \sum_{i=1}^n X_i$ because when summing over all the messages that each party sent and received, we count every message twice. Therefore, $\mathbb{E}[M] = \frac{1}{2} \sum_{i=1}^n \mathbb{E}[X_i] > \frac{\epsilon}{2}(n-1)f$, in contradiction. Therefore, there exist at least two parties $i, j \in [n]$ for which $\mathbb{E}[X_i], \mathbb{E}[X_j] \leq \epsilon f$. Let i be a non-sender party for which $\mathbb{E}[X_i] \leq \epsilon f$. From the Markov inequality, $\Pr[X_i \geq f] \leq \frac{\mathbb{E}[X_i]}{f} \leq \frac{\epsilon f}{f} = \epsilon$.

We will now define an adversary's attack in W_{hybrid} . The sender s has the input 0, and the adversary will attempt to cause i to output 1 while other parties output 0. Informally, the adversary's strategy is to corrupt all parties that communicate with i (either by sending or receiving messages) throughout the run and delete all messages to i . The adversary then simulates i 's responses to the messages it would have received, corrupts the parties that would have received those messages and causes them to act as if they received those messages. In addition, the adversary simulates a full run in W_1 , and whenever a party sends a message to i in its simulation, it corrupts that party and causes it to send that message to i . This causes i to think it is in W_0 and all other parties to think they are in W_1 , causing them to output different values.

More formally, whenever a party j sends a message to party i , the adversary corrupts j and erases the message. In addition, whenever i sends a message to some party k , the adversary corrupts k . In parallel, the adversary simulates party i 's responses in W_0 , given all of the messages it was sent. If party i ever sends a message to party j in that simulation in a given round, the adversary corrupts party j , erases its outgoing messages for that round, and makes it act as a nonfaulty party would if it received all of the messages it already received and the messages sent by i in the simulated run. Finally, the adversary simulates all of the communication between all parties in W_1 given the messages sent by i in W_{hybrid} . This is done by internally running all parties in each round of the protocol except i , and using i 's messages in each round. Whenever a party k sends i a message in the simulated run of W_1 , the adversary corrupts it in W_{hybrid} and sends that message to i . If at any point the adversary is required to corrupt more than f parties, it aborts. Before analyzing the probability that the attack succeeds, we will define several random variables. Let A_0 be the event that all nonfaulty parties except i output 0 in W_{hybrid} . Let A_1 be the event that i outputs 1 in W_{hybrid} . Similarly, let B_0 be the event that all nonfaulty parties except i output 0 in W_0 , and let B_1 be the event that i outputs 1 in W_1 . Note that the definitions of A_0, B_0 allows i to output 0 as long as all other nonfaulty parties output 0. Define G to be the event that no more than f parties communicate with i in total in W_0 and W_1 combined. Finally, define G_{hybrid} to be the event that the adversary does not abort in W_{hybrid} .

Our goal is to show that $\Pr[A_0 \cap A_1] > \frac{1}{3} - \epsilon$. This contradicts the fact that the protocol is $(\frac{2}{3} + \epsilon)$ -correct, because with more than $\frac{1}{3} - \epsilon$ probability, all honest parties except for i output 0, and i outputs 1. By assumption $n \geq f + 2$, so there actually are at least two nonfaulty parties. Before doing so, note that as long as the adversary isn't required to corrupt more than f parties, the view of all nonfaulty parties except i in W_{hybrid} is identical to the view they would have in W_0 , given that no more than f parties communicate with i in both W_0 and W_1 . Similarly, as long as that event doesn't happen, i 's view is identical

to the view it would have in W_1 , given that no more than f parties communicate with i in both W_0 and W_1 . Therefore, we know that $\Pr[G] = \Pr[G_{hybrid}]$, $\Pr[A_0|G_{hybrid}] = \Pr[B_0|G]$ and $\Pr[A_1|G_{hybrid}] = \Pr[B_1|G]$. We are now ready to analyze $\Pr[A_0 \cap A_1]$:

$$\begin{aligned}
\Pr[A_0 \cap A_1] &= \Pr[A_0] + \Pr[A_1] - \Pr[A_0 \cup A_1] \\
&\geq \Pr[G_{hybrid}] (\Pr[A_0|G_{hybrid}] + \Pr[A_1|G_{hybrid}]) - 1 \\
&= \Pr[G] (\Pr[B_0|G] + \Pr[B_1|G]) - 1 \\
&= \Pr[B_0 \wedge G] + \Pr[B_1 \wedge G] - 1 \\
&= \Pr[B_0] - \Pr[B_0 \wedge \overline{G}] + \Pr[B_1] - \Pr[B_1 \wedge \overline{G}] - 1 \\
&\geq \Pr[B_0] + \Pr[B_1] - 2\Pr[\overline{G}] - 1 \\
&= \Pr[B_0] + \Pr[B_1] - 2\Pr[X_i > f] - 1 \\
&\geq \left(\frac{2}{3} + \epsilon\right) + \left(\frac{2}{3} + \epsilon\right) - 2\epsilon - 1 = \frac{1}{3} > \frac{1}{3} - \epsilon,
\end{aligned}$$

reaching a contradiction, and completing the proof. \blacktriangleleft

The main insight of the previous theorem was that if fewer than $\Omega(nf)$ messages are sent in a protocol in expectation, then there is a good probability that at least one party communicates with f parties or fewer, and can be isolated. The next lower bound generalizes this insight and shows that if for some $c \in [0, 1]$ fewer than $\Omega(nf^{1-c})$ messages are sent in expectation, there exist f^c parties that can be isolated. From this point, the proof is extremely similar to the one of the previous theorem. Note that the exact same techniques can be used in the deterministic case with a static adversary, but the theorem is omitted due to its similarity. It is also important to note that similar theorems with different choices instead of $f^c - 1$ can easily be formulated for more general results. This specific choice was made as it simplifies some calculations, and it is enough to show that as the number of messages approaches a $O(\epsilon n)$, the number of isolated parties approaches $\Omega(f)$.

► Theorem 6. *Let there be a probabilistic $(\frac{2}{3} + \epsilon)$ -correct protocol solving all-but $(f^c - 1)$ Binary Crusader Broadcast in lockstep synchrony for some $c \in [0, 1]$ and $\epsilon \in (0, \frac{1}{3}]$. If the protocol is resilient to f strongly adaptive Byzantine corruptions, then the expected number of messages sent in the protocol is at least $\frac{\epsilon}{8}(n-1)f^{1-c}$ for $n \geq 3f$.²*

Proof. Assume that is not the case. This means that there exists a $(\frac{2}{3} + \epsilon)$ -correct all-but $(f^c - 1)$ Binary Crusader Broadcast protocol with expected message complexity smaller than $\frac{\epsilon}{8}(n-1)f^{1-c}$. Similarly to the previous theorem, we will define W_0 and W_1 as runs in which the adversary does not corrupt any party and the sender has inputs 0 and 1 respectively. In both of these runs, the probability that all parties terminate and output the sender's input must be at least $\frac{2}{3} + \epsilon$. Define M_0 and M_1 to be random variables indicating the number of messages sent by nonfaulty parties in W_0 and W_1 respectively. In addition, define $M = M_0 + M_1$ to be the number of messages sent in both runs. By assumption, $\mathbb{E}[M] = \mathbb{E}[M_0] + \mathbb{E}[M_1] < \frac{\epsilon}{4}(n-1)f^{1-c}$. Similarly to before, the adversary will seek a set of $\lfloor f^c \rfloor > f^c - 1$ parties that don't contain the sender and don't send many messages. In order to do that, assume without loss of generality that the sender is party n . Let $m = \lfloor f^c \rfloor$, $\ell = \lceil \frac{n-1}{m} \rceil$, and define ℓ sets of m parties as follows: $\forall i \in \{0, \dots, \ell-2\} P_i = \{i \cdot m + 1, \dots, (i+1)m\}$ and

² It is actually enough that $n \geq f + 2f^c$, since all we need is f faulty parties and 2 sets of at least f^c nonfaulty parties to disagree on the output.

$P_{\ell-1} = \{n-m, \dots, n-1\}$. We would like to guarantee that the sender is not in any of the sets P_i , and that every other party appears in one of the sets, but in no more than two of the sets. First note that the sender is not in $P_{\ell-1}$ by definition. The largest number in any of the other P_i sets is $(\ell-2+1)m$. Using the definition of ℓ , $(\ell-2+1)m = (\lceil \frac{n-1}{m} \rceil - 1)m \leq \frac{n-1}{m} \cdot m < n$, and thus the sender (party n) is not in any of those sets. Secondly, note that all of the sets up to $P_{\ell-2}$ are disjoint. This means that every party appears at most once in one of the sets $P_0, \dots, P_{\ell-2}$ and at most once more in $P_{\ell-1}$. Finally, the sets $P_0, \dots, P_{\ell-2}$ exactly contain the parties $1, \dots, (\ell-2+1)m$. Note that $(\ell-2+1)m = (\lceil \frac{n-1}{m} \rceil - 1)m \geq (\frac{n-1}{m} - 1)m = n-1-m$, and thus $P_{\ell-1}$ contains all of the rest of the parties, except for the sender.

As defined in the previous lower bound, for every $i \in [n]$, let X_i be a random variable indicating the total number of messages sent or received by party i in total both in W_0 and in W_1 . In addition, for every $i \in \{0, \dots, \ell-1\}$ let Y_i be the total number of messages sent or received by all parties $j \in P_i$ in total both in W_0 and in W_1 . It is always the case that $\sum_{j \in P_i} X_j \geq Y_i$ because $\sum_{j \in P_i} X_j$ counts all messages sent or received by parties in P_i , and might even count some of those messages twice. Assume by way of contradiction that for every $i \in \{0, \dots, \ell-1\}$, $\mathbb{E}[Y_i] > \epsilon f$. First note that $M = \frac{1}{2} \sum_{i=1}^n X_i$ because when summing over all the messages that each party sent and received, we count every message twice. In addition, seeing as each party j appears in at most two of the sets P_i , $2 \sum_{i=1}^n X_i \geq \sum_{i=0}^{\ell-1} \sum_{j \in P_i} X_j$. Combining these observations:

$$\begin{aligned}
\mathbb{E}[M] &= \mathbb{E}\left[\frac{1}{2} \sum_{i=1}^n X_i\right] \\
&= \frac{1}{4} \mathbb{E}\left[2 \sum_{i=1}^n X_i\right] \\
&\geq \frac{1}{4} \mathbb{E}\left[\sum_{i=0}^{\ell-1} \sum_{j \in P_i} X_j\right] \\
&\geq \frac{1}{4} \sum_{i=0}^{\ell-1} \mathbb{E}[Y_i] \\
&\geq \frac{1}{4} \ell \epsilon f \\
&= \frac{1}{4} \lceil \frac{n-1}{m} \rceil \epsilon f \\
&\geq \frac{1}{4} \cdot \frac{n-1}{\lfloor f^c \rfloor} \epsilon f \\
&\geq \frac{1}{4} \frac{n-1}{f^c} \epsilon f = \frac{\epsilon}{4} (n-1) f^{1-c}
\end{aligned}$$

in contradiction. This means that there exists at least one $k \in \{0, \dots, \ell-1\}$ for which $\mathbb{E}[Y_k] \leq \epsilon f$. Let P_k be such a set. From the Markov inequality, $\Pr[Y_k \geq f] \leq \frac{\mathbb{E}[Y_k]}{f} \leq \frac{\epsilon f}{f} = \epsilon$. In other words, the probability that in total all parties in P_k send and receive more than f messages in W_0 and in W_1 combined is no greater than ϵ .

We will now define an adversary's attack in W_{hybrid} , similar to the attack in Theorem 5. The sender s has the input 0. Whenever a party $j \notin P_k$ sends a message to a party $i \in P_k$, the adversary corrupts j and erases the message. In addition, whenever a party $i \in P_k$ sends a message to a party $j \notin P_k$, the adversary corrupts j . In parallel, the adversary simulates all of the messages parties $i \in P_k$ send in W_0 , given all of the messages they were sent by parties not in P_k . If any party $i \in P_k$ ever sends a message to party $j \notin P_k$ in that simulation in a given round, the adversary corrupts party j , erases its outgoing messages for that round, and

makes it act as a nonfaulty party would if it received all of the messages it already received and the messages sent by all parties in P_k in the simulated run. Finally, the adversary simulates all of the communication between all parties in W_1 given the messages sent by all parties $i \in P_k$ in W_{hybrid} . This is done by internally running all parties in each round of the protocol except for parties in P_k , and using the messages sent by parties in P_k in each round. Whenever a party $j \notin P_k$ sends some party $i \in P_k$ a message in the simulated run of W_1 , the adversary corrupts j in W_{hybrid} and sends that message to i . If at any point the adversary is required to corrupt more than f parties, it aborts. The adversary never corrupts any party $i \in P_k$, so all parties in P_k remain nonfaulty. Before analyzing the probability that the attack succeeds, we will define several random variables. Let A_0 be the event that all nonfaulty parties except parties in P_k output 0 in W_{hybrid} . Let A_1 be the event that all parties in P_k output 1 in W_{hybrid} . Similarly, let B_0 be the event that all nonfaulty parties except parties in P_k output 0 in W_0 , and let B_1 be the event that all parties in P_k output 1 in W_1 . Note that the definitions of A_0, B_0 allow all parties in P_k to output 0, as long as all other nonfaulty parties do so as well. Define G to be the event that no more than f parties communicate with parties in P_k in total in W_0 and W_1 combined. Finally, define G_{hybrid} to be the event that the adversary does not abort in W_{hybrid} .

Our goal is to show that $\Pr[A_0 \cap A_1] > \frac{1}{3} - \epsilon$. Note that in this case, all parties in P_k output 1 in W_{hybrid} and all other nonfaulty parties output 0. There are f^c parties in P_k and at least $n - f - f^c \geq n - 2f \geq f \geq f^c$ nonfaulty parties not in P_k . Therefore, with probability greater than $(\frac{1}{3} - \epsilon)$ at least f^c nonfaulty parties output 0 and at least f^c nonfaulty parties output 1, contradicting the fact that the protocol is an $(\frac{2}{3} + \epsilon)$ -correct all-but $(f^c - 1)$ Binary Crusader Broadcast protocol. Before doing so, note that as long as the adversary isn't required to corrupt more than f parties, the view of all nonfaulty parties except parties in P_k in W_{hybrid} is identical to the view they would have in W_0 , given that no more than f parties communicate with parties in P_k in both W_0 and W_1 . Similarly, as long as that event doesn't happen, the view of all parties in P_k in W_{hybrid} is identical to the view they would have in W_1 , given that no more than f parties communicate with parties in P_k in both W_0 and W_1 . Therefore, we know that $\Pr[G] = \Pr[G_{hybrid}]$, $\Pr[A_0|G_{hybrid}] = \Pr[B_0|G]$ and $\Pr[A_1|G_{hybrid}] = \Pr[B_1|G]$. We are now ready to analyze $\Pr[A_0 \cap A_1]$:

$$\begin{aligned}
\Pr[A_0 \cap A_1] &= \Pr[A_0] + \Pr[A_1] - \Pr[A_0 \cup A_1] \\
&\geq \Pr[G_{hybrid}] (\Pr[A_0|G_{hybrid}] + \Pr[A_1|G_{hybrid}]) - 1 \\
&= \Pr[G] (\Pr[B_0|G] + \Pr[B_1|G]) - 1 \\
&= \Pr[B_0 \wedge G] + \Pr[B_1 \wedge G] - 1 \\
&= \Pr[B_0] - \Pr[B_0 \wedge \overline{G}] + \Pr[B_1] - \Pr[B_1 \wedge \overline{G}] - 1 \\
&\geq \Pr[B_0] + \Pr[B_1] - 2\Pr[\overline{G}] - 1 \\
&= \Pr[B_0] + \Pr[B_1] - 2\Pr[Y_k > f] - 1 \\
&\geq (\frac{2}{3} + \epsilon) + (\frac{2}{3} + \epsilon) - 2\epsilon - 1 = \frac{1}{3} > \frac{1}{3} - \epsilon,
\end{aligned}$$

reaching a contradiction, and completing the proof. \blacktriangleleft

5 Eclipse Attacks in Blockchain Systems

Blockchain system provided new revolutionary consensus protocols [21, 13] and with them came a new set of attacks [4, 18, 11]. One new style of attack focuses on the underlying peer-to-peer communication network and the ways it might affect the security of the system

as a whole. These works suggest **Eclipse attacks** [15, 19], in which an adversary isolates a specific party (or group of parties), and causes it to fork off in ways that are economically advantageous to the attacker.

A natural question arises: are Eclipse-style attacks unique to the blockchain space or are they connected to more traditional attacks in the theory and literature on consensus? In consensus research, generic attacks on protocols are captured as lower bounds.

In this section, we make the conceptual connection between Eclipse-style attacks and theoretical lower bounds for Crusader broadcast. We show that many Eclipse-style attacks work because the underlying blockchain protocols are subquadratic and the protocol was not designed to take full power of forcing the adversary to be adaptive or to force the adversary to simulate. Hence Eclipse-style attacks can be viewed as specific attacks following the general lower bound for crusader broadcast, even with a mildly static adversary that cannot fully simulate as many other parties as it wants.

In the Eclipse attack, [15, 19] the adversary, by controlling a sufficient number of IP addresses, can monopolize all connections to and from a victim node. Once the node is isolated, the adversary can cause nodes to briefly locally confirm transactions that conflict with the majority of nodes. This is analogous to outputting different values in the attacks or our lower bound for crusader broadcast. Eclipse-style attacks may also combine selfish mining attacks [4, 11]. In this attack, the adversary filters communication to and from the isolated nodes and abuses the nodes' mining power to the adversary's advantage. This attack is also similar to the one described in Theorem 6, which suggests that these lower bounds could be of interest also when not directly attacking the agreement of the protocol, but rather notions like liveness or fairness of a consensus protocol.

We note that the difference between the attacks described in Theorems 4 and 5 stems from the randomized nature of the communication graph, and not from the difference in the content of messages. The proof of Theorem 4 would not change if a randomized Crusader Broadcast protocol uses a **static** communication graph. In addition, Theorem 6 generalizes the result even further and shows that as the number of messages decreases, or more precisely the number of edges in the communication graph decreases, a larger number of nonfaulty parties can be isolated and made to output a different value. As the number of edges in the communication graph tends towards $O(\epsilon \cdot n)$, the number of isolated parties tends towards $\Omega(f)$. This means a weak protocol with near linear communication may allow a large adversary to partition the nonfaulty parties into two large groups that disagree on the output of the protocol.

Limitations of Real-World Adversaries

The attacks described in Section 4 assume extremely strong adversaries. First of all, in all lower bounds, the adversary is assumed to be able to simulate other parties. This assumption may not hold in some real-world systems. Since adversaries have limited compute power, they generally cannot arbitrarily simulate other parties in proof-of-work systems. Furthermore, in systems with a public key infrastructure, adversaries cannot forge other parties' signatures or break other cryptographic primitives during the simulation of the protocol. The adversary in Theorems 5 and 6 also needs to be strongly adaptive. In the real world, adversaries generally cannot corrupt parties at will, let alone retroactively delete their messages and replace them. Given all of these limitations, one could reasonably ask: *are the attacks described in these lower bounds applicable to the real world?*

Surprisingly, the answer seems to be that they are applicable, as evidenced by previous works on eclipse attacks. We discuss how both limitations are overcome next.

Overcoming the need for strong adaptivity, due to protocol level flaws

As shown in [15, 19], both Bitcoin’s and Ethereum’s peer-to-peer communication protocols had flaws that allowed an adversary to easily monopolize a victim node’s connections. When nodes restart, they initiate outgoing connections from tables storing addresses of known peers. The adversary fills those tables in advance with addresses of nodes controlled by the adversary and then causes the node to restart. After restarting, the node connects to peers from those tables, hence connecting to the adversary’s nodes. Nodes may also receive incoming connections from peers. After causing a node to restart, the adversary also sends incoming connection requests and monopolizes all of the incoming connections. These attacks are performed in advance, allowing the adversary to essentially structure the communication graph in a malicious manner. Compare the Eclipse attack strategy above to our lower bounds for crusader broadcast. In this attack, the protocol flaw is such that the adversary does not need to be adaptive, let alone strongly adaptive. Even worse, the lower bound in Theorem 4 only shows that there must exist some party that can be isolated. In that attack, the adversary has to have some special knowledge of that specific party and tailor its attack to it. However, in the Eclipse attacks described in [15, 19], the adversary can choose whichever node it wants and isolate it in a static manner, without the need to find out which node can be isolated.

Overcoming the need for simulation due to the nature of proof of work

The second challenging assumption for a real-world attack is the assumption that the adversary has the power to simulate many honest parties. In our lower bounds for crusader broadcast, this stems from the fact that we do not know what the parties may do in the protocol. For example, parties may use cryptography in order to guarantee that a large portion of the network saw some value (see [25, 2] for such examples). In order to fully simulate the behavior of the nonfaulty parties, an adversary needs to be able to break some of the cryptographic assumptions made in the design of the protocol. On the other hand, in current proof-of-work based blockchain systems, simulating honest parties “only” requires mining blocks with the correct information. The adversary is limited by its own compute-power, so it can’t actually fully simulate the rest of the network for the isolated parties. However, Eclipse-style attacks suggest ways to mitigate this issue. For example, the adversary could conceivably utilize honest nodes to simulate the protocol for it. This can be done by letting only parts of the network see a given block. The adversary could then use the fact that nodes would continue to mine on top of it as a means of simulating the work required, and then showing the mined blocks to the rest of the network when needed.

To conclude, the adversary described in the lower bounds of Section 4 seems too powerful to be of interest when discussing real-world systems. However, some of the real-world systems used today had flaws that didn’t require the adversary to be so powerful in order to levy attacks.

Lessons from theory

Our lower bound suggests principled ways to design more secure protocols that will not allow adversaries with limited adaptivity and simulation power to succeed in their attacks. As suggested by [15, 19], measures could be taken in order to make it harder to fill the outgoing link tables with the adversary-controlled nodes’ addresses. The proof of Theorem 4 suggests that having a dynamically changing communication graph with outgoing edges being chosen randomly without much adversary control is the best long-term solution.

In addition, one could make it harder to simulate parts of the protocol. For example, by requiring more nodes to sign blocks, or by making more use of cryptography in the communication layer itself. This is indeed obtained using BFT-based finality gadgets [5].

6 Subquadratic Crusader Broadcast Protocol

After focusing on lower bounds for Crusader Broadcast, in this section, we explore upper bounds. We start with a trivial information theoretic crusader broadcast protocol with $O(n^2)$ communication. Our lower bound proves that this folklore construction is in fact tight for an unconditional adversary.

We then explore how using randomization and assuming a PKI can circumvent the $\Omega(n^2)$ lower bound for crusader broadcast. In the second protocol, the $O(n^2)$ all-to-all cost is replaced by a gossip procedure [22]. This lowers the overall communication cost to $O(n \cdot \text{polylog}n)$ at the cost of increasing the round complexity. The gossip protocol heavily relies on randomization to limit the adversary’s ability to guess the communication pattern. We analyze this protocol against static adversaries. This protocol is a sort of “minimal example” showing that it is easy to force the adversary to either be adaptive or to be able to simulate other parties in order to break subquadratic crusader broadcast protocols.

We start with a simple construction of a Crusader Broadcast protocol resilient to f strongly adaptive and computationally unbounded Byzantine corruptions, as long as $n > 3f$.

Algorithm 1 IT – CrusaderBroadcast_i.

```

1: if  $i$  is the sender  $s$  with input  $x$  then
2:   send the message  $\langle \text{“sender”}, x \rangle$  to all parties
3: wait  $\Delta$  time
4: if a single  $\langle \text{“sender”}, m \rangle$  message was received from  $s$  while waiting then
5:   send  $\langle \text{“forward”}, m \rangle$  to all parties
6: wait  $\Delta$  time
7: if there exists a value  $m$  for which  $\langle \text{“forward”}, m \rangle$  was received from at least  $n - f$  parties
   then
8:   output  $m$  and terminate
9: else
10:  output  $\perp$  and terminate

```

► **Theorem 7.** *The IT – CrusaderBroadcast protocol is a Crusader Broadcast protocol resilient to f strongly adaptive, computationally unbounded Byzantine corruptions in a synchronous system if $n > 3f$.*

Proof. Each property is proven individually.

Validity. Assume the sender s is nonfaulty with input x . In the beginning of the protocol it sends the message $\langle \text{“sender”}, x, \rangle$ to all parties. Every nonfaulty party receives that message up to Δ time after that, and sends $\langle \text{“forward”}, x \rangle$ to all parties. After Δ time, every nonfaulty party will receive a $\langle \text{“forward”}, x \rangle$ message from every nonfaulty party. Since there are $n - f$ nonfaulty parties, every nonfaulty party then outputs x .

Correctness. Observe two nonfaulty parties i, j that output two non- \perp values m_i, m_j respectively. This means that i received the message $\langle \text{“forward”}, m_i \rangle$ from at least $n - f$ parties, and j received the message $\langle \text{“forward”}, m_j \rangle$ from at least $n - f$ parties. By assumption,

$n > 3f$, and thus $2(n - f) = 2n - 2f = n + (n - 2f) \geq n + f + 1$. Therefore, i and j received the aforementioned messages from at least $f + 1$ common parties. At least one of those parties must be nonfaulty, and nonfaulty parties only send a single “forward” message to all parties. Therefore, it must be the case that $m_i = m_j$.

Termination. All parties wait for 2Δ overall and terminate. ◀

Note that in the above protocol, the sender sends $O(n)$ messages, and each nonfaulty party sends $O(n)$ messages as well. This results in a protocol with $O(n^2)$ message complexity, showing that the lower bound above is tight.

The folklore $O(n^2)$ protocol, presented in Algorithm 2, proceeds in two rounds. In the first round, the sender s sends a signed message with its input to all parties. Parties then inform each other of the message they’ve seen. Finally, any party that received a message m from the sender without seeing any conflicting message outputs m . If either of these conditions doesn’t hold, that party outputs \perp instead. This protocol is captured in Algorithm 2. In general, for a protocol X , denote X_i to be the code for party i executing protocol X . We assume the existence of a PKI such that every party i knows a signing key sk_i and all parties know the associated public key pk_i . The PKI is used in a signature scheme consisting of the signing algorithm Sign and verification algorithm Verify . We analyze the signature scheme as perfectly secure, meaning that only i can produce signatures which verify with respect to pk_i . A similar analysis can be done allowing for a negligible probability of error (meaning that the resulting protocol is $1 - \text{negl}(\lambda)$ correct, with λ being the security parameter).

■ **Algorithm 2** CrusaderBroadcast _{i} .

```

1:  $val \leftarrow \perp$ 
2: if  $i$  is the sender  $s$  with input  $x$  then
3:    $\sigma \leftarrow \text{Sign}(\text{sk}_i, x)$ 
4:   send the message  $\langle \text{“sender”}, x, \sigma \rangle$  to all parties
5: wait  $\Delta$  time
6: if a  $\langle \text{“sender”}, m, \sigma \rangle$  message was received from  $s$  while waiting such that
    $\text{Verify}(\text{pk}_s, m, \sigma) = 1$  then
7:    $val \leftarrow m$ 
8:   send  $\langle \text{“forward”}, m, \sigma \rangle$  to all parties
9: wait  $\Delta$  time
10: if a  $\langle \text{“forward”}, m', \sigma' \rangle$  message was received while waiting such that  $m' \neq val$  and
     $\text{Verify}(\text{pk}_s, m', \sigma') = 1$  then
11:    $val \leftarrow \perp$ 
12: output  $val$  and terminate

```

The protocol consists of a single multicast requiring $O(n)$ messages, and a single all-to-all round requiring $O(n^2)$ messages. A proof of the protocol follows:

► **Theorem 8.** *The CrusaderBroadcast protocol is a Crusader Broadcast protocol resilient to any number of Byzantine corruptions f in a synchronous system.*

Proof. Each property is proven individually. Denote val_i to be the variable val stored by party i .

Validity. Assume the sender s is nonfaulty with input x . In the beginning of the protocol it produces a signature σ for x , and sends the message $\langle \text{“sender”}, x, \sigma \rangle$ to all parties. Every nonfaulty party receives that message up to Δ time after that, and updates val to

x . The sender didn't sign any other value $m' \neq x$, so no nonfaulty party will receive a $\langle \text{"forward"}, m', \sigma' \rangle$ message with such that $m' \neq val$ and $\text{Verify}(\text{pk}_i, m', \sigma') = 1$. Therefore no nonfaulty party reverts val back to \perp . Finally, after 2Δ time, all nonfaulty parties output $val = x$ and terminate.

Correctness. Assume by way of contradiction two nonfaulty parties $i \neq j$ output two non- \perp values m_i, m_j respectively such that $m_i \neq m_j$. Those parties output the variable val at the end of the protocol, after 2Δ time. By assumption, they output non- \perp values, so $val_i \neq \perp$ and $val_j \neq \perp$. Party i only updates val_i to $m_i \neq \perp$ at time Δ in line 7, if it received a $\langle \text{"sender"}, m_i, \sigma_i \rangle$ message from s such that $\text{Verify}(\text{pk}_s, m_i, \sigma_i) = 1$. It then sends the message $\langle \text{"forward"}, m_i, \sigma_i \rangle$ to all parties at time Δ . Party j receives that message by time 2Δ , sees that $m_i \neq m_j$ and $\text{Verify}(\text{pk}_s, m_i, \sigma_i) = 1$ and updates val_j to \perp . Finally, j outputs $val_j = \perp$, contradicting the fact that it output some value $m_j \neq \perp$.

Termination. All parties wait for 2Δ overall and terminate. \blacktriangleleft

This simple protocol is based on the fact that for correctness to hold, it is enough to make sure that any value heard by a nonfaulty party needs to be heard by all nonfaulty parties (or at least the fact that two nonfaulty parties heard different values). In order to reduce the communication costs of the protocol, it is possible to replace the expensive all-to-all communication round with a more efficient *gossip* procedure. Using well-known results about gossip [7, 22, 16], we know that parties can exchange information between them by proceeding in rounds and in each round choosing a party to divulge all heard information to. Using this technique, it is guaranteed that in $O(\log n)$ rounds all parties will hear all nonfaulty parties' initial information. When dealing with a constant number of Byzantine faults, simply raising the number of parties with which each party communicates in each round yields the same analysis, showing that such a protocol requires $O(n \cdot \text{polylog}(n))$ messages to be sent overall, yielding a subquadratic Crusader Broadcast protocol. It is also possible to reduce the size of the messages by parties sending up to 2 of the values they heard up until that point. This is enough to detect equivocation, while guaranteeing that message size remains constant. Note that an adaptive adversary can make sure that no nonfaulty party receives a message m from an informed party i in the first rounds of the protocol by corrupting the parties which received messages from i , requiring more rounds and more overall communication. This shows that a subquadratic randomized protocol exists which is resilient to non-adaptive adversaries, but stops working when the adversary can corrupt parties adaptively.

Another approach for breaking the quadratic message barrier is by relying on stronger cryptographic primitives. This has been useful in reducing the communication costs of protocols solving related tasks such as consensus [20, 25]. For example, if $n > 3f$ it is possible to use threshold cryptography. Given a well known threshold k , a threshold signature scheme allows parties to sign a message individually, and then compressing k such signatures into a single collective signature, proving that at least k parties signed the message individually. Instead of having an all-to-all round as described in Algorithm 2, parties that hear a value m from the sender can reply, sending a signature on m to the sender. Using a threshold signature scheme with a threshold of $2f + 1$, the sender can combine those signatures into a single threshold signature proving that at least $2f + 1$ parties replied with a signature on the value m . Since $2f + 1$ parties constitute a Byzantine quorum, it is guaranteed that there is only one such threshold signature, meaning that if the sender then sends the threshold signature to all parties, they can safely output it. This technique, used in the non-equivocation round of the HotStuff protocol [25], yields a protocol with $O(n)$ communication costs and $O(1)$ rounds, but more heavily relies on the assumption that the adversary cannot simulate other parties.

References

- 1 Ittai Abraham, TH 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*, pages 317–326, 2019.
- 2 Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Validated asynchronous byzantine agreement with optimal resilience and asymptotically optimal time and word communication, 2018. doi:10.48550/arXiv.1811.01332.
- 3 Ittai Abraham and Kartik Nayak. The dolev and reischuk lower bound: Does agreement need quadratic messages? <https://decentralizedthoughts.github.io/2019-08-16-byzantine-agreement-needs-quadratic-messages/>, 2019.
- 4 Lear Bahack. Theoretical bitcoin attacks with less than half of the computational power (draft), 2013. doi:10.48550/arXiv.1312.7013.
- 5 Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *arXiv preprint*, 2017. arXiv:1710.09437.
- 6 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.
- 7 Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12, 1987.
- 8 Danny Dolev. The byzantine generals strike again. *Journal of algorithms*, 3(1):14–30, 1982.
- 9 Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for byzantine agreement. *Journal of the ACM (JACM)*, 32(1):191–204, 1985.
- 10 C Dwork, D Peleg, N Pippenger, and E Upfal. Fault tolerance in networks of bounded degree. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, STOC '86, pages 370–379, New York, NY, USA, 1986. Association for Computing Machinery. doi:10.1145/12130.12169.
- 11 Ittay Eyal and Emin Gun Sirer. Majority is not enough: Bitcoin mining is vulnerable, 2013. doi:10.48550/arXiv.1311.0243.
- 12 Michael J Fischer, Nancy A Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1(1):26–39, 1986.
- 13 Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 281–310. Springer, 2015.
- 14 Vassos Hadzilacos and Joseph Y Halpern. Message-optimal protocols for byzantine agreement. In *Proceedings of the tenth annual ACM symposium on Principles of distributed computing*, pages 309–323, 1991.
- 15 Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse attacks on bitcoin’s peer-to-peer network. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 129–144, 2015.
- 16 R. Karp, C. Schindelhauer, S. Shenker, and B. Vocking. Randomized rumor spreading. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 565–574, 2000.
- 17 Valerie King, Jared Saia, Vishal Sanwalani, and Erik Vee. Scalable leader election. In *SODA '06*, 2006.
- 18 Joshua A Kroll, Ian C Davey, and Edward W Felten. The economics of bitcoin mining, or bitcoin in the presence of adversaries. In *Proceedings of WEIS*. Washington, DC, 2013.
- 19 Yuval Marcus, Ethan Heilman, and Sharon Goldberg. Low-resource eclipse attacks on ethereum’s peer-to-peer network. *Cryptology ePrint Archive*, 2018.

- 20 Atsuki Momose and Ling Ren. Optimal communication complexity of authenticated 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 32:1–32:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.DISC.2021.32.
- 21 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, page 21260, 2008.
- 22 Boris Pittel. On spreading a rumor. *SIAM Journal on Applied Mathematics*, 47(1):213–223, 1987.
- 23 Peter Robinson, Christian Scheideler, and Alexander Setzer. Breaking the $\tilde{\omega}(\sqrt{n})$ barrier: Fast consensus under a late adversary. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures, SPAA '18*, pages 173–182, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3210377.3210399.
- 24 Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.
- 25 Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan-Gueta, and Ittai Abraham. Hotstuff: BFT consensus with linearity and responsiveness. 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 347–356. ACM, 2019. doi:10.1145/3293611.3331591.

Quorum Systems in Permissionless Networks

Christian Cachin  

University of Bern, Switzerland

Giuliano Losa  

Stellar Development Foundation, San Francisco, CA, USA

Luca Zanolini  

University of Bern, Switzerland

Abstract

Fail-prone systems, and their quorum systems, are useful tools for the design of distributed algorithms. However, fail-prone systems as studied so far require every process to know the full system membership in order to guarantee safety through globally intersecting quorums. Thus, they are of little help in an open, permissionless setting, where such knowledge may not be available. We propose to generalize the theory of fail-prone systems to make it applicable to permissionless systems. We do so by enabling processes not only to make assumptions about failures, but also to make assumptions about the assumptions of other processes. Thus, by transitivity, processes that do not even know of any common process may nevertheless have intersecting quorums and solve, for example, reliable broadcast. Our model generalizes existing models such as the classic fail-prone system model [Malkhi and Reiter, 1998] and the asymmetric fail-prone system model [Cachin and Tackmann, OPODIS 2019]. Moreover, it gives a characterization with standard formalism of the model used by the Stellar blockchain.

2012 ACM Subject Classification Theory of computation → Cryptographic protocols; Software and its engineering → Distributed systems organizing principles

Keywords and phrases Permissionless systems, fail-prone system, quorum system

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2022.17

Related Version *Full Version:* <https://doi.org/10.48550/arxiv.2211.05630>

Funding This work has been funded by the Swiss National Science Foundation (SNSF) under grant agreement Nr. 200021_188443 (Advanced Consensus Protocols).

Acknowledgements The authors thank anonymous reviewers for helpful feedback.

1 Introduction

A common problem in distributed computing is to implement synchronization abstractions such as reliable broadcast, shared memory, or consensus, given some assumptions about the possible Byzantine failures that may occur in an execution.

A *fail-prone system* \mathcal{F} [14] is a set of sets of processes, called *fail-prone sets*, where no fail-prone set is a subset of another. A fail-prone system \mathcal{F} denotes the assumption that the set of processes A that may suffer Byzantine failures is contained in one of the fail-prone sets. For example, in a system of n processes, it is common to assume that less than a third will fail, i.e., the fail-prone sets are the sets of cardinality exactly $\lfloor (n-1)/3 \rfloor$.

Fail-prone systems are useful because of their relationship to quorum systems [14]. A Byzantine quorum system \mathcal{Q} for \mathcal{F} is a collection of subsets of processes, called *quorums*, such that for every two quorums Q_1 and Q_2 in \mathcal{Q} and for every fail-prone set $F \in \mathcal{F}$ it holds that Q_1 and Q_2 have a common member outside F (Consistency) and for every fail-prone set $F \in \mathcal{F}$, there exists a quorum disjoint from F (Availability).



© Christian Cachin, Giuliano Losa, and Luca Zanolini;
licensed under Creative Commons License CC-BY 4.0

26th International Conference on Principles of Distributed Systems (OPODIS 2022).

Editors: Eshcar Hillel, Roberto Palmieri, and Etienne Rivière; Article No. 17; pp. 17:1–17:22

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Many distributed algorithms (implementing, e.g., Byzantine reliable broadcast or consensus) are parameterized by a quorum system \mathcal{Q} and their guarantees hold under the assumptions of a fail-prone system \mathcal{F} if and only if Consistency and Availability of \mathcal{Q} hold. This allows the designers of a distributed system to make assumptions about failures, pick a corresponding quorum system, and then choose among existing algorithms to solve the desired synchronization problem.

Traditionally, such fail-prone systems have been used in closed systems with assumptions of the form “less than one third of the processes are faulty”. This can work even in permissionless systems using Proof-of-Stake, e.g., assuming that less than one third of the stake-holders are faulty, or Proof-of-Work, e.g., assuming that the faulty set of processes holds less than one third of the mining power [17]. Proof-of-Stake and Proof-of-Work however have their disadvantages, e.g., long-range attacks in Proof-of-Stake, or excessive energy consumption in Proof-of-Work. Both may also suffer from the “rich getting richer” problem, leading to very few entities eventually controlling the system.

Instead of Proof-of-Stake and Proof-of-Work, one can envision letting every participant make its own, subjective failure assumptions. Distributed computing models based on this idea was first investigated by Damgård *et al.* [8], followed by Sheff *et al.* [18], and more recently by Cachin and Tackmann [5] with the *asymmetric-trust model*, Malkhi *et al.* [13] with *flexible quorums*, and Sheff *et al.* [19] with Heterogeneous Paxos. On the practical side, Ripple (<https://ripple.com>) deployed a permissionless consensus protocol based on subjective quorums in 2012, in every process declares a *Unique Node List* (UNL) of processes that it trusts. Stellar (<https://stellar.org>, [16]) later followed suite with a different permissionless model based on subjective trust.

One problem is that, in all the models cited above, even though participants are free to make their own failure assumptions or choose their own quorums, maintaining consistency requires compatible assumptions (in the sense that the resulting quorums will sufficiently intersect) and thus prior common knowledge or prior synchronization, which is not desirable in a permissionless system.

For example, Cachin and Tackmann [5] assume that for every two participants p_i and p_j , for every two quorums Q_i of p_i and Q_j of p_j , if F is a set that can fail according to the assumptions of both p_i and p_j , then $(Q_i \cap Q_j) \setminus F \neq \emptyset$; this is called the *consistency* property. Together with an *availability* property, this defines an *asymmetric quorum system*. Consistency ensures that if both p_i and p_j make correct assumptions, then they can avoid disagreeing in, e.g., a reliable broadcast protocol. Achieving liveness additionally requires the availability property, i.e., that a group of participants, called a *guild*, whose assumptions are correct and which do not fail, satisfy the consistency property (pairwise) and additionally that every member of the guild has a quorum in the guild.

Maintaining safety in such a model requires a strong form of a-priori coordination. Indeed, two participants cannot be prevented from disagreeing unless every two of their quorums have at least one non-faulty participant in common. Thus participants must coordinate beforehand in order to pick sufficiently overlapping survivor sets, and corresponding quorums. In the model of Ripple, for instance, every two processes must have UNLs that overlap by some sufficient fraction [1].

Global assumptions implying the intersection of survivor sets and quorums, as in the two preceding examples, are problematic in a permissionless setting because they postulate some form of pre-agreement or common knowledge, which might be hard to achieve in practice.

Interestingly, the Stellar network (<https://stellar.org>, [16]), a deployed blockchain system based on quorums, is able to maintain safety and liveness without requiring that participants choose intersecting quorums. Instead, participants choose *quorum slices* that

need not intersect, and the quorums of a participant are defined in terms of the slices of other participants. Consensus can then be solved within an *intact set* \mathcal{I} , a set of correct processes such that every two processes p_i and p_j in \mathcal{I} have all their own quorums that intersect in at least a member of \mathcal{I} and such that \mathcal{I} is itself a quorum for every of its members.

We observe that quorum slices can be interpreted as a new kind of failure assumptions: a participant assumes that at least one of its quorum slices is made exclusively of participants that do not fail and make correct assumptions. In other words, a participant's assumption are not only about failures, but also about whether other participants make correct assumptions. In practice, the Stellar model makes it easier for participants to achieve quorum intersection by relying on the failure assumptions of other participants that might have more knowledge about the system than they have.

The main contribution of this paper is to show that this new kind of failure assumptions yield a generalization of the theory of fail-prone systems (i.e., classic fail-prone systems are a special case) which allows to obtain intersecting quorums even when participants do not know any common third party.

Moreover, based on this, we introduce the notion of *permissionless fail-prone system* from which it is possible to derive a *permissionless quorum system*.

This paper is structured as follows. In Section 2 we formally define the assumptions of a process; each process assumes that one of its slices S will not fail and, *additionally*, that the assumptions of every process in S will be satisfied too. Then, we introduce the notion of permissionless fail-prone system. This extends and generalizes the *asymmetric fail-prone system* [5, 8]. Crucially, we note that the new meaning of the assumptions of the processes allows processes to transitively rely on the assumptions of other processes. However, this in turn enables malicious processes to lie about their assumptions.

In Section 3, we propose a computation model, based on the notion of *view*, that takes this phenomenon into account. Moreover, always in Section 3, we derive from the permissionless fail-prone system the notion of *permissionless Byzantine quorum system*.

In Section 4 we introduce the notion of *league*, which is a set of processes L that enjoys Consistency (quorums intersection) and Availability (existence of a quorum in L consisting of correct processes) among the correct members of L even when faulty processes lie about their configuration.

We compare our permissionless model with the classic model based on fail-prone systems [14], with the asymmetric model [5, 8], with the federated Byzantine agreement system [16] and with the personal Byzantine quorum system model [12] in Section 5. Interestingly we show that classic fail-prone systems can be understood as a special case of our model.

In Section 6 we show how a traditional synchronization protocol, i.e., Bracha broadcast [2], can be adapted to work in our model, thereby offering a new toolbox for the design of permissionless distributed systems.

In the full version of this paper [4] we present another application of permissionless quorum systems through the emulation of shared memory, represented by a register. In particular we show how to implement a single-writer multi-reader register with permissionless quorum systems.

Related work and conclusions are presented in Section 7 and Section 8, respectively. Finally, our model also leads to a characterization of the Stellar model with standard formalism [5, 8, 14].

2 Model

We consider an unbounded set of *processes* $\Pi = \{p_1, p_2, \dots\}$ that communicate asynchronously with each other by sending messages. We assume that processes do not necessarily know which other processes are in the system (i.e., each process only knows a subset of Π).

Processes are assigned a protocol to follow. Protocols are presented in a modular way using the event-based notation of Cachin et al. [3]. A process that follows its algorithm during an execution is called *correct*. Initially, all processes are correct, but a process may later fail, in which case it is called *faulty*. We assume Byzantine failures, where a process that fails thereafter behaves arbitrarily.

We assume that point-to-point communication between any two processes (that know each other) is available, as well as a best-effort gossip primitive that will reach all processes. In a protocol, this primitive is accessed through the events “sending a message through gossip” and “receiving a gossiped message.” We assume that messages from correct processes to correct process are eventually received and cannot be forged. The system itself is asynchronous, i.e., the delivery of messages among processes may be delayed arbitrarily and the processes have no synchronized clocks.

Processes make failure assumptions about other processes. However, since a process does not know exactly who is part of the system, it cannot make failure assumptions about the whole system. Instead, each process p_i makes assumptions about a set $P_i \subseteq \Pi$, called p_i 's *trusted set*, using a *fail-prone system* \mathcal{F}_i over P_i (Section 1). Here, \mathcal{F}_i is a collection of subsets of P_i and p_i believes that up to any set $F \in \mathcal{F}_i$ may jointly fail. We say that P_i and \mathcal{F}_i constitute p_i 's *assumptions* and we assume that they remain fixed during an execution.

Note that assuming that the assumptions of the processes are fixed is a simplification. In practice, the system may experience churn, i.e., processes frequently entering and departing the system. Processes that remain in the system can adjust their assumptions in response to churn: for example, if a process stops responding (maybe because it has left the system), then other processes can remove it from their assumptions. Conversely, if a new process joins the system, existing processes may adjust their assumptions to include that process. However, analyzing the system under churn is out of the scope of this paper.

► **Definition 1** (Fail-prone system). *A fail-prone system \mathcal{F} over $P \subseteq \Pi$ is a set of subsets of Π called fail-prone sets, none of which contain the other (i.e., if $F \in \mathcal{F}$ and $F' \subset F$, then $F' \notin \mathcal{F}$).*

A *permissionless fail-prone system* (abbreviated PFPS) describes the assumptions of all the processes:

► **Definition 2** (Permissionless fail-prone system). *A permissionless fail-prone system is an array $\mathbb{F} = [(P_1, \mathcal{F}_1), (P_2, \mathcal{F}_2), \dots]$ that associates each process p_i to a trusted set $P_i \subseteq \Pi$ and a fail-prone system \mathcal{F}_i over P_i . We refer to (P_i, \mathcal{F}_i) as the configuration of process p_i .*

We now consider a fixed PFPS \mathbb{F} .

► **Definition 3** (Tolerated execution and tolerated set). *We say that the assumptions of a process p_i are satisfied in an execution if the set A of processes that actually fail is such that there exists a fail-prone set $F \in \mathcal{F}_i$ and:*

1. $A \cap P_i \subseteq F$; and
2. the assumptions of every member of $P_i \setminus F$ are satisfied.

If $p_i \in \Pi$ has its assumptions satisfied in an execution e , we say that p_i tolerates the execution e .

Finally, a set of processes L tolerates a set of processes A if and only if every process $p_i \in L \setminus A$ tolerates an execution e with set of faulty processes A .

► **Example 4.** Consider a set of processes $\Pi = \{p_1, p_2, p_3, p_4\}$ and a permissionless fail-prone system $\mathbb{F} = [(\Pi, \mathcal{F}_1), (\Pi, \mathcal{F}_2), (\Pi, \mathcal{F}_3), (\Pi, \mathcal{F}_4)]$ with $\mathcal{F}_1 = \{\{p_3, p_4\}\}$, $\mathcal{F}_2 = \{\{p_1, p_4\}\}$, $\mathcal{F}_3 = \{\{p_1, p_4\}\}$, and $\mathcal{F}_4 = \{\{p_1, p_2\}\}$. Then, Π tolerates the sets \emptyset , $\{p_1\}$, $\{p_4\}$ and $\{p_1, p_4\}$. To see this, let us assume an execution e with set of faulty processes $A = \{p_1, p_4\}$. Then, for every $p_i \in \Pi \setminus A$, there exists a fail-prone set $F \in \mathcal{F}_i$ such that $A \cap \Pi \subseteq F$. In particular, $\Pi \setminus A = \{p_2, p_3\}$ and $\{p_1, p_4\} \in \mathcal{F}_2$ and $\{p_1, p_4\} \in \mathcal{F}_3$. The same reasoning can be applied for the other sets.

Note that here we depart significantly from the traditional notion of fail-prone systems [5, 14]: in a PFPS, a process not only makes assumptions about failures, but also makes assumptions about the assumptions of other processes.

Next we define *survivor sets* analogously to Junqueira and Marzullo [11]. In the traditional literature, a survivor set of p_i is the complement, within Π , of some fail-prone set. However, defining them as the complement of fail-prone sets within P_i does not work because of Item 2 in Definition 3. To obtain this definition, we first define a *slice*.

► **Definition 5 (Slice).** A set $\bar{F} \subseteq \Pi$ is a slice of p_i if and only if p_i has a fail-prone set $F \in \mathcal{F}_i$ such that $\bar{F} = P_i \setminus F$.

For any $S \subseteq \Pi$ we often say p_i has a slice in S when a slice of p_i is contained in S or when S contains a superset of a slice of p_i .

► **Definition 6 (Survivor-set system).** A survivor-set system \mathcal{S}_i of p_i is the minimal set of subsets S of Π such that:

1. p_i has a slice in S ; and
2. every member of S has a slice in S .

Each $S \in \mathcal{S}_i$ is called a survivor set of p_i .

► **Example 7.** Continuing from Example 4, process p_1 has only one slice consisting of $\{p_1, p_2\}$, processes p_2 and p_3 have the set $\{p_2, p_3\}$ as slice, and process p_4 has the set $\{p_3, p_4\}$ as slice. Moreover, the survivor-set systems are $\mathcal{S}_1 = \{\{p_1, p_2, p_3\}, \{p_1, p_2, p_3, p_4\}\}$ for process p_1 , $\mathcal{S}_2 = \{\{p_2, p_3\}, \{p_1, p_2, p_3, p_4\}\}$ for process p_2 , $\mathcal{S}_3 = \{\{p_2, p_3\}, \{p_1, p_2, p_3, p_4\}\}$ for process p_3 , and $\mathcal{S}_4 = \{\{p_2, p_3, p_4\}, \{p_1, p_2, p_3, p_4\}\}$ for process p_4 . This follows from Definition 6: given a survivor set $S \in \mathcal{S}_i$ for p_i , process p_i must have a slice in S and every member of S must have a slice in S . So, for example, given the survivor set $\{p_1, p_2, p_3\}$ in the survivor set system \mathcal{S}_1 for p_1 , process p_1 has a slice in $\{p_1, p_2, p_3\}$, i.e., $\{p_1, p_2\}$, and every process $p_i \in \{p_1, p_2, p_3\}$ has a slice in $\{p_1, p_2, p_3\}$, i.e., $\{p_2, p_3\}$.

► **Lemma 8.** The assumptions of a process $p_i \in \Pi$ are satisfied in an execution e with set of faulty processes A if and only if there exists a survivor set $S \in \mathcal{S}_i$ of p_i such that S does not fail.

Proof. Let p_i be a process such that, given an execution e with set of faulty processes A , the assumptions of p_i are satisfied in e . This implies that, by Definition 3, there exists a set of processes such that each of these processes has its assumptions satisfied. Moreover, by Definition 5, each of these processes has a slice \bar{F}_j such that $\bar{F}_j \cap A = \emptyset$. This leads to have a set S obtained as union of all of these slices such that $S \cap A = \emptyset$ and such that S is minimal with respect to this union, in the sense that is the minimal set of processes such that every process in S has its assumptions satisfied. The set S is a survivor set of p_i .

Conversely, we show that given a survivor set S of p_i , given a process $p_i \in S$ and given an execution e with set of faulty processes A , if $S \cap A = \emptyset$, then the assumptions of p_i are satisfied in e . Observe that, from the assumptions, we have that every process in S has a slice \bar{F} in S such that $\bar{F} \cap A = \emptyset$. This means that for every process p_i in S , there exists a fail-prone set $F \in \mathcal{F}_i$ such that $P_i \cap A \subseteq F$. This implies that every process in S has its assumptions satisfied and, in particular, that $p_i \in S$ has its assumptions satisfied in e . ◀

3 Permissionless Quorum Systems

A classic (or symmetric) fail-prone system [14] determines a canonical quorum system known to all processes through the Q^3 -condition. Specifically, given a fail-prone system \mathcal{F} , the Q^3 -condition requires that no three fail-prone sets of \mathcal{F} cover the complete set of processes and this condition holds if and only if there exists a quorum system for \mathcal{F} [10, 14]. Such a quorum system could be, for example, the complement of every fail-prone set of \mathcal{F} , which we call the *canonical* quorum system. Traditional algorithms such as read-write register emulations [14], Byzantine reliable broadcasts [2, 20] or the PBFT algorithm [7] make use of quorums.

In the model of asymmetric trust [8] the assumptions of the processes may differ, and asymmetric quorum systems [5, 6] allow to implement the above-mentioned algorithms in a more flexible way. However, they still require a system that is known to every process.

In a permissionless system, processes do not know the membership and have different, partial, and potentially changing views of its composition.

Given a PFPS, we would therefore like to obtain a quorum system to implement algorithms for register emulation, broadcast, consensus and more, while allowing the processes to have different assumptions in an open network.

We are therefore interested in defining a notion of quorums for open systems where:

1. each process has its own quorum system; and
2. the quorums of a process p_i depend on the assumptions of other processes, which p_i learns by communicating with them.

In other words, we consider scenarios in which each process p_i communicates with other processes, continuously discovers new processes and learns their assumptions. During this execution, p_i determines its current set of quorums as a function of what it has learned so far. Importantly, this means that the quorums of a process evolve as the process learns new assumptions, and that faulty processes can influence p_i 's quorums by lying about their assumption.

We now formalize this model using the notions of a *view* and a *quorum function*.

► **Definition 9 (View).** A view $\mathbb{V} = [\mathcal{V}_1, \mathcal{V}_2, \dots]$ is an array with one entry $\mathbb{V}[j] = \mathcal{V}_j$ for each process p_j such that:

1. either \mathcal{V}_j is the special value \perp ; or
2. $\mathcal{V}_j = (P_j, \mathcal{F}_j)$ consists of a set of processes P_j and a fail-prone system \mathcal{F}_j over P_j .

Observe that every process p_i has its *local* view \mathbb{V} , whose non- \perp entries represent the assumptions that p_i has learned at some point in an execution. Every other process p_j such that $\mathbb{V}[j] = \perp$ is a process that p_i has not heard from. We denote with Υ the set of all the possible views.

We assume that, for every process p_j , a process p_i 's view contains the assumption that p_i has most recently received from p_j . Finally, note that \mathbb{F} is a view in which no process is mapped to \perp . In particular, \mathbb{F} represents the global view if the system could be entirely observed. Since processes cannot observe the complete system, they normally only have partial knowledge of \mathbb{F} . Moreover, this knowledge evolves over time.

► **Definition 10 (Domain of a view).** For a view \mathbb{V} , the set of processes p_i such that $\mathbb{V}[i] \neq \perp$ is the domain of \mathbb{V} .

Next, we assume that every process determines its quorums according to its view using a function \mathcal{Q} called a *quorum function*. We assume that all correct processes use the same \mathcal{Q} and that they do not change it during an execution. We then have the following definition.

► **Definition 11** (Quorum function). *The quorum function $\mathcal{Q} : \Pi \times \Upsilon \rightarrow 2^\Pi$ maps a process p_i and a view \mathbb{V} to a set of sets of processes such that $Q \in \mathcal{Q}(p_i, \mathbb{V})$ if and only if:*

1. *a slice of p_i is contained in Q ; and*
2. *for every process $p_j \in Q$ with $\mathbb{V}[j] \neq \perp$ and $\mathbb{V}[j] = (P_j, \mathcal{F}_j)$, there exists $F \in \mathcal{F}_j$ such that $P_j \setminus F \subseteq Q$.*

Every element of $\mathcal{Q}(p_i, \mathbb{V})$ is called a quorum for p_i (in \mathbb{V}).

Notice that in the first condition, the quorum Q may itself be a slice of p_i . Moreover, Q is a quorum for every one of its members and it is defined by slices of every $p_i \in Q$. As shown in the following lemma, a quorum for p_i in view \mathbb{V} for p_i is a survivor set of p_i .

► **Lemma 12.** *For every view \mathbb{V} for $p_i \in \Pi$, every quorum $Q_i \in \mathcal{Q}(p_i, \mathbb{V})$ is a survivor set of p_i . Moreover, given S a survivor set of p_i , there exists a view \mathbb{V} for p_i such that $S \in \mathcal{Q}(p_i, \mathbb{V})$.*

Proof. Let $Q_i \in \mathcal{Q}(p_i, \mathbb{V})$ be a quorum for p_i with \mathbb{V} a view for p_i . By Definition 11, all processes in Q including p_i have a slice in Q . From Definition 6, this implies that Q is a survivor set of p_i .

Moreover, given a survivor set S of p_i , the set S consists of slices of every member of S . This means that there exists a view \mathbb{V} for p_i in which S satisfies Definition 11 and it is a quorum for p_i . This is the view \mathbb{V} defined as follows:

1. for every $p_j \in S$, $\mathbb{V}[j] = \mathbb{F}[j]$, and
2. for every $p_j \notin S$, $\mathbb{V}[j] = (\emptyset, \{\emptyset\})$. ◀

► **Example 13.** Let us consider Example 4 with survivor-set systems as shown in Example 7. Since all the processes already know all the configurations of every other process, we have that $\mathcal{S}_i = \mathcal{Q}(p_i, \mathbb{F})$, with \mathbb{F} the permissionless fail-prone system.

Combining the quorum sets of all processes, we now obtain a *permissionless quorum system* for \mathbb{F} .

► **Definition 14** (Permissionless quorum system). *A permissionless quorum system for Π and \mathbb{F} is an array of collections of sets $\mathcal{Q}_{perm} = [\mathcal{Q}(p_1, \mathbb{F}), \mathcal{Q}(p_2, \mathbb{F}), \dots]$, where $\mathcal{Q}(p_i, \mathbb{F})$ is called the quorum system for p_i and is determined by the quorum function \mathcal{Q} .*

Observe that our notion of a quorum system differs from that in the existing literature [5, 14, 15]. In particular, standard Byzantine quorum systems are defined through a pair-wise intersection among quorums. This is possible in scenarios where the full system membership is known to every process. However, in permissionless settings, this requirement cannot as clearly be achieved globally.

► **Definition 15** (Current quorum system). *Let \mathbb{V} be the view representing the assumptions that a process p_i has learned so far. Then the current quorum system of p_i is the set $\mathcal{Q}(p_i, \mathbb{V})$. Moreover, a set of processes Q is a current quorum of p_i if and only if $Q \in \mathcal{Q}(p_i, \mathbb{V})$; we also say that p_i has a quorum Q .*

Note that, in this model, each process has its own set of quorums and the set of quorums of a process changes throughout an execution as the process learns the assumptions of more processes. Importantly, note that faulty processes may lie about their configuration and influence the quorums of correct processes. In an execution e with faulty set A , a correct process p_i might have a view in which the assumptions of processes in A are arbitrary because processes in A lied about their assumptions. However, processes outside A do not lie about their assumptions. We capture this with the following definition.

► **Definition 16** (*T-resilient view*). Given a set of processes T , we say that a view \mathbb{V} is T -resilient if and only if for every process $p_i \notin T$, either $\mathbb{V}[i] = \perp$ or $\mathbb{V}[i] = \mathbb{F}[i]$.

Intuitively, a correct process p_i will either not have heard from $p_j \notin A$ or it will have the correct assumption for p_j . Thus, p_i 's view is A -resilient at all times in execution e .

As we said, processes in A may lie about their assumptions causing quorums to contain unreliable slices. Moreover, processes in A may aim at preventing intersection among quorums of correct processes. In the following definition we characterize the notion of worst-case view, i.e., when faulty processes gossip only empty configurations. By doing so, quorums of correct processes will contain fewer members, increasing the chances of an empty intersection among them.

► **Definition 17** (*Worst-case view*). Given a set of processes T , the worst-case view with respect to T is the view \mathbb{V}_T^* such that:

1. for every $p_i \in \Pi \setminus T$, $\mathbb{V}_T^*[i] = \mathbb{F}[i]$, and
2. for every $p_i \in T$, $\mathbb{V}_T^*[i] = (\emptyset, \{\emptyset\})$.

Finally, every quorum for a process $p_i \notin A$ in a A -resilient view contains a quorum for p_i in a worst-case view with respect to A . This is shown in the following lemma.

► **Lemma 18**. Consider a set of processes T , a T -resilient view \mathbb{V} , and a process $p_i \notin T$. Moreover, let us assume that processes in T may lie about their assumptions. For every quorum $Q_i \in \mathcal{Q}(p_i, \mathbb{V})$, there exists a quorum $Q'_i \in \mathcal{Q}(p_i, \mathbb{V}_T^*)$ such that $Q'_i \subseteq Q_i$.

Proof. Let T be a set of processes, \mathbb{V} be a T -resilient view, p_i a process not in T . Since \mathbb{V} is a T -resilient view, for every $p_j \notin T$ it holds either $\mathbb{V}[j] = \perp$ or $\mathbb{V}[j] = \mathbb{F}[j]$. However, processes in T may lie about their assumptions and, because of that, the view of process $p_i \notin T$ may contain arbitrary configurations received from processes in T .

If $Q_i \in \mathcal{Q}(p_i, \mathbb{V})$ is a quorum for p_i in \mathbb{V} , then Q_i might contain slices of processes in T which are derived from false assumptions. One can easily show that by starting from a T -compatible view and by removing the configurations received by processes in T , it is possible to obtain the corresponding worst-case view. By removing configurations from \mathbb{V} , also Q_i becomes smaller, i.e., with less members, obtaining a quorum $Q'_i \subseteq Q_i$. In fact, by removing from Q_i a slice \bar{F}_j of a process $p_j \in T$, also slices of other processes in \bar{F}_j might get removed in order for Definition 11 to be satisfied on Q'_i . This proves the lemma. ◀

4 Leagues

We now define the notion of a *league*. In Section 6 we show how a league allows to implement Bracha broadcast.

► **Definition 19** (*League*). set of processes L is a league for the quorum function \mathcal{Q} if and only if the following property holds:

Consistency: For every set $T \subseteq \Pi$ tolerated by L , for every two T -resilient views \mathbb{V} and \mathbb{V}' , for every two processes $p_i, p_j \in L \setminus T$, and for every two quorums $Q_i \in \mathcal{Q}(p_i, \mathbb{V})$ and $Q_j \in \mathcal{Q}(p_j, \mathbb{V}')$ it holds $(Q_i \cap Q_j) \setminus T \neq \emptyset$.

Availability: For every set $T \subseteq \Pi$ tolerated by L and for every $p_i \in L \setminus T$, there exists a quorum $Q_i \in \mathcal{Q}(p_i, \mathbb{F})$ for p_i such that $Q_i \subseteq L \setminus T$.

If we consider an execution e tolerated by a league L , where A is the set of faulty processes, the consistency property of L implies that, at any time, any two quorums of correct processes in L have some correct process in common. This is similar to the consistency property of symmetric and asymmetric Byzantine quorum systems [5, 14].

Moreover, since the set of faulty processes A is tolerated by L , by the availability property of L , every correct process in L has a quorum in \mathbb{F} consisting of only correct processes.

► **Example 20.** Observe that the set Π as introduced in Example 4 is a league. In fact, for every set T tolerated by Π , i.e., \emptyset , $\{p_1\}$, $\{p_4\}$ and $\{p_1, p_4\}$, for every two processes $p_i, p_j \in \Pi \setminus T$ and for every two quorums $Q_i \in \mathcal{S}_i$ and $Q_j \in \mathcal{S}_j$ as in Example 13, it holds $(Q_i \cap Q_j) \setminus T \neq \emptyset$, and for every $p_i \in \Pi \setminus T$, there exists a quorum $Q_i \in \mathcal{S}_i$ such that $Q_i \subseteq \Pi \setminus T$.

The following lemma shows that the union of two intersecting leagues L_1 and L_2 is again a league, assuming that for every set T tolerated by both the leagues, L_1 and L_2 have a common process not in T .

► **Lemma 21.** *If L_1 and L_2 are two leagues such that $L_1 \cap L_2 \neq \emptyset$ and such that for every set T tolerated by $L_1 \cup L_2$, there exists a process $p_k \in (L_1 \cap L_2) \setminus T$, then $L_1 \cup L_2$ is a league.*

Proof. Let L_1 and L_2 be two leagues such that $L_1 \cap L_2 \neq \emptyset$. For every T tolerated by $L_1 \cup L_2$ (and so, tolerated by L_1 and L_2 , independently), for every $p_i \in L_1 \setminus T$ and $p_j \in L_2 \setminus T$, for every two T -resilient views \mathbb{V} and \mathbb{V}' for p_i and p_j , respectively, let $Q_i \in \mathcal{Q}(p_i, \mathbb{V})$ and $Q_j \in \mathcal{Q}(p_j, \mathbb{V}')$ be two quorums for p_i and p_j , respectively. Observe that, by assumption, for every tolerated set T by $L_1 \cup L_2$ there exists a process $p_k \in (L_1 \cap L_2) \setminus T$. Let $p_k \in L_1 \cap L_2$ and let $Q_k \in \mathcal{Q}(p_k, \mathbb{V})$ be a quorum for p_k such that $Q_k \subseteq L_1$, according to availability property of L_1 . From consistency property of L_2 , $(Q_k \cap Q_j) \setminus T \neq \emptyset$ and every process in this intersection belongs to L_1 . Observe that, Q_j is a quorum for every of its member. This implies that Q_j is a quorum for every process in $(Q_k \cap Q_j) \setminus T$ and every process in $(Q_k \cap Q_j) \setminus T$ has quorum in L_1 . Moreover, $(Q_k \cap Q_i) \setminus T \neq \emptyset$. It follows that $(Q_i \cap Q_j) \setminus T \neq \emptyset$.

Finally, by availability property of L_1 and L_2 , for every tolerated set T by L_1 and L_2 and for every process $p_i \in L_1 \setminus T$ and $p_j \in L_2 \setminus T$, eventually there exists a quorum $Q_i \in \mathcal{Q}(p_i, \mathbb{F})$ for p_i and a quorum $Q_j \in \mathcal{Q}(p_j, \mathbb{F})$ for p_j such that $Q_i \subseteq L_1 \setminus T$ and $Q_j \subseteq L_2 \setminus T$, respectively. If $p_i = p_j \in L_1 \cap L_2$, then there exists a quorum Q_i for p_i such that $Q_i \subseteq (L_1 \cup L_2) \setminus T$. ◀

In the following lemma we show that we can characterize the consistency property of a league just by considering worst-case views. Intuitively, this result relies on the observation that every T -resilient view can be seen as extensions of worst-case views with respect to $T \subseteq \Pi$, in the sense that a T -resilient view can be obtained by starting from a worst-case view with respect to T and by considering the non-empty configurations received by processes in T .

► **Lemma 22.** *The consistency property a league L holds if and only if for every set $T \subseteq \Pi$ tolerated by L , for every two worst-case views \mathbb{V}_T^* and \mathbb{V}'_T^* with respect to T , for every two processes $p_i, p_j \in L \setminus T$, and for every two quorums $Q_i \in \mathcal{Q}(p_i, \mathbb{V}_T^*)$ and $Q_j \in \mathcal{Q}(p_j, \mathbb{V}'_T^*)$ it holds $(Q_i \cap Q_j) \setminus T \neq \emptyset$.*

Proof. Let us assume that the consistency property of a league L holds. Since the property holds for every pair of views, it must hold also for worst-case views. The implication easily follows.

Let us now assume that for every set $T \subseteq \Pi$ tolerated by L , for every two worst-case views \mathbb{V}_T^* and \mathbb{V}'_T^* with respect to T , for every two processes $p_i, p_j \in L \setminus T$, and for every two quorums $Q_i \in \mathcal{Q}(p_i, \mathbb{V}_T^*)$ and $Q_j \in \mathcal{Q}(p_j, \mathbb{V}'_T^*)$ it holds $(Q_i \cap Q_j) \setminus T \neq \emptyset$. Observe that, given a quorum $Q_i \in \mathcal{Q}(p_i, \mathbb{V}_T^*)$ for p_i in a worst-case view \mathbb{V}_T^* , all the quorums obtained by also considering all the possible configurations received from processes in T that are not in

17:10 Quorum Systems in Permissionless Networks

\mathbb{V}_T^* do contain Q_i . Moreover, there cannot exist a T -resilient view that does not consist of configurations of a worst-case view with respect to T . If this was the case, then by removing configurations received from processes in T one would obtain a worst-case view with respect to T , reaching a contradiction. So, all the quorums obtained from T -resilient views will also intersect in processes that are not contained in T . ◀

Now we show how a league can be abstracted and defined without considering views. This will be useful in Section 5 when we compare our model with other permissionless models. First, we introduce the following definitions.

► **Definition 23** (Inclusive up to). *A set $I \subseteq \Pi$ is inclusive up to a set $T \subseteq \Pi$ if and only if for every $p_i \in I \setminus T$, process p_i has a slice in I .*

If we consider an execution e with set of faulty processes A then a set of processes I is inclusive up to A if and only if every correct process in I has a slice contained in I .

► **Definition 24** (Rooted at). *A set $R \subseteq \Pi$ is rooted at a process p_i if and only if p_i has a slice in R . A set $R \subseteq \Pi$ is rooted in a set $T' \subseteq \Pi$ whenever R is rooted at a member of T' .*

► **Lemma 25.** *If \mathbb{V} is a T -resilient view and $Q_i \in \mathcal{Q}(p_i, \mathbb{V})$ for some process p_i , then Q_i is inclusive up to T and rooted at p_i .*

Proof. If \mathbb{V} is a T -resilient view then, by Definition 16, processes outside T do not lie about their assumptions. By definition of a quorum Q_i for a process p_i in a view \mathbb{V} , every process in Q_i , and so in $Q_i \setminus T$, has a slice in Q_i . This implies that Q_i is inclusive up to T and rooted at p_i . ◀

In the following lemma we show that given a set of processes T tolerated by $L \subseteq \Pi$, for every set of processes I inclusive up to T and rooted at $p_i \in L \setminus T$ it is possible to find a T -resilient view in which I is a quorum for p_i . This view is a worst-case view with respect to T .

► **Lemma 26.** *Let L be a set of processes. For every set $T \subseteq \Pi$ tolerated by L , if $I \subseteq \Pi$ is a set inclusive up to T and rooted at $p_i \in L \setminus T$, then there is a T -resilient view in which I is a quorum for p_i .*

Proof. Let $T \subseteq \Pi$ be a tolerated set by a set of processes L and let $I \subseteq \Pi$ be a set inclusive up to T and rooted at $p_i \in L \setminus T$. This implies that p_i and every other process $p_j \in I \setminus T$ have a slice in I . Let us consider the worst-case view \mathbb{V}_T^* with respect to T . Clearly, \mathbb{V}_T^* is T -resilient. This implies that, in \mathbb{V}_T^* , the set I is a quorum for p_i , i.e., $I \in \mathcal{Q}(p_i, \mathbb{V}_T^*)$. ◀

► **Remark 27.** Observe that given a set of processes L and a worst-case view \mathbb{V}_T^* with respect to a set $T \subseteq \Pi$ tolerated by L , every quorum $Q_i \in \mathcal{Q}(p_i, \mathbb{V}_T^*)$ for $p_i \in L \setminus T$ is inclusive up to T and rooted at p_i .

Moreover, given the set \mathcal{I} of all the sets $I \subseteq \Pi$ inclusive up to T and rooted at $p_i \in L \setminus T$, the set \mathcal{I} contains all the quorums $Q_i \in \mathcal{Q}(p_i, \mathbb{V})$ for every T -resilient view \mathbb{V} . In fact, by Definition 23, given a set of processes I inclusive up to a set of processes T , the requirement of having a slice in I is only for processes in $I \setminus T$, leaving processes in $T \cap I$ with no requirements on the choice of their slices.

However, given a T -resilient view \mathbb{V} , by Definition 11, a quorum Q_i for p_i requires instead every process in Q_i to have a slice contained in Q_i . This means that given a T -resilient view \mathbb{V} , quorum Q_i for p_i is contained in \mathcal{I} , being a special case of inclusive set up to T .

► **Lemma 28.** *The consistency property of a league L holds if and only if for every set $T \subseteq \Pi$ tolerated by L , and for every two sets $I \subseteq \Pi$ and $I' \subseteq \Pi$ that are rooted at $L \setminus T$ and inclusive up to T it holds $(I \cap I') \setminus T \neq \emptyset$.*

Proof. Let us assume that the consistency property of a league L holds. Suppose by contradiction that there is a set $T \subseteq \Pi$ tolerated by L and two sets $I \subseteq \Pi$ and $I' \subseteq \Pi$ that are inclusive up to T and rooted at $L \setminus T$ in p_i and p_j , respectively, such that $(I \cap I') \setminus T = \emptyset$.

By Lemma 26, there are a T -resilient view \mathbb{V} in which I is a quorum for p_i and a T -resilient view \mathbb{V}' in which I' is a quorum for p_j and we reached a contradiction.

Let us now assume that for every set $T \subseteq \Pi$ tolerated by L , and for every two sets $I \subseteq \Pi$ and $I' \subseteq \Pi$ that are inclusive up to T and rooted at $L \setminus T$ it holds $(I \cap I') \setminus T \neq \emptyset$.

Let \mathcal{I} and \mathcal{I}' be the sets of all the sets $I \subseteq \Pi$ and $I' \subseteq \Pi$ inclusive up to T and rooted at $p_i \in L \setminus T$ and $p_j \in L \setminus T$, respectively. The proof follows from the reasoning in Remark 27: for every two T -resilient views \mathbb{V} and \mathbb{V}' , every quorum $Q_i \in \mathcal{Q}(p_i, \mathbb{V})$ for p_i is contained in \mathcal{I} and every quorum $Q_j \in \mathcal{Q}(p_j, \mathbb{V})$ for p_j is contained in \mathcal{I}' . ◀

► **Lemma 29.** *The availability property of a league L holds if and only if for every set $T \subseteq \Pi$ tolerated by L , every member of $L \setminus T$ has a survivor set in $L \setminus T$.*

Proof. Let us assume that the availability property of a league L holds, i.e., for every set of processes T tolerated by L and for every $p_i \in L \setminus T$, there exists a quorum $Q_i \in \mathcal{Q}(p_i, \mathbb{F})$ for p_i such that $Q_i \subseteq L \setminus T$. This means that, by Definition 6, every process in $L \setminus S$ has a survivor set in $L \setminus S$.

Let us now assume that for every set $T \subseteq \Pi$ tolerated by L , every member of $L \setminus T$ has a survivor set S in $L \setminus T$. Let p_i be a process in L , by Definition 11 we have that $S \in \mathcal{Q}(p_i, \mathbb{F})$ for p_i . The proof follows. ◀

5 Comparison with Other Models

In this section we compare our model with the classic model based on fail-prone systems [14], with the asymmetric model [5, 8], with the federated Byzantine agreement system model [16], and with the personal Byzantine quorum system model [12].

5.1 Comparison with Fail-Prone Systems

We show that classic fail-prone systems and quorums can be understood as a special case of our model, when every process knows the entire system and assumes the same, global fail-prone system.

Let $\Pi = \{p_1, \dots, p_n\}$ be a set of processes. A Byzantine quorum system for a fail-prone system \mathcal{F} (Definition 1) satisfies (Consistency) $\forall Q_1, Q_2 \in \mathcal{Q}, \forall F \in \mathcal{F} : Q_1 \cap Q_2 \not\subseteq F$; and (Availability) $\forall F \in \mathcal{F} : \exists Q \in \mathcal{Q} : F \cap Q = \emptyset$. Moreover, a Byzantine quorum system \mathcal{Q} for \mathcal{F} exists if and only if the Q^3 -condition holds [10, 14], which means that for every $F_1, F_2, F_3 \in \mathcal{F} : \Pi \not\subseteq F_1 \cup F_2 \cup F_3$. In particular, if $Q^3(\mathcal{F})$ holds, then $\overline{\mathcal{F}}$, the bijective complement of \mathcal{F} , is a Byzantine quorum system; this is the Byzantine quorum system consisting of survivor sets [11]. Notice that, in a classic system, the failures that are tolerated by the processes are all possible subsets of fail-prone sets in \mathcal{F} and we have $P_i = \Pi$ for every $p_i \in \Pi$. Every process also knows the global quorum system.

We define a bijective function f between the set of fail-prone systems and a subset of PFPS such that $f(\mathcal{F}) = [(\Pi, \mathcal{F}), \dots, (\Pi, \mathcal{F})]$ with n repetitions and we notice that in classic fail-prone systems there is only one view, namely $\mathbb{V} = f(\mathcal{F})$.

We define the quorum function $\Omega : \Pi \times \Upsilon \rightarrow 2^\Pi$ such that for every process $p_i \in \Pi$, $\Omega(p_i, f(\mathcal{F})) = \overline{\mathcal{F}}$. Observe that any set in $\overline{\mathcal{F}}$ is a slice of every process $p_i \in \Pi$ according to Definition 5. In the next theorem we consider this quorum function and show that, given some assumptions on \mathcal{F} , any set in $\overline{\mathcal{F}}$ is also a quorum for every process $p_i \in \Pi$ according to Definition 11.

► **Theorem 30.** *Let Π be the set of all processes and \mathcal{F} the fail-prone system for Π . Then $Q^3(\mathcal{F})$ holds if and only if Π is a league for the quorum function Ω in $f(\mathcal{F})$.*

Proof. Let us first assume that $Q^3(\mathcal{F})$ holds. This means that for every $F_1, F_2, F_3 \in \mathcal{F}$, $\Pi \not\subseteq F_1 \cup F_2 \cup F_3$. It follows that $\overline{\mathcal{F}}$ is a quorum system for \mathcal{F} . Consistency property of $\overline{\mathcal{F}}$ implies that for every tolerated set F , for every two processes p_i and p_j in $\Pi \setminus F$ and for every two quorums $Q_i \in \Omega(p_i, f(\mathcal{F}))$ and $Q_j \in \Omega(p_j, f(\mathcal{F}))$ for p_i and p_j , respectively, it holds that $(Q_i \cap Q_j) \setminus T \neq \emptyset$.

The availability property of $\overline{\mathcal{F}}$ implies that for every set $F \in \mathcal{F}$ tolerated by Π , every process $p_i \in \Pi \setminus F$ has a quorum in $\Pi \setminus F$: given F , there exists a quorum $Q \in \Omega(p_i, f(\mathcal{F}))$ such that $Q \cap F = \emptyset$ and $Q \subseteq \Pi \setminus F$. It follows that Π is a league for the quorum function Ω in $f(\mathcal{F})$.

Let us now assume that Π is a league for the quorum function Ω in $f(\mathcal{F})$. The consistency property of Π implies that for every T tolerated by Π (which are all the sets in \mathcal{F}), for every two processes p_i and p_j in $\Pi \setminus T$, for every two quorums $Q_i \in \overline{\mathcal{F}}$ and $Q_j \in \overline{\mathcal{F}}$ for p_i and p_j , respectively, it holds $(Q_i \cap Q_j) \setminus T \neq \emptyset$. Moreover, by availability property of Π there exists a quorum in $\Pi \setminus T$ (which is the same for every process $p_i \notin T$). This implies that, for every fail-prone set $F \in \mathcal{F}$, there is a quorum Q_i such that $Q_i \cap F = \emptyset$.

These two facts imply that $\overline{\mathcal{F}}$ is a classic Byzantine quorum system for \mathcal{F} and so $Q^3(\mathcal{F})$ holds. ◀

5.2 Comparison with Asymmetric Fail-Prone Systems

In the asymmetric model [8], every process is free to express its own trust assumption about the processes in one common globally known system through a subjective fail-prone system.

An asymmetric fail-prone system $\mathbb{F}' = [\mathcal{F}'_1, \dots, \mathcal{F}'_n]$ consists of an array of fail-prone systems, where $\mathcal{F}'_i \subseteq 2^\Pi$ denotes the trust assumption of p_i .

An *asymmetric Byzantine quorum system* for \mathbb{F}' [5], is an array of collections of sets $\mathcal{Q}' = [\mathcal{Q}'_1, \dots, \mathcal{Q}'_n]$, where $\mathcal{Q}'_i \subseteq 2^\Pi$ for $i \in [1, n]$. The set $\mathcal{Q}'_i \subseteq 2^\Pi$ is called the *quorum system of p_i* and any set $Q_i \in \mathcal{Q}'_i$ is called a *quorum (set) for p_i* . Moreover, defining $\mathcal{F}'^* = \{F' \mid F' \subseteq F, F \in \mathcal{F}'\}$, the following conditions hold: (Consistency) $\forall i, j \in [1, n], \forall Q_i \in \mathcal{Q}'_i, \forall Q_j \in \mathcal{Q}'_j, \forall F_{ij} \in \mathcal{F}'^*_i \cap \mathcal{F}'^*_j : Q_i \cap Q_j \not\subseteq F_{ij}$; and (Availability) $\forall i \in [1, n], \forall F_i \in \mathcal{F}'_i : \exists Q_i \in \mathcal{Q}'_i : F_i \cap Q_i = \emptyset$.

The Q^3 -condition in the classic model can be generalized as follows: we say that \mathbb{F}' satisfies the B^3 -condition [8], abbreviated as $B^3(\mathbb{F}')$, whenever it holds for all $i, j \in [1, n]$ that $\forall F_i \in \mathcal{F}'_i, \forall F_j \in \mathcal{F}'_j, \forall F_{ij} \in \mathcal{F}'^*_i \cap \mathcal{F}'^*_j : \Pi \not\subseteq F_i \cup F_j \cup F_{ij}$.

An asymmetric fail-prone system \mathbb{F}' satisfying the B^3 -condition is sufficient and necessary [5] for the existence of a corresponding asymmetric quorum system $\mathcal{Q}' = [\mathcal{Q}'_1, \dots, \mathcal{Q}'_n]$, with $\mathcal{Q}'_i = \overline{\mathcal{F}'_i}$. Processes in this model are classified in three different types, given an execution e with faulty set A : a process $p_i \in A$ is *faulty*, a correct process p_i for which $A \notin \mathcal{F}'^*_i$ is called *naive*, while a correct process p_i for which $A \in \mathcal{F}'^*_i$ is called *wise*.

Finally, a *guild* \mathcal{G} for A is a set of wise processes that contains at least one quorum for each member.

Let Π be a set of processes in the asymmetric model and $\mathbb{F}' = [\mathcal{F}'_1, \dots, \mathcal{F}'_n]$ be an asymmetric fail-prone system. Define the function g from asymmetric fail-prone systems to PFPS such that $g(\mathbb{F}') = [(\Pi, \mathcal{F}'_1), \dots, (\Pi, \mathcal{F}'_n)]$. Observe that, in an asymmetric system,

the failures that may be tolerated by the processes are possible subsets of fail-prone sets in the fail-prone systems of \mathbb{F}' and $P_i = \Pi$ for every $p_i \in \Pi$. Moreover, as in the classic model, there is only one view, which is $\mathbb{V} = g(\mathbb{F}')$.

We define the quorum function $\mathcal{Q} : \Pi \times \Upsilon \rightarrow 2^\Pi$ such that for every guild $\mathcal{G} \subseteq 2^\Pi$, if $p_i \in \mathcal{G}$ then $\mathcal{Q}(p_i, g(\mathbb{F}')) = \{\mathcal{G}, \Pi\}$, otherwise $\mathcal{Q}(p_i, g(\mathbb{F}')) = \{\Pi\}$.

Observe that a quorum in the asymmetric model is a slice according to Definition 5 and, for every process $p_i \in \Pi$, every set in $\mathcal{Q}(p_i, g(\mathbb{F}'))$ is a quorum according to Definition 11.

Through the following theorem we establish the relationship between the asymmetric model and the permissionless model.

► **Theorem 31.** *Let us consider an asymmetric model among a set Π of processes with asymmetric fail-prone system \mathbb{F}' . If $B^3(\mathbb{F}')$ holds and Π tolerates some sets $T \subseteq \Pi$, then there exists a quorum function \mathcal{Q} such that Π is a league in $g(\mathbb{F}')$.*

Proof. Let us assume that Π tolerates some sets $T \subseteq \Pi$ and let us consider the quorum function \mathcal{Q} define in this section in the context of the asymmetric model. This means that, for every set T tolerated by Π , every process $p_i \in \Pi \setminus T$ has a slice contained in $\Pi \setminus T$. This implies that in every execution in which T is the set of faulty processes, every process in $\Pi \setminus T$ is wise and $\Pi \setminus T$ is a guild.

Moreover, let us also assume that $B^3(\mathbb{F}')$ holds. This implies the existence of an asymmetric Byzantine quorum system \mathcal{Q}' such that for every set T tolerated by Π , for every two processes p_i and p_j in $\Pi \setminus T$ and for every two quorums $Q_i \in \mathcal{Q}'_i$ and $Q_j \in \mathcal{Q}'_j$ for p_i and p_j , respectively, it holds that $(Q_i \cap Q_j) \setminus T \neq \emptyset$. Observe that the set $\Pi \setminus T \in \mathcal{Q}(p_i, g(\mathbb{F}'))$ is a quorum in the permissionless model for every $p_i \in \Pi \setminus T$ according to Definition 11. This implies that Π satisfies availability property of a league.

Finally, for the consistency property observe that for every process $p_i \in \Pi$, the set system $\mathcal{Q}(p_i, g(\mathbb{F}'))$ satisfies Definition 11; by construction we have at most only two quorums for every p_i which are Π and $\Pi \setminus T$ both satisfying Definition 11. Consistency of \mathcal{Q}' implies intersection among the quorums in $\mathcal{Q}(p_i, g(\mathbb{F}'))$, for every process in $\Pi \setminus T$.

It follows that Π is a league for the quorum function \mathcal{Q} in $g(\mathbb{F}')$. ◀

Theorem 31 shows a relation between the asymmetric model and the permissionless model. In particular, if $B^3(\mathbb{F}')$ holds and Π tolerates some sets T , then the quorum function \mathcal{Q} makes Π a league. However, we could have scenarios in which only a subset of Π tolerates some sets T . In particular, we have the following result.

► **Lemma 32.** *Let $\Pi = \{p_1, \dots, p_n\}$ be a set of processes, \mathbb{F}' be an asymmetric fail-prone system over Π and $g(\mathbb{F}')$ the corresponding PFPS as described in the text. Moreover, let us consider an execution e with set of faulty processes A with guild \mathcal{G} . Then, \mathcal{G} is the only set that tolerates e .*

Proof. By definition of guild, every process in \mathcal{G} is wise and has a quorum contained in \mathcal{G} . Observe that, given a wise process p_i , there exists a fail-prone set $F \in \mathcal{F}'_i$ in \mathbb{F}' such that $A \subseteq F$. Moreover, a quorum Q_i for p_i in the asymmetric model satisfies Definition 5 and it is then a slice of p_i . This implies that every process in \mathcal{G} has its assumptions satisfied according to Definition 3. Moreover, every process in \mathcal{G} has a slice contained in \mathcal{G} . ◀

In the following lemma we characterize a link between the notion of a guild, in a given execution, and a league.

► **Lemma 33.** *Let us consider an asymmetric Byzantine quorum system \mathcal{Q}' and a guild \mathcal{G} in any execution with set of faulty processes A . Then, \mathcal{G} is a league for the quorum function \mathcal{Q} in $g(\mathbb{F}')$.*

Proof. The result follows from Theorem 31 by applying the same reasoning with \mathcal{G} instead of $\Pi \setminus T$ as a guild. ◀

In the following lemma we show a scenario where no asymmetric Byzantine quorum systems exist but it is possible to find a league for \mathcal{Q} in $g(\mathbb{F}')$.

► **Lemma 34.** *There exists an asymmetric fail-prone system \mathbb{F}' such that:*

- *there is no asymmetric Byzantine quorum system for \mathbb{F}' , but*
- *there exists a quorum function \mathcal{Q} that make Π a league in $g(\mathbb{F}')$.*

Proof. We prove this lemma through an example with four processes. Consider an asymmetric fail-prone system \mathbb{F}'_4 over four processes $p_1, p_2, p_3,$ and p_4 with $\mathcal{F}'_1 = \{\{p_3, p_4\}\}$, $\mathcal{F}'_2 = \{\{p_1, p_4\}\}$, $\mathcal{F}'_3 = \{\{p_1, p_4\}\}$, and $\mathcal{F}'_4 = \{\{p_1, p_2\}\}$, as in Example 4.

Observe that, by the availability property of an asymmetric Byzantine quorum system, p_1 must have a quorum in $\{p_1, p_2\}$ and p_4 must have a quorum in $\{p_3, p_4\}$. Since $\{p_1, p_2\}$ and $\{p_3, p_4\}$ are disjoint, it is impossible to satisfy the consistency property. Thus, there does not exist any asymmetric Byzantine quorum system for \mathbb{F}'_4 . Another way to see this is by observing that $B^3(\mathbb{F}'_4)$ does not hold: $\{p_3, p_4\} \cup \{p_1, p_2\} = \Pi$.

However, as shown in Example 20, Π is a league in $g(\mathbb{F}'_4)$. So, there is no asymmetric Byzantine quorum system for \mathbb{F}'_4 but \mathcal{Q} makes Π a league in $g(\mathbb{F}'_4)$. ◀

5.3 Comparison with Federated Byzantine Agreement Systems

The federated Byzantine agreement system (FBAS) model has been introduced by Mazières [16] in the context of the Stellar white paper. Differently from the models presented before in this section, the FBAS model is a permissionless model, where processes, each with an initial set of known processes, continuously discover new processes. In a FBAS, every process p_i chooses a set of slices, which are sets of processes sufficient to convince p_i of agreement and a set of processes Q_i is a quorum for p_i whenever p_i has at least one slice inside Q_i and every member of Q_i has a slice that is a subset of Q_i . In particular, a quorum Q_i is a quorum for every of its members. However, despite the permissionless nature of a FBAS, a global intersection property among quorums is required for the analysis of the Stellar Consensus Protocol (SCP), and the scenario with disjoint quorums is not considered by Mazières.

A central notion in FBAS is that of *intact* set; given a set of processes Π , an execution with set of faulty processes A and a set of correct processes $\mathcal{W} = \Pi \setminus A$, a set of processes $\mathcal{I} \subseteq \mathcal{W}$ is an *intact set* [9, 12] when the following conditions hold: (Consistency) for every two processes p_i and p_j in \mathcal{I} and for every two quorums Q_i and Q_j for p_i and p_j , respectively, $Q_i \cap Q_j \cap \mathcal{I} \neq \emptyset$; and (Availability) \mathcal{I} is a quorum for every of its members.

Every process in \mathcal{I} is called *intact*, while every process in $\Pi \setminus \mathcal{I}$ (correct or faulty) is called *befouled* and some properties of the Stellar Consensus Protocol are guaranteed only for intact processes. Moreover, the union of two intersecting intact sets is an intact set. Finally, by requiring a system-wide intersection among quorums (as in the case of the SCP) one obtains an unique intact set (Lemma 34, [9]).

We first show that our model generalizes the FBAS model by showing that a quorum in FBAS satisfies Definition 11.

In FBAS a notion of fail-prone system is missing and definitions are given with respect to an execution with a fixed set of faulty processes A . However, because processes define slices, an implicit fail-prone system for every process can be derived.

In particular, given a set of processes $\Pi = \{p_1, p_2, \dots\}$, every process in Π defines its slices based on a known subset $P_i \subseteq \Pi$ by p_i and S_i is a slice for $p_i \in \Pi$ if and only if $p_i \in S_i$ and $S_i \subseteq P_i$ [9]. Let $\mathcal{S}_i \subseteq 2^\Pi$ be the set of slices of p_i , we can derive the following definition.

► **Definition 35** (Federated fail-prone system). *A set $F \subseteq \Pi$ is a fail-prone set of p_i if and only if there exists a slice $S_i \in \mathcal{S}_i$ of p_i such that $F = P_i \setminus S_i$. The set $\mathcal{F}_i'' \subseteq 2^\Pi$ of all the fail-prone sets of p_i is called fail-prone system of p_i . Finally, we call the set $\mathbb{F}'' = [(P_1, \mathcal{F}_1''), (P_2, \mathcal{F}_2''), \dots]$ the federated fail-prone system.*

In a FBAS, processes discover other processes' slices during an execution and so p_i implicitly learns other processes' federated fail-prone sets. Moreover, correct processes do not lie about their slices [9, 16].

It is easy to observe that given different sets of slices received from different processes, Definition 35 implies Definition 9, obtaining a notion of view \mathbb{V} in the FBAS model, and, because correct processes do not lie about their slices, Definition 16. We define the set Υ' to be the set of all the possible views in the FBAS model.

Given the notion of view in the FBAS model, we define the quorum function $\mathcal{Q} : \Pi \times \Upsilon' \rightarrow 2^\Pi$ such that $\mathcal{Q}(p_i, \mathbb{V})$ contains all the sets Q_i , called quorums, with $p_i \in Q_i$ and such that every process $p_j \in Q_i$ has a slice in Q_i . So, a quorum as defined by Mazières [16] satisfies Definition 11. Finally, in the FBAS model we introduce the notion of survivor set as defined in Definition 6.

In the following theorem we show that, by assuming a stronger consistency property for a league L , i.e., that the intersection among any two quorums of any two correct processes in the league contains some correct member of the league, then L is an intact set in every execution tolerated by L .

► **Theorem 36.** *Let L be a league for the quorum function \mathcal{Q} and let us assume that for every set $T \subseteq \Pi$ tolerated by L , for every two T -resilient views \mathbb{V} and \mathbb{V}' , for every two processes $p_i, p_j \in L \setminus T$, and for every two quorums $Q_i \in \mathcal{Q}(p_i, \mathbb{V})$ and $Q_j \in \mathcal{Q}(p_j, \mathbb{V}')$ it holds $(Q_i \cap Q_j \cap L) \setminus T \neq \emptyset$, then L is an intact set for every every set $T \subseteq \Pi$ tolerated by L .*

Proof. Let L be a league for the quorum function \mathcal{Q} and let T be a set of processes tolerated by L . If for every two T -resilient views \mathbb{V} and \mathbb{V}' , for every two processes $p_i, p_j \in L \setminus T$, and for every two quorums $Q_i \in \mathcal{Q}(p_i, \mathbb{V})$ and $Q_j \in \mathcal{Q}(p_j, \mathbb{V}')$ it holds $(Q_i \cap Q_j \cap L) \setminus T \neq \emptyset$, then the consistency property of intact sets follows. The availability property of an intact set follows by observing that, in \mathbb{F} , the set $L \setminus T$ is a quorum for every of its members. ◀

Observe that without the stronger consistency property assumed in Theorem 36, since quorums of correct processes in L may intersect in correct processes (not necessarily in L), it may be the case that L is not an intact set.

5.4 Comparison with Personal Byzantine Quorum Systems

The personal Byzantine quorum system (PBQS) model has been introduced by Losa et al. [12] in the context of Stellar consensus aiming at removing the system-wide intersection property among quorums required by Mazières [16] for the SCP.

In the PBQS model a quorum for p_i is a non-empty set of processes Q_i such that if Q_i is a quorum for p_i and $p_j \in Q_i$, then there exists a quorum Q_j for p_j such that $Q_j \subseteq Q_i$. In other terms, a quorum Q_i for some process p_i must contain a quorum for every one of its

17:16 Quorum Systems in Permissionless Networks

members. Losa et al. point out that a global consensus among processes may be impossible since the full system membership is not known by the processes, and define the notion of *consensus cluster* as a set of processes that can instead solve a *local* consensus, i.e., consensus among the processes in a consensus cluster can be solved. In particular, given an execution with set of faulty processes A , a set of correct processes \mathcal{C} is a consensus cluster when the following conditions hold: (Consistency) for every two processes p_i and p_j in \mathcal{C} and for every two quorums Q_i and Q_j for p_i and p_j , respectively, $Q_i \cap Q_j \not\subseteq A$; and (Availability) for every $p_i \in \mathcal{C}$ there exists a quorum Q_i for p_i such that $Q_i \subseteq \mathcal{C}$. Losa et al. prove that the union of two intersecting consensus clusters is a consensus cluster and that maximal consensus clusters are disjoint. The latter implies that maximal consensus clusters might diverge from each other.

In the following we show a relationship between the notions of league and consensus cluster. To do so, we first show that a quorum Q_i for p_i as defined in Definition 11 is also a quorum for p_i in the PBQS model.

► **Lemma 37.** *Let $Q_i \in \mathcal{Q}(p_i, \mathbb{V})$ be a quorum for a process p_i in a view \mathbb{V} according to Definition 11. Then Q_i is a quorum for p_i in the PBQS model.*

Proof. Definition 11 implies that Q_i is a quorum for every of its members. This means that for every process $p_j \in Q_i$, the set Q_i is a quorum for p_j such that $Q_i \subseteq Q_j$. The result follows. ◀

In the following result we show that, given a league L , for every set $T \subseteq \Pi$ tolerated by L , the set $L \setminus T$ is a consensus cluster.

► **Theorem 38.** *Let L be a league for the quorum function \mathcal{Q} . Then, for every set $T \subseteq \Pi$ tolerated by L , the set $L \setminus T$ is a consensus cluster.*

Proof. Let L be a league for the quorum function \mathcal{Q} and let T be a set of processes tolerated by L . Lemma 37 implies that for every process p_i and for every view \mathbb{V} , all the quorums in $\mathcal{Q}(p_i, \mathbb{V})$ for p_i are quorums in the PBQS model. So, the consistency and availability properties of a league imply that $L \setminus T$ satisfies the consistency and availability properties of a consensus cluster, making $L \setminus T$ a consensus cluster. ◀

6 Permissionless Reliable Broadcast

In this section we show how the Bracha broadcast [2], protocol that implements Byzantine reliable broadcast, can be adapted to work in our model. First, we introduce the following definitions and results.

► **Definition 39** (Blocking set). *A set $B \subseteq \Pi$ is said to block a process p_i if B intersects every slice of p_i .*

► **Definition 40** (Inductively blocked). *Given a set of processes B , the set of processes inductively blocked by B , denoted by B^+ , is the smallest set closed under the following rules:*

1. $B \subseteq B^+$; and
2. if a process p_i is blocked by B^+ , then $p_i \in B^+$.

As a consequence of Definition 40, given an execution, the set B^+ can be obtained by repeatedly adding to it all the processes that are blocked by $B^+ \cup B$. Eventually no more processes will be added to B^+ .

Moreover, given an execution e with set of faulty processes A , if a league L tolerates A , then processes in $L \setminus A$ cannot be inductively blocked by A . This is shown in the following lemma.

► **Lemma 41.** *Let L be a league and T be a set tolerated by L . Then, no process in $L \setminus T$ is inductively blocked by T , i.e., $T^+ \cap (L \setminus T) = \emptyset$.*

Proof. Let us assume that $T^+ \cap (L \setminus T) \neq \emptyset$. This means that there exists a process $p_i \in L \setminus T$ that is blocked by T^+ , i.e., T^+ intersects every slice of p_i , including the slice contained in the quorum $Q_i \subseteq L \setminus T$ for p_i . Clearly, $(L \setminus T) \cap T = \emptyset$, and this means that there exists a set T' with $T' \subseteq T^+ \setminus T$ such that T' intersects every slice of p_i , including the slice contained in the quorum for p_i consisting only of correct processes in L . This means that we can find a process $p_j \in T'$ with $p_j \in L \setminus T$ and p_j blocked by T . Since L is a league, process p_j must have a slice in $L \setminus T$. However, T cannot intersect every slice of p_j because $L \setminus T$ is disjoint from T . We reached a contradiction. ◀

Intuitively, starting from $A^+ = \emptyset$, we first consider the processes that are blocked by A . Trivially, every process in A is blocked by A , and so $A^+ = A$. Moreover, no process in $L \setminus A$ can be blocked by A . If this was the case, then there would exist a process $p_i \in L \setminus A$ such that A intersected all of its slices, including the slice contained in the quorum $Q_i \subseteq L \setminus A$, which we know to exist due to the availability property of L . So, only processes p_j not in $L \setminus A$ can be blocked by A . Let p_j be such process. This means that $A \cup \{p_j\} \subseteq A^+$. Now, we can repeat the same reasoning, by considering all the processes blocked by $A \cup \{p_j\}$. Again, no processes in $L \setminus A$ can be blocked by $A \cup \{p_j\}$. In fact, if $A \cup \{p_j\}$ blocked a process $p_k \in L \setminus A$, then every slice of p_k would contain p_j , including the slice contained in $L \setminus A$. However, this would imply that $p_j \in L \setminus A$ which would contradict the fact that p_j is a process not in $L \setminus A$.

In the following theorem we show that if a correct process p_i in a league L is blocked by a set B , then $B = B \cup \{p_i\}$ blocks another process $p_j \notin B \cup A$. Then, $B' = B \cup \{p_j\}$ blocks another process $p_k \notin B' \cup A$ and so on, until, eventually, every correct process in the league is blocked.

► **Theorem 42 (Cascade theorem).** *Consider the quorum function \mathcal{Q} , a league L , and a set $T \subseteq \Pi$ tolerated by L . Moreover, let us consider a process $p_i \in L \setminus T$, a T -resilient view \mathbb{V} for p_i , a quorum $Q_i \in \mathcal{Q}(p_i, \mathbb{V})$, and a set $B \subseteq \Pi$ disjoint from T such that $Q_i \setminus T \subseteq B$. Then, either $L \setminus T \subseteq B$ or there exists a process $p_j \notin B \cup T$ that is blocked by B .*

Proof. It suffices to assume by contradiction that $L \setminus (B \cup T) \neq \emptyset$ and that, for every $p_j \notin B \cup T$, process p_j has a slice disjoint from B . This implies that $S = \overline{B \cup T}$ is a survivor set of every process $p_j \in S$; since $L \setminus (B \cup T) \neq \emptyset$, this includes also at least one process $p_j \in L \setminus (B \cup T)$.

Let us consider such a process $p_j \in L \setminus (B \cup T)$ and consider the view \mathbb{V}' for p_j such that: (1) for every $p_k \notin T$, $\mathbb{V}'[k] = \mathbb{F}[k]$; and (2) for every $p_k \in T$, $\mathbb{V}'[k] = (\emptyset, \{\emptyset\})$. Observe that \mathbb{V}' is a T -resilient view for p_j . By Lemma 12, we have that $S \in \mathcal{Q}(p_j, \mathbb{V}')$. This implies that $S \cap Q_i \subseteq T$. But combined with the fact that $p_j \in L \setminus (B \cup T)$, this contradicts the consistency property of L . ◀

We will see how this theorem has a direct effect on the liveness of permissionless Byzantine reliable broadcast.

17:18 Quorum Systems in Permissionless Networks

In a Byzantine reliable broadcast, the sender process may *broadcast* a value v by invoking $r\text{-broadcast}(v)$. The broadcast primitive outputs a value v through an $r\text{-deliver}(v)$ event. Moreover, the broadcast primitive presented in this section delivers only one value per instance. Every instance has an implicit label and a fixed, well-known sender p_s .

► **Definition 43** (Permissionless Byzantine reliable broadcast). *A protocol for permissionless Byzantine reliable broadcast satisfies the following properties. For every league L and every execution tolerated by L :*

Validity: *If a correct process p_s r -broadcasts a value v , then all correct processes in L eventually r -deliver v .*

Integrity: *For any value v , every correct process r -delivers v at most once. Moreover, if the sender p_s is correct and the receiver is correct and in L , then v was previously r -broadcast by p_s .*

Consistency: *If a correct process in L r -delivers some value v and another correct process in L r -delivers some value v' , then $v = v'$.*

Totality: *If a correct process in L r -delivers some value v , then all correct processes in L eventually r -deliver some value.*

We implement this primitive in Algorithm 1, which is derived from Bracha broadcast [2] but differs in some aspects.

In principle, the protocol follows the original one, but does not use one global quorum system known to all processes. Instead, the correct processes implicitly use the same quorum function \mathcal{Q} (Definition 11), of which they initially only know their own entry in \mathcal{Q} . They discover the quorums of other processes during the execution.

Because of the permissionless nature of our model, we consider a best-effort gossip primitive to disseminate messages among processes instead of point-to-point messages.

A crucial element of Bracha's protocol is the "amplification" step, when a process receives $f + 1$ READY messages with some value v , with f the number of faulty processes in an execution, but has not sent a READY message yet. Then it also sends a READY message with v . This generalizes to receiving the same READY message with value v from a *blocking set* for p_i and is crucial for the *totality* property.

Finally, we introduce the ANY message as a message sent by a process p_i that is blocked by two sets carrying two different values v and v' . The reason for this new message lies in the consistency property of L : given an execution e with set of faulty processes A tolerated by L , the consistency property of L implies that any two quorums of any two correct processes in L have some correct process in common. Quorum intersection is then guaranteed only for correct processes in L and nothing is assured for correct processes outside L , which might gossip different values received by non-intersecting quorums. In particular, if a correct process p_i is blocked by a set containing a value v and later is blocked by a set containing a value $v' \neq v$, then p_i gossips an ANY message containing $*$. ANY messages are then ignored by correct processes in L . As we show in the Theorem 44, correct process in L cannot be blocked by sets containing different values.

► **Theorem 44.** *Algorithm 1 implements permissionless Byzantine reliable broadcast.*

Proof. Observe that all the properties assume the existence of a league L and an execution e with set of faulty processes A tolerated by L .

Let us start with the *validity* property. Since the sender p_s is correct and from the availability property of L , every correct process p_i in L eventually receives a quorum Q_i for itself of ECHO messages containing the value v sent from p_s and updates its view \mathbb{V} according to the views received from every process in Q_i .

■ **Algorithm 1** Permissionless Byzantine reliable broadcast protocol for process p_i , with sender p_s .

State

$sent-echo \leftarrow \text{FALSE}$: indicates whether p_i has gossiped ECHO
 $echos[j] \leftarrow [\perp]$: collects the received ECHO messages from other processes
 $sent-ready \leftarrow \text{FALSE}$: indicates whether p_i has gossiped READY
 $readys[j] \leftarrow [\perp]$: collects the received READY messages from other processes
 $sent-any \leftarrow \text{FALSE}$: indicates whether p_i has gossiped [ANY, *, $\mathbb{F}[i]$]
 $delivered \leftarrow \text{FALSE}$: indicates whether p_i has delivered a value
 $\mathbb{V}[j] \leftarrow$ if $i = j$ then $\mathbb{F}[i]$ else \perp : the current view of p_i

upon invocation $r\text{-broadcast}(v)$ **do**
 send message [SEND, v , $\mathbb{F}[s]$] through gossip // only sender p_s
upon receiving a gossiped message [SEND, v , (P_s, \mathcal{F}_s)] from p_s **and** $\neg sent-echo$ **do**
 $sent-echo \leftarrow \text{TRUE}$
 $\mathbb{V}[s] \leftarrow (P_s, \mathcal{F}_s)$
 send message [ECHO, v , $\mathbb{F}[i]$] through gossip
upon receiving a gossiped message [ECHO, v , (P_j, \mathcal{F}_j)] from p_j **do**
if $echos[j] = \perp$ **then**
 $\mathbb{V}[j] \leftarrow (P_j, \mathcal{F}_j)$
 $echos[j] \leftarrow v$
upon exists $v \neq \perp$ **such that** $\{p_j \in \Pi \mid echos[j] = v\} \in \mathcal{Q}(p_i, \mathbb{V})$ **and** $\neg sent-ready$ **do**
 $sent-ready \leftarrow \text{TRUE}$
 send message [READY, v , $\mathbb{F}[i]$] through gossip
upon receiving a gossiped message [READY, v , (P_j, \mathcal{F}_j)] from p_j **do**
if $readys[j] = \perp$ **then**
 $\mathbb{V}[j] \leftarrow (P_j, \mathcal{F}_j)$
 $readys[j] \leftarrow v$
upon exists $v \neq \perp$ **such that** $\{p_j \in \Pi \mid readys[j] = v\}$ blocks p_i **and** $\neg sent-ready$ **do**
 $sent-ready \leftarrow \text{TRUE}$
 send message [READY, v , $\mathbb{F}[i]$] through gossip
upon exists $v' \neq \perp$ **such that** $\{p_j \in \Pi \mid readys[j] = v'\}$ blocks p_i **and** $readys[i] = v$ **and**
 $v \neq v'$ **and** $sent-ready$ **and** $\neg sent-any$ **do**
 $sent-any \leftarrow \text{TRUE}$
 send message [ANY, *, $\mathbb{F}[i]$] through gossip
upon receiving a gossiped message [ANY, *, (P_j, \mathcal{F}_j)] from p_j **do**
 $\mathbb{V}[j] \leftarrow (P_j, \mathcal{F}_j)$
 $readys[j] \leftarrow *$
upon exists $v \neq \perp$ **such that** $\{p_j \in \Pi \mid readys[j] = v\} \in \mathcal{Q}(p_i, \mathbb{V})$ **and** $\neg delivered$ **do**
 $delivered \leftarrow \text{TRUE}$
output $r\text{-deliver}(v)$

Then, p_i gossips [READY, v , $\mathbb{F}[i]$] containing the value v and its current view $\mathbb{F}[i]$ unless $sent-ready = \text{TRUE}$. If $sent-ready = \text{TRUE}$ then p_i already gossiped [READY, v , $\mathbb{F}[i]$].

Observe that there exists a unique value v such that if a correct process in L sends a READY message, this message contains v . In fact, if a process $p_i \in L \setminus A$ sends a READY message, either it does so after receiving a quorum Q_i for itself of ECHO messages containing v or after being blocked by a set of processes that received READY messages containing v .

In the first case, if a correct process p_i in L receives a quorum Q_i for itself of ECHO messages containing v and another correct process p_j in L receives a quorum Q_j for itself of ECHO messages containing v' , by the consistency property of L , $v = v'$ and both send a READY message containing the same v .

In the second case, first observe that by Lemma 41 we know that $p_i \in L \setminus A$ cannot be inductively blocked by processes in A . Moreover, correct processes in L cannot be blocked by sets containing different values. If this was the case, then there would exist two correct processes p_i and p_j in L and two slices of p_i and p_j , respectively, in $L \setminus A$ containing two correct processes in L that received two different values v after ECHO. Again, by the consistency property of L , this is not possible. Hence, every correct process p_j in L gossips $[\text{READY}, v, \mathbb{F}[j]]$. Eventually, every correct process p_i in L receives a quorum for itself containing $[\text{READY}, v, (P_j, \mathcal{F}_j)]$ messages and r -delivers v .

The first part of the *integrity* property is ensured by the *delivered* flag. For the second part observe that, by assumption, the receiver p_i is correct and in L . This implies that the quorum for p_i used to reach a decision contains some correct processes that have gossiped ECHO containing a value v they received from p_s .

For the *totality* property, let us assume that a correct process $p_i \in L$ r -delivered some value v . If $p_i \in L \setminus A$ r -delivered some value v , then it has received READY messages containing v from a quorum Q_i for itself. From Theorem 42 we know that exists a set B such that $Q_i \setminus A \subseteq B$ and either $L \setminus A \subseteq B$ or B blocks at least a process $p_j \in L \setminus (B \cup A)$ in an A -resilient view \mathbb{V}' for p_j . In the latter case, p_j gossips a READY message containing v and B becomes $B \cup \{p_j\}$. Observe that, by assumption, if a correct process receives a gossiped message, then eventually every other correct process receives it too. Eventually, $L \setminus A$ is covered by B and this means that every correct process in L is blocked with the same value v .

Moreover, observe that given two correct processes not in L , they may become ready for different values received from non-intersecting quorums of ECHO messages. Because of this, if a correct process $p_j \notin L$ observes a blocking set B containing a value v' different from a value v that has previously gossiped in a READY message and such that $\text{sent-any} = \text{FALSE}$, process p_j gossips an ANY message containing the value $*$. Eventually every correct process p_i in L receives a quorum Q_i for itself of $[\text{READY}, v, (P_j, \mathcal{F}_j)]$ messages and it r -delivers v .

Finally, for the *consistency* property notice that by the consistency property of L , every two quorums Q_i and Q_j of any two correct processes p_i and p_j in L intersect in some correct process p_k . Process p_k could then be outside L . If $p_k \notin L$ then, as seen for the totality property, it can be blocked by sets containing different values. If this is the case then p_k gossips an ANY message. Correct processes in L then ignore the values received from p_k and wait until receiving a quorum unanimously containing the same value v . Observe that, because L tolerates A , by availability property of L every correct process in L eventually receives a quorum made by correct processes in L . The consistency property then follows. ◀

7 Related Work

A *fail-prone system* [14], also called adversary structure [10], is a well-adopted way to describe the failure assumptions in a distributed system. This is a collection of subsets of participants in the system that may fail together, and that are tolerated to fail, in a given execution. Fail-prone systems implicitly define *Byzantine quorum systems* [14] which are used to ensure consistency and availability to distributed fault-tolerant protocols in the presence of arbitrary failures. Originally, fail-prone systems have been expressed globally, shared by every participant in the system. Damgård et al. [8] introduce the notion of *asymmetric* fail-prone system in which every participant in the system subjectively selects its own fail-prone system, allowing for a more flexible model and where the guarantees of the system are derived from personal assumptions. This is the *asymmetric-trust model* [5]. Processes in this model are classified in three different types, *faulty*, *naive*, and *wise* and this characterization is done

with respect to an execution. In particular, given an execution e with faulty set F , a process p_i is faulty if it belongs to F , p_i is naive if it does not have F in its subjective fail-prone system and p_i is wise if F is contained in its subjective fail-prone system. Properties of protocols are then guaranteed for wise processes and, in some cases [5, 6], for a subset of the wise processes called *guild*.

Cachin and Tackmann [5] introduce *asymmetric Byzantine quorum systems*, a generalization of the original Byzantine quorum in the asymmetric-trust model. They show how to implement register abstraction and broadcast primitives using asymmetric Byzantine quorum systems. An asynchronous consensus protocol has subsequently been devised by Cachin and Zanolini [6]. Moreover, they extended the knowledge about the guild and about the relation between naive and wise processes in protocols with asymmetric trust.

As a basis for the *Stellar consensus protocol*, Mazières [16] introduces a new model called *federated Byzantine agreement* (FBA) in which participants may also lie. Here, every participant declares *quorum slices* – a collection of trusted sets of processes sufficient to convince the particular participant of agreement. These slices make a *quorum*, a set of participants that contains one slice for each member and sufficient to reach agreement. All quorums constitute a *federated Byzantine quorum system* (FBQS). In this model, even if the processes do not a priori choose intersecting quorums as in the classic model [14] or as in the asymmetric one [5, 8], an intersection property among quorums is later required for the analysis of the Stellar consensus protocol.

García-Pérez and Gotsman [9] study the theoretical foundations of a FBQS, build a link between FBQS and the classical Byzantine quorum systems and show the correctness of broadcast abstractions over federated quorum systems. Moreover, they investigate decentralized quorum constructions by means of FBQS. Finally, they propose the notion of subjective dissemination quorum system, where different participants may have different Byzantine quorum systems and where there is a system-wide intersection property. FBQS are a way towards an extension of quorum systems in a permissionless setting.

Losa et al. [12] introduce *personal Byzantine quorum systems* (PBQS) by removing from FBQS the requirement of a system-wide intersection among quorums. This might lead to disjoint *consensus clusters* in which safety and liveness are guaranteed in each of them, separately, in a given execution. Moreover, they abstract the Stellar Network as an instance of PBQS and use a PBQS to solve consensus.

8 Conclusions

This work introduces a new way of specifying trust assumptions among processes in a permissionless setting: processes not only make assumptions about failures, but also make assumptions about the assumptions of other processes. This leads to formally define the notions of permissionless fail-prone system and permissionless quorum system and to design protocols to solve known synchronization problems such as Byzantine reliable broadcast.

We introduce the notion of league, a set of processes for which consistency and availability properties hold for a given quorum function. Properties of our protocols are guaranteed assuming the existence of a league.

As a future work we plan to generalize known consensus protocols such as, for example, PBFT [7], to work in our permissionless model. We believe that, by assuming the existence of a league L , properties of consensus protocols can be guaranteed to every correct process in L .

References

- 1 Ignacio Amores-Sesar, Christian Cachin, and Jovana Micic. Security analysis of ripple consensus. In *OPODIS*, volume 184 of *LIPICs*, pages 10:1–10:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020.
- 2 Gabriel Bracha. Asynchronous byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, 1987.
- 3 Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.
- 4 Christian Cachin, Giuliano Losa, and Luca Zanolini. Quorum systems in permissionless network, 2022. doi:10.48550/arXiv.2211.05630.
- 5 Christian Cachin and Björn Tackmann. Asymmetric distributed trust. In *OPODIS*, volume 153 of *LIPICs*, pages 7:1–7:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019.
- 6 Christian Cachin and Luca Zanolini. Asymmetric asynchronous byzantine consensus. In *DPM/CBT@ESORICS*, volume 13140 of *Lecture Notes in Computer Science*, pages 192–207. Springer, 2021.
- 7 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.
- 8 Ivan Damgård, Yvo Desmedt, Matthias Fitzi, and Jesper Buus Nielsen. Secure protocols with asymmetric trust. In *ASIACRYPT*, volume 4833 of *Lecture Notes in Computer Science*, pages 357–375. Springer, 2007.
- 9 Álvaro García-Pérez and Alexey Gotsman. Federated byzantine quorum systems. In *OPODIS*, volume 125 of *LIPICs*, pages 17:1–17:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018.
- 10 Martin Hirt and Ueli M. Maurer. Player simulation and general adversary structures in perfect multiparty computation. *J. Cryptol.*, 13(1):31–60, 2000.
- 11 Flavio Paiva Junqueira and Keith Marzullo. Synchronous consensus for dependent process failure. In *ICDCS*, pages 274–283. IEEE Computer Society, 2003.
- 12 Giuliano Losa, Eli Gafni, and David Mazières. Stellar consensus by instantiation. In *DISC*, volume 146 of *LIPICs*, pages 27:1–27:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019.
- 13 Dahlia Malkhi, Kartik Nayak, and Ling Ren. Flexible byzantine fault tolerance. In *CCS*, pages 1041–1053. ACM, 2019.
- 14 Dahlia Malkhi and Michael K. Reiter. Byzantine quorum systems. *Distributed Comput.*, 11(4):203–213, 1998.
- 15 Dahlia Malkhi, Michael K. Reiter, and Avishai Wool. The load and availability of byzantine quorum systems. *SIAM J. Comput.*, 29(6):1889–1906, 2000.
- 16 David Mazières. The Stellar consensus protocol: A federated model for Internet-level consensus. Stellar, available online, <https://www.stellar.org/papers/stellar-consensus-protocol.pdf>, 2016.
- 17 Rafael Pass and Elaine Shi. Hybrid consensus: Efficient consensus in the permissionless model. In *DISC*, volume 91 of *LIPICs*, pages 39:1–39:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017.
- 18 Isaac C. Sheff, Robbert van Renesse, and Andrew C. Myers. Distributed protocols and heterogeneous trust: Technical report. *CoRR*, abs/1412.3136, 2014. arXiv:1412.3136.
- 19 Isaac C. Sheff, Xinwen Wang, Robbert van Renesse, and Andrew C. Myers. Heterogeneous paxos. In *OPODIS*, volume 184 of *LIPICs*, pages 5:1–5:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020.
- 20 T. K. Srikanth and Sam Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Comput.*, 2(2):80–94, 1987.

Make Every Word Count: Adaptive Byzantine Agreement with Fewer Words

Shir Cohen ✉

Technion, Haifa, Israel

Idit Keidar ✉

Technion, Haifa, Israel

Alexander Spiegelman ✉

Aptos, San Francisco, CA, USA

Abstract

Byzantine Agreement (BA) is a key component in many distributed systems. While Dolev and Reischuk have proven a long time ago that quadratic communication complexity is necessary for worst-case runs, the question of what can be done in practically common runs with fewer failures remained open. In this paper we present the first Byzantine Broadcast algorithm with $O(n(f+1))$ communication complexity in a model with resilience of $n = 2t + 1$, where $0 \leq f \leq t$ is the actual number of process failures in a run. And for BA with strong unanimity, we present the first optimal-resilience algorithm that has linear communication complexity in the failure-free case and a quadratic cost otherwise.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Byzantine Agreement, Byzantine Broadcast, Adaptive communication

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2022.18

Funding *Shir Cohen*: Supported by the Adams Fellowship Program of the Israel Academy of Sciences and Humanities.

1 Introduction

Byzantine Agreement (BA) is a key component in many distributed systems. As these systems are being used at larger scales, there is an increased need to find efficient solutions for BA. Arguably, the most important aspect of an efficient BA solution is its communication costs. That is, how much information needs to be transferred in the network to solve the BA problem. Indeed, improving the communication complexity, often measured as word complexity, was the focus of many recent works and deployed systems [1, 11, 16, 2, 13, 7].

In the BA problem, a set of n processes attempt to agree on a *decision* value despite the presence of Byzantine processes. One of the properties of a BA algorithm is a threshold t on how many Byzantine processes it can withstand. Namely, the algorithm is correct as long as up to t processes are corrupted in the course of a run. In this paper we focus on $n = 2t + 1$ and we assume a trusted setup of a public-key infrastructure (PKI) that enables us to use a threshold signature scheme [15, 4, 6].

A large and growing body of literature has investigated how to reduce the word complexity of BA algorithms. Recently, Momose and Ren [13] have presented a synchronous protocol with $O(n^2)$ words, which meets Dolev and Reischuk's long-standing lower bound [9]. Spiegelman [16] considered the more common case, where the number of actual failures, denoted by f , is smaller than t with resilience of $n = 3t + 1$. In this paper we consider better resilience and ask:

Can we design a BA protocol with $O(n(f+1))$ communication complexity in runs with $f \leq t$ failures, where $n = 2t + 1$?



© Shir Cohen, Idit Keidar, and Alexander Spiegelman;
licensed under Creative Commons License CC-BY 4.0

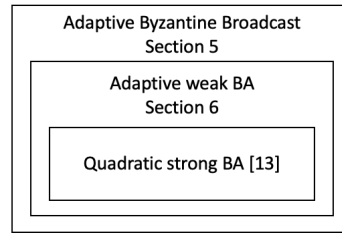
26th International Conference on Principles of Distributed Systems (OPODIS 2022).

Editors: Eshcar Hillel, Roberto Palmieri, and Etienne Rivière; Article No. 18; pp. 18:1–18:21



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



■ **Figure 1** Relation between various Byzantine Agreement solutions. Each box uses the primitives within it.

Whereas Dolev and Reischuk’s better-known lower bound applies to worst-case runs, they further proved a lower bound of $\Omega(nt)$ signatures in failure-free runs ($f = 0$) in a model with a PKI. At the time, one could have thought that this bound extends to the communication complexity, rendering it $\Omega(nt)$ even with small f values. However, the introduction of threshold signature schemes [8, 15, 4, 6] exposed the possibility to compact many signatures into one word, potentially saving many words.

In this paper, we first revisit the original problem as stated in Dolev and Reischuk’s work. In this problem there is a single sender who proposes a value and we refer to this problem as Byzantine Broadcast (BB). We prove that although $O(nt)$ signatures are inevitable, $O(nt)$ messages are not necessary with $f \in o(t)$ failures by presenting an adaptive BB solution with $O(n(f + 1))$ words.

The idea behind our algorithm is to reduce this problem to another BA variant. There is a simple reduction from BB to BA with the strong unanimity validity property (from hereon: *strong BA*), which states that if all correct processes propose the same value, this is the only allowed decision. In this reduction, the sender initially sends its value to all other processes who then run a BA solution. Unfortunately, no *adaptive strong BA* is known to date. I.e., a strong BA solution where communication complexity depends on f , rather than on t . Instead, in Section 5 we reduce the problem to a new *weak BA* problem with a weaker validity property, *unique validity*, which we define in this paper.

Intuitively, the validity condition of weak BA is somewhere between weak unanimity, where if all processes are correct and propose the same value this is the only allowed decision, and external validity [5], where a decision value must satisfy some external predicate. In weak BA, one can define its desired predicate and the requirement is that if all correct processes propose the same value and Byzantine processes cannot devise a value that satisfies the chosen predicate, then the decision must be valid. Otherwise, \perp is allowed.

While the unique validity condition seems to be weak, it is surprisingly powerful when provided the “right” external predicate. For example, we can determine that a value is valid if it has at least $t + 1$ unique signatures, assuring that some correct process in the system knows this value. Unique validity may be of independent interest as a tool for designing algorithms. We present our adaptive weak BA in Section 6. The weak BA, in turn, exploits the quadratic solution by Momose and Ren [13]. Figure 1 describes the relation between the various solutions.

Finally, we consider strong BA. In Section 7, we present the first optimally resilient strong binary BA protocol with $O(n)$ communication complexity in the failure-free case. This leaves open the question whether a fully adaptive (to any f) strong BA protocol exists. We summarize the results in Table 1.

■ **Table 1** Bounds on communication complexity of deterministic synchronous Byzantine Agreement algorithms with resilience $n = 2t + 1$.

	Upper Bound	Lower Bound
Byzantine Broadcast	$O(n(f + 1))$ Section 5 + Section 6	$\Omega(nf)$ ($\Omega(n^2)$ signatures) [9]
Strong BA	$O(n^2)$ multi-valued Momose-Ren [13] $O(n)$ with $f = 0$, binary Section 7	$\Omega(nf)$ binary ($\Omega(n^2)$ signatures) [9]
Weak BA	$O(n(f + 1))$ multi-valued Section 6	$\Omega(n)$

2 Model and Preliminaries

We consider a distributed system consisting of a well-known static set Π of n processes and an adaptive adversary. The adversary may adaptively corrupt up to $t < n$, $n = 2t + 1$ processes in the course of a run. A corrupted process is *Byzantine*; it may deviate arbitrarily from the protocol. In particular, it may crash, fail to send or receive messages, and send arbitrary messages. As long as a process is not corrupted by the adversary, it is *correct* and follows the protocol. We denote by $0 \leq f \leq t$ the actual number of corrupted processes in a run.

Cryptographic tools. We assume a trusted public-key infrastructure (PKI) and a computationally bounded adversary. Hence, we can construct and use a threshold signature scheme [15, 4, 6]. We denote by $\langle m \rangle_p$ the message m signed by process p . Using a (k, n) -threshold signature scheme, k unique signatures on the same message m can be batched into a threshold signature for m with the same length as an individual signature. For simplicity we abstract away the details of cryptography and assume the threshold signature schemes are ideal. In practice, our results hold except with arbitrarily small probability, depending on the security parameters.

Communication. Every pair of processes is connected via a reliable link. If a correct process p_i receives a message m indicating that m was sent by a correct process p_j , then m was indeed generated by p_j and sent to p_i . The network is synchronous. Namely, there is a known bound δ on message delays, allowing us to design protocols that proceed in rounds. Specifically, if a correct process sends a message to any other correct process at the beginning of some round, it is received by the end of the same round.

Complexity. We use the following standard complexity notions [2, 16, 13]. While measuring complexity, we say that a *word* contains a constant number of signatures and values from a finite domain, and each message contains at least 1 word. The communication complexity of a protocol is the maximum number of words sent by all correct processes, across all runs. The *adaptive* complexity is a complexity that depends on f .

3 Problem Definitions

We consider a family of agreement problems all satisfy agreement and termination defined as follows:

Agreement No two correct processes decide differently.

Termination Every correct process eventually decides.

In addition, each variant of the problem satisfies some validity property. In the Byzantine Broadcast (BB) problem, a designated sender has an input to broadcast to all n processes. The goal is that all correct processes decide upon the sender's value. If the sender is Byzantine, however, it is enough that all correct processes decide upon some common value. Formally,

► **Definition 1** (Byzantine Broadcast). *In Byzantine Broadcast, a designated sender has an input value v_{sender} to broadcast to all processes, and each correct process decides on an output value decision_i . BB solution must satisfy agreement, termination and the following validity property:*

Validity *If sender is correct, then all correct processes decide v_{sender} .*

Byzantine Agreement (BA) is a closely related problem to BB. In this problem, a set Π of n processes each propose an initial value and they all attempt to reach a common decision. In addition, the decided value must be “valid” in some sense that makes the problem non-trivial. The classic notion of validity states that if all correct processes in Π share the same initial value, then the decision must be on this value. This property is known as *strong unanimity*, and it entails a limitation on the resilience of a protocol, requiring that $n \geq 2t + 1$. For hereon we refer to BA with strong unanimity validity condition as *strong BA*. Formally,

► **Definition 2** (Strong Byzantine Agreement). *In Byzantine Agreement, each correct process $p_i \in \Pi$ proposes an input value v_i and decides on an output value decision_i . Any strong BA solution must satisfy agreement, termination and the following validity property:*

Strong unanimity *If all correct processes propose the same value v , then the output is v .*

A different validity property requires that a decision satisfies some external boolean predicate (we call such value a *valid* value). It is used under the assumption that all correct processes propose valid values. This is known as *external validity* [5] and only requires $n > t$. External validity by itself is trivial in case there is a well-known predefined value that satisfies the predicate. However, it is commonly used in settings with signatures, where valid values can be verified by all but generated only by specific users or sets thereof. For instance, consider a predicate that verifies that v is signed by $n - t$ processes – no process can unilaterally generate a default valid value.

Our notion of unique validity adopts external validity to allow default values to be decided in cases when there is no unanimous valid value. We say that a value v *exists* in a run of a BA protocol if v is either the input value of a correct process or can be generated by a Byzantine process. E.g., any value signed by a non-Byzantine process cannot be generated locally by a Byzantine process. Unique validity stipulates that there is a default value if and only if there exists more than one valid value in a BA run. Formally,

► **Definition 3** (Weak Byzantine Agreement). *In weak Byzantine Agreement, each correct process $p_i \in \Pi$ proposes an input value v_i and decides on an output value decision_i . Any weak BA solution must satisfy agreement, termination and the following validity property:*

Unique Validity *Assume an arbitrary predicate $\text{validate}(v) \in \{\text{true}, \text{false}\}$ that can be computed locally. If a correct process decides v then either $v = \perp$ or $\text{validate}(v) = \text{true}$, and if $v = \perp$ then more than one valid value exists in the run.*

As the definition suggests, unique validity is satisfied in weak BA with respect to any chosen external predicate. This allows for the application level to determine the desired properties, and choose the relevant external predicate accordingly. As a simple example, one can think of a predicate that specifies that a value is valid if it is signed by at least $t + 1$ processes stating that this value was their initial value. In this scenario, unique validity yields exactly the common strong unanimity property on the underlying signed values.

In fact, unique validity is a useful tool when designing distributed algorithms as it allows to use BA as a framework. Different applications may require different validity conditions, yet still unique validity prevents the system from having a trivial solution in the presence of Byzantine processes. Note, in addition, that every solution to BA with external validity property immediately solves weak BA.

4 Related Work

The starting point of this work goes back to 1985 when Dolev and Reischuk proved two significant lower bounds for the Byzantine Broadcast problem. Specifically, they have studied the worst-case message complexity over all runs and proved it to be $\Omega(nt)$. Moreover, in the authenticated model, which was somewhat undeveloped at the time, they proved a lower bound of $\Omega(nt)$ signatures – even in a failure-free run.

Since the publication of their fundamental results, the paradigm of complexity measurement has shifted. The number of messages is of little importance nowadays, compared to the number of words it entails. The total number of words (the communication complexity) better reflects the load on the system and is commonly used today when analyzing distributed algorithms. For example, Dolev and Reischuk presented in their paper a BB algorithm that matches their messages' lower bound. It requires $O(nt)$ messages, but as a single message can be composed of many different signatures it requires a cubic number of words. It was not until recently that a solution with quadratic communication complexity was presented for synchronous BA with optimal resilience [13].

Dolev and Reischuk's complementary lower bound on signatures does not translate to a bound on the communication complexity of an algorithm. Only a few years after Dolev and Reischuk's work, the threshold signature scheme was introduced [8]. This scheme allows multiple signatures to be compacted into a single combined signature of the same size. That is, a single word can carry multiple signatures. In this work, we focus on the communication complexity of the BB and BA problems while taking advantage of such schemes.

To make our algorithms efficient in real-world systems, we adjust the complexity to match the actual number of faults. Moreover, we do so without compromising the worst-case complexity. If all t possibly Byzantine processes crash, the complexity of our algorithms is $O(nt)$. However, in most runs, where systems do not exhibit the worst crash patterns, the complexity is much lower. In fact, it is linear in the number of faults times n .

While consensus algorithms were designed to be adaptive in the number of failures over 30 years ago [10], these works focus on the number of rounds that it takes to reach a decision rather than on communication complexity. A special case of adaptivity is focusing on failure-free runs. This problem was addressed both by Amdur et al. [3] (only for crash failures) and by Hadzilacos and Halpern [12]. However, both works measure the number of messages rather than words and have sub-optimal communication complexity.

A recent work by Spiegelman [16] tackled the problem of adaptive communication complexity in the asynchronous model. It presents a protocol that achieves correctness in asynchronous runs and requires $O(ft + t)$ communication in synchronous runs. However, due to the need to tolerate asynchrony, its resilience is only $n \geq 3t + 1$. This solution relies on threshold signatures schemes, as we do.

As noted also by Momose and Ren [13], designing optimally-resilient protocols for the synchronous model limits the use of threshold signatures. While this primitive has been used in various eventually synchronous and asynchronous works over the last few years [1, 16, 14, 6], usually with a threshold of $n - t$. Using this threshold in settings with resilience $n = 3t + 1$, we get certificates signed by at least $t + 1$ correct processes. However, for a resilience of $n = 2t + 1$, this is no longer the case. The threshold signatures “lose” their power as $n - t = t + 1$ for which no intersection properties between correct processes signing two distinct certificates can be derived. In this work, we exploit threshold signatures with this improved resilience by carefully choosing a better threshold for our needs, as we discuss in Section 6. We mention that although not using threshold schemes, Xiang et al. [17] also benefit from collecting more than $n - t$ signatures in some scenarios.

5 From Weak BA to Adaptive Byzantine Broadcast

In this section we study the BB problem, and optimize its adaptive communication complexity over all runs. We present a new BB protocol with resilience $n = 2t + 1$ and adaptive communication complexity of $O(n(f + 1))$.

Recall that in the BB problem there is only one sender who aims to broadcast its initial value and have all correct processes agree on it. If the sender is Byzantine, it may attempt to cause disagreement across correct processes. There is a known simple and efficient reduction from BB to strong BA. Given a strong BA solution, the designated sender starts by sending its value to all processes, and then they all execute the BA solution and decide on its output. It is easy to see that if the sender is correct, all correct processes begin the strong BA algorithm with the same input, and by strong unanimity they then decide upon the sender's value.

However, trying to apply the same reduction from BB to weak BA no longer works. If the sender is Byzantine, the correct processes do not have a valid initial value for the BA. Nonetheless, in this section we present a reduction from BB to weak BA¹, which incurs a cost of $O(n(f + 1))$ words. Thus, together with an adaptive weak BA with the same complexity, we obtain a synchronous adaptive BB algorithm with a total of $O(n(f + 1))$ words and resilience $n = 2t + 1$. At this point we assume that such adaptive weak BA is given as a black box. An implementation for this primitive is presented in Section 6.

■ **Algorithm 1** BB algorithm: code for process p_i , sender's input is v_{sender} .

Initially $v_i, val, decision, ba_decision = \perp$

Round 1:

```

1: if  $sender = p_i$  then
2:   send  $\langle v_{sender} \rangle_{sender}$  to all
3: if received message  $\langle v \rangle_{sender}$  from  $sender$  then
4:    $v_i \leftarrow \langle v \rangle_{sender}$ 
5: for  $j = 1$  to  $n$  do
6:    $val \leftarrow invokePhase(j, v_i)$ 
7:   if  $val \neq \perp$  then
8:      $v_i \leftarrow val$ 
9:  $ba\_decision \leftarrow$  weak BA with  $BB\_valid$  predicate and initial value  $v_i$ 
10: if  $ba\_decision$  is of the form  $\langle v \rangle_{sender}$  then
11:    $decision \leftarrow v$ 
12: else
13:    $decision \leftarrow \perp$ 

```

Our algorithm, presented in Algorithms 1 and 2, is composed of three parts. The first part (lines 1 – 4 in Algorithm 1) is the first round in which the leader disseminates its value. Processes that receive that value adopt it as their BA initial value (line 4). The second part (lines 5 – 8 in Algorithm 1 and Algorithm 2) is a “vetting” part. It consists of n phases, with a rotating leader. Leaders initiate phases to learn about the first part's initial value. Finally, the third part (lines 9 – 13 in Algorithm 1) is a weak BA execution.

¹ This reduction only works if $n \geq 2t + 1$.

Deciding upon the weak BA output takes care of the agreement and termination properties. It is left to (1) satisfy the BB validity property and (2) make sure that the preconditions for the weak BA hold, that is, each correct process has a valid input to propose. To achieve these properties, we define the $BB_valid(v)$ predicate in the following way. $BB_valid(v) = true$ if and only if v is signed by either the sender or by $t + 1$ processes.

Note that if the sender happens to be Byzantine, it is acceptable to decide on any value. However, it is important to make sure that if the sender is correct, then the only valid value is its initial BB input. Simply setting a value to be valid only if it is signed by the sender would not work, as it allows a faulty sender to cause a scenario in which there are no valid values to agree upon by not sending its value to any process. Note that we cannot simply fix this by introducing some default valid value: If we were to do so, it would be valid to agree on that value also in the case of a correct sender, violating the BB validity condition.

Our algorithm makes sure that if the sender is correct, the second condition in the BB_valid definition cannot be satisfied, and hence there is only one possible outcome to the BA algorithm. However, if the sender is Byzantine, it is guaranteed that there is some value to decide upon. That is, all correct processes start the weak BA with an initial value that satisfies the predicate.

In the vetting part of the algorithm, we ensure that the above-mentioned conditions hold. Moreover, we do so with a communication complexity that is adaptive to the number of actual process failures. The core idea is to work in leader-based *phases*. Every phase has a unique leader and is composed of a constant number of leader-to-all and all-to-leader synchronous rounds. Every phase is initiated by a leader-to-all message. If the leader decides not to send the initial message then no messages will be sent by correct processes in this phase and we say that this phase is *silent*, and otherwise, it is *non-silent*. In our algorithm, a phase is non-silent if the phase's leader did not choose an initial value for the BA prior to that phase.

In every phase, each process p_i starts the phase with some initial value v_i and if the phase is non-silent it returns some value. The requirements from the phase are: (1) If the phase's leader is correct and the phase is non-silent, then all correct processes return a valid value. (2) All correct processes return either \perp or a valid v . And (3) if the sender is correct, then no correct process returns a value signed by $t + 1$ processes.

Upon a non-silent phase, the leader starts by asking all processes for help by sending a `help_req` message (line 16). A correct process that receives a help request message answers the leader. If it has set a BA initial value, it sends it to the leader at line 19, and otherwise, it sends a signed `idk` (i don't know) message at line 21. If the leader receives a value signed by the designated sender it broadcasts it (line 24). Otherwise, if it receives $t + 1$ `idk` messages, it uses a threshold signature scheme to create an `idk` quorum certificate and broadcasts it (line 27). A process that receives from the leader a value signed by either the sender or any $t + 1$ processes returns it. Otherwise, it returns \perp .

At the end of each non-silent phase, a correct process that returns a $v \neq \perp$ from the phase, updates its local v_i accordingly at line 8. This value at the end of the n^{th} phase is the input for the weak BA algorithm. Since we execute n phases, all correct processes set valid values by the end of all phases. This is because once there is a correct process that did not set a value it initiates its phase and then all correct processes return with a valid value. At this point, all processes execute the weak BA and decide upon its output (line 9).

A formal correctness proof of Algorithms 1 and 2 appears in Appendix A, proving the following theorem:

► **Theorem 4.** *Algorithm 1 solves BB.*

■ **Algorithm 2** $invokePhase(j, v_i)$: code for process p_i .

```

14:  $leader \leftarrow p_{j \bmod n}$ 
    Round 1:
15: if  $leader = p_i$  and  $v_i = \perp$  then
16:   broadcast the message  $\langle \text{help\_req}, j \rangle_{leader}$ 
    Round 2:
17: if received  $\langle \text{help\_req}, j \rangle_{leader}$  then
18:   if  $v_i \neq \perp$  then
19:     send  $\langle v_i, j \rangle$  to  $leader$ 
20:   else
21:     send  $\langle \text{idk}, j \rangle_{p_i}$  to  $leader$ 
    Round 3:
22: if  $leader = p_i$  then
23:   if received  $\langle v', j \rangle$  s.t.  $v' = \langle v \rangle_{sender}$  then
24:     broadcast the message  $\langle \langle v \rangle_{sender}, j \rangle$ 
25:   else if received  $t + 1$  unique signatures  $\langle \text{idk}, j \rangle_{p'}$  then
26:     batch these messages into  $QC_{\text{idk}}$  using a  $(t + 1, n)$ -threshold signature scheme
27:     broadcast the message  $\langle QC_{\text{idk}}, j \rangle$ 
28: if received  $\langle v, j \rangle$  from  $leader$  and  $BB\_valid(v) = true$  then
29:   return  $v$ 
30: else
31:   return  $\perp$ 

```

Complexity

We prove that the complexity of Algorithms 1 and 2 is $O(n(f + 1))$.

Each non-silent phase is composed of a constant number of all-to-leader and leader-to-all rounds and thanks to the use of threshold signatures, all messages sent have a size of one word. Thus, each phase incurs $O(n)$ words. In total, there are potentially n phases. However, we prove in Appendix A that after the first non-silent phase by a correct leader, all following phases with correct leaders are silent. Thus, the number of non-silent phases is linear in f . We conclude that all phases in lines 5 – 8 use $O(n(f + 1))$ words. The complexity of the weak BA black box is also $O(n(f + 1))$ (as we will show in the next section), resulting in a total of $O(n(f + 1))$ words.

6 Adaptive Weak BA

In this section, we present a synchronous adaptive weak BA algorithm with resilience $n = 2t + 1$. This algorithm is the missing link for the adaptive BB presented in the previous section. Once again, we use the concept of phases and exploit the pattern of possible *silent* phases. In this algorithm, the phases are slightly different and the decision to start a phase as a leader depends on whether or not the leader has reached a decision in previous phases.

Unlike the BB problem, in BA every process begins the algorithm with its own input value. Communication-efficient solutions to this problem usually employ threshold signatures schemes [1, 16]. This technique is widely used in asynchronous and eventually synchronous protocols, with resilience $n = 3t + 1$. In these contexts, one can use a scheme of $(n - t)$ -out-of- n signatures and benefit from the fact that any two such quorum certificates intersect by at least $t + 1$ processes, and therefore at least one correct process.

Unfortunately, when trying to apply the same technique to a system with resilience $n = 2t + 1$, it fails. A correct process might be unable to obtain $2t + 1$ unique signatures on any value as Byzantine processes might not sign it. On the other hand, a quorum certificate with only $t + 1$ unique signatures is not very useful as it does not guarantee the desired intersection property.

Our first key observation is that the intersection property can be achieved as long as we have $\lceil \frac{n+t+1}{2} \rceil$ unique signatures. If we obtain this number of signatures out of $n = 2t + 1$, safety is preserved in the sense that conflicting certificates cannot be formed by a malicious adversary. Of course, there are runs in which we cannot reach that threshold since $\lceil \frac{n+t+1}{2} \rceil > n - t$ (e.g., if t processes crash immediately as the run begins). But in this case, $f \geq \frac{t}{2}$, and $O(f)$ becomes asymptotically $O(t)$. Hence, we can use a fallback algorithm with $O(nt)$ communication complexity.

As we assume that $t \in \Theta(n)$, we can use Momose and Ren's synchronous algorithm that has $O(n^2)$ communication complexity [13] for the fallback. We denote that algorithm $\mathcal{A}_{fallback}$. Note that their algorithm is "stronger" than our proposed algorithm as it provides strong unanimity for validity (i.e., it solves strong BA). We can use their solution by checking the validity of $\mathcal{A}_{fallback}$'s output according to the predicate. If it is valid, this is the decision value, and otherwise a default valid value is decided. Equipped with these insights, we next present our algorithm.

During the phases part of the protocol, a correct process must commit a value before reaching a decision. When it has certainty about a value it updates that value in a *commit* variable, along a *commit_proof* of this commitment (a quorum certificate, signed by sufficiently many processes). Once a correct process commits to a certain value it does not commit to any other value during the run. It may, however, decide on another value eventually. For example, if it reaches the fallback and no correct process has decided. Once a correct process reaches a decision it updates it in its local *decision* variable as well as a matching quorum certificate in *decide_proof* variable.

A single phase. The code for a single phase is given in Algorithm 4. Each process p_i starts a phase with its initial value v_i and information about possible previous commits (*commit*, *commit_proof*) and decisions (*decision*, *decide_proof*). Correct processes return with updated information about commits and decisions that were made in that phase (or prior to that). The guarantees of the phases are: (1) Every *decision* updated during a phase is valid; (2) All decisions updated by correct processes are the same and there exists at most one valid *decide_proof* in the system; and (3) If the phase's leader is correct, the phase is non-silent, and $n - f > \lceil \frac{n+t+1}{2} \rceil$, then all correct processes return with the same valid decision.

Every non-silent phase starts with the leader broadcasting a **propose** message with its value in line 32. Upon receiving this message, correct processes either vote for this value by signing it (line 34) or answer with a value that was previously committed as well as its commit quorum certificate (line 36). If the leader receives a committed value it simply broadcasts it. Otherwise, if it manages to achieve the required $\lceil \frac{n+t+1}{2} \rceil$ threshold of signatures, it can form a quorum certificate committing its proposed value (line 40).

Note that at this point the committed value is not "safe enough" to be decided by correct processes. Byzantine leaders may cause correct processes to participate in forming a commit certificate for more than one value. As correct processes that have decided do not initiate phases, they might never communicate without going through Byzantine leaders. Thus,

18:10 Make Every Word Count: Adaptive Byzantine Agreement with Fewer Words

■ **Algorithm 3** weak BA algorithm: code for process p_i with initial value v_i .

Initially $decision = undecided, bu_decision = v_i, fallback_start \leftarrow \infty$
 $decide_proof, commit, commit_proof, bu_proof, fallback_val, phase_decision = \perp$

- 1: **for** $j = 1$ to $t + 1$ **do**
- 2: $phase_decision, decide_proof, commit, commit_proof \leftarrow$
 $invokePhase(j, v_i, decision, commit, commit_proof)$
- 3: **if** $decision = undecided$ and $phase_decision \neq undecided$ **then**
- 4: $decision \leftarrow phase_decision$
- Round 1:**
- 5: **if** $decision = undecided$ **then**
- 6: broadcast $\langle help_req \rangle_{p_i}$
- Round 2:**
- 7: **if** received $\langle help_req \rangle_{p'}$ message and $decision \neq undecided$ **then**
- 8: send $\langle help, decision, decide_proof \rangle_{p_i}$ to p'
- 9: **if** received $t + 1$ messages of $\langle help_req \rangle_{p'}$ from different processes **then**
- 10: batch these messages into $QC_{fallback}(v)$ using a $(t + 1, n)$ -threshold signature scheme
- 11: broadcast the message $\langle fallback, QC_{fallback}, decision, proof \rangle_{p_i}$
- 12: $fallback_start \leftarrow now + 2\delta$
- Round 3:**
- 13: **if** received $\langle help, v, decide_proof \rangle_{p'}$ with valid v and $decide_proof$ for v and $decision = undecided$ **then**
- 14: $decision \leftarrow v$
- 15: $bu_decision \leftarrow decision$
- 16: **while** $fallback_start > now$ **do**
- 17: **if** received valid $\langle fallback, QC_{fallback}, v, proof_{p'} \rangle_{p'}$ **then**
- 18: **if** $decision = undecided$ and $proof_{p'} \neq \perp$ is a valid proof for a valid v **then**
- 19: $bu_decision \leftarrow v$
- 20: $bu_proof \leftarrow proof_{p'}$
- 21: **if** $fallback_start = \infty$ **then**
- 22: broadcast the message $\langle fallback, QC_{fallback}, bu_decision, bu_proof \rangle_{p_i}$
- 23: $fallback_start \leftarrow now + 2\delta$
- 24: $fallback_val \leftarrow A_{fallback}$ with $\delta' = 2\delta$ and initial value $bu_decision$
- 25: **if** $decision = undecided$ **then**
- 26: **if** $fallback_val$ is valid **then**
- 27: $decision \leftarrow fallback_val$
- 28: **else**
- 29: $decision \leftarrow \perp$

we need another level of certainty, in the form of the finalize certificate (to be stored in $decide_proof$). We maintain the invariant that if a correct process receives a valid finalize certificate, then no finalize certificate on another value can be formed.

Thus, after a correct process learns about a committed certificate it sends a matching **decide** message to the leader (line 44). In addition, if this is the first commit certificate it receives, it commits to it. If the leader receives the necessary threshold of **decide** messages, it forms a finalize quorum certificate. Every process that receives such a certificate can safely return the certificate's value as its decision.

■ **Algorithm 4** *invokePhase*($j, v_i, decision, decide_proof, commit, commit_proof$): code for process p_i .

```

30:  $leader \leftarrow p_{j \bmod n}$ 
    Round 1:
31: if  $leader = p_i$  and  $decision = \perp$  then
32:   broadcast the message  $\langle \text{propose}, v_i, j \rangle_{leader}$ 
    Round 2:
33: if received  $\langle \text{propose}, v, j \rangle_{leader}$  with a valid  $v$  for the first time and  $commit = \perp$  then
34:   send  $\langle \text{vote}, v, j \rangle_{p_i}$  to  $leader$ 
35: else if received  $\langle \text{propose}, v, j \rangle_{leader}$  and  $commit \neq \perp$  then
36:   send  $\langle \text{commit}, commit, commit\_proof, j \rangle_{p_i}$  to  $leader$ 
    Round 3:
37: if  $leader = p_i$  then
38:   if received  $\langle \text{commit}, w, QC_{\text{commit}}(w), j \rangle'_p$  then
39:     broadcast the message  $\langle \text{commit}, w, QC_{\text{commit}}(w), j \rangle_{leader}$ 
40:   else if received  $\lceil \frac{n+t+1}{2} \rceil$  messages of  $\langle \text{vote}, v, j \rangle_{p'}$  then
41:     batch these messages into  $QC_{\text{commit}}(v)$  using a  $(\lceil \frac{n+t+1}{2} \rceil, n)$ -threshold signature
    scheme
42:     broadcast the message  $\langle \text{commit}, v, QC_{\text{commit}}(v), j \rangle_{leader}$ 
    Round 4:
43: if received  $\langle \text{commit}, v, QC_{\text{commit}}(v), j \rangle_{leader}$  then
44:   send  $\langle \text{decide}, v, j \rangle_{p_i}$  to  $leader$ 
45:   if  $commit = \perp$  then
46:      $commit \leftarrow v$ 
47:      $commit\_proof \leftarrow QC_{\text{commit}}(v)$ 
    Round 5:
48: if  $leader = p_i$  then
49:   if received  $\lceil \frac{n+t+1}{2} \rceil$  messages of  $\langle \text{decide}, v, j \rangle_{p'}$  then
50:     batch these messages into  $QC_{\text{finalized}}(v)$  using a  $(\lceil \frac{n+t+1}{2} \rceil, n)$ -threshold signature
    scheme
51:     broadcast the message  $\langle \text{finalized}, v, QC_{\text{finalized}}(v), j \rangle_{leader}$ 
52: if received  $\langle \text{finalized}, v, QC_{\text{finalized}}(v), j \rangle_{leader}$  then
53:    $decision \leftarrow v$ 
54:    $decide\_proof \leftarrow QC_{\text{finalized}}(v)$ 
55: return  $(decision, decide\_proof, commit, commit\_proof)$ 

```

Main algorithm. The BA algorithm is given in Algorithm 3, using the phase algorithm as a building block. In our algorithm, all correct processes eventually decide by updating their *decision* variable. However, they do not halt. In our BA algorithm, we start by executing n phases with a rotating leader, ensuring that every correct process has a chance to reach a decision before executing the fallback algorithm. After the phases end there are several possibilities. First, if there are at most $\frac{n-t-1}{2}$ Byzantine processes, all correct processes must have decided. If there are more Byzantine processes, it may be the case that some correct processes decided and others did not. This could happen, for example, if a Byzantine leader causes the single correct leader to decide and not initiate its phase. By the phase guarantees, we know that all correct processes that decide by this point, decide the same valid value.

18:12 Make Every Word Count: Adaptive Byzantine Agreement with Fewer Words

To address the case where not all correct processes decided, we have processes that have not decided ask for help from all other processes (line 6). If a correct process has decided and receives a `help_req` message, it answers with a help message including the decision value along with its proof at line 8. Note that in this round, the number of messages sent by correct processes is linear in the number of help requests. Specifically, if only Byzantine processes send `help_req` messages, the number of answers is $O(nf)$ and independent of t .

We note that if $t + 1$ help requests are sent, then at least one of them is sent by a correct process that did not manage to form quorum certificates when it served as leader. Thus, in this case, $f \in \Theta(t)$, and we can execute the fallback algorithm. To make sure that all correct processes participate in the fallback algorithm, a `fallback` certificate with $t + 1$ signature is formed.

We now encounter a new challenge. We must have all correct processes start a synchronous fallback algorithm at the same time. However, an adversary can form the `fallback` certificate and deal it to only some correct processes. This scenario can happen, for example, if less than $t + 1$ `help_req` messages are sent, and the adversary adds t `help_req` signatures of its own. We thus require a correct process that receives a `fallback` certificate to broadcast it (line 22). This ensures that whenever one correct process runs the fallback algorithm, all of them do, but may still cause different correct processes to start the fallback at different times. Nevertheless, we know that the starting time difference is at most the δ it takes the message to arrive. We therefore run the fallback algorithm with $\delta' = 2\delta$, ensuring that all correct processes enter a fallback round before any of them exits from it.

Another subtle point is making sure that the fallback algorithm does not output a decision value that contradicts previous decisions made by correct processes. For that reason we add another 2δ safety window between getting notified about a fallback and initiating it. Correct processes that broadcast the `fallback` certificate attach their decision value and a proof (if exists). In the 2δ safety window, processes that learn about a decision value in the system adopt it as the initial value for the fallback algorithm (line 17). Recall that $\mathcal{A}_{fallback}$ is a strong BA protocol. If a correct process decides v prior to the fallback algorithm, all other correct processes learn about v during the safety window. Then, by strong unanimity, they all decide v .

Note that if the decision returned from $\mathcal{A}_{fallback}$ is not valid then it must be that strong unanimity preconditions are not satisfied (since correct processes always have valid inputs) and a default value is returned. Furthermore, whenever the strong unanimity precondition is not satisfied, it follows that not all correct processes propose the same value. As a result, there must exist more than one valid value in the run (the different correct proposals). And the \perp default value is a valid weak BA output.

A formal correctness proof of Algorithms 3 and 4 appears in Appendix B, proving the following theorem:

► **Theorem 5.** *Algorithm 3 solves weak BA.*

Complexity

We show that if $f < \frac{n-t-1}{2}$, correct processes never perform the fallback algorithm.

► **Lemma 6.** *If $f < \frac{n-t-1}{2}$, correct processes never perform the fallback algorithm.*

Proof. In Appendix B we prove that if a correct process is the leader of a non-silent phase and $f < \frac{n-t-1}{2}$, then all correct processes return the same valid decision. Since Algorithm 3 is composed of n phases, every correct process has a chance to invoke its phase and all correct processes decide by line 4. Assume by way of contradiction that there exists a correct

process that invokes the fallback algorithm. By the code, it has received a `fallback` certificate. However, such certificate can only be formed by $t + 1$ unique `help_req` signatures, meaning that at least one correct process sent a `help_req` message. But this is impossible if all correct processes decide by line 4. ◀

Each phase is composed of a constant number of all-to-leader and leader-to-all rounds. Thus, it incurs $O(n)$ words. Potentially, there are n phases. However, a lemma in Appendix B proves that once a correct leader invokes `invokePhase()` and the number of actual failures is $f < \frac{n-t-1}{2}$, all correct processes decide by the end of that phase. Since correct leaders that had already decided do not invoke their phases (their phases are silent), the number of invoked phases depends on f itself. Thus, all phases combined send $O(n(f + 1))$ words.

After n `invokePhase` invocations end, help request messages are sent only by correct processes that did not decide. By the above-mentioned lemma, it happens only if $f > \frac{n-t-1}{2}$. In this case, $f = \Theta(n)$ and since $t = \Theta(n)$ it holds that $O(nf) = O(n^2)$. Correct processes that decide by this point answer directly to whoever sent them help requests, without affecting the asymptotic complexity. If some correct process receives a `fallback` certificate, another all-to-all round is added, keeping the complexity $O(n^2)$. All other communication costs are incurred in the fallback algorithm, whose complexity is also $O(n^2)$.

7 strong BA: the failure free case

Recall that the optimal resilience for strong BA is $n = 2t + 1$. In this section we present a binary strong BA protocol that has communication complexity of $O(n)$ in the failure free case. Otherwise, it has complexity $O(n^2)$. The question of whether an adaptive protocol with $O(n(f + 1))$ complexity can be designed for strong BA with optimal resilience remains open.

In the algorithm, presented in Algorithm 5, a single leader first collects all initial values. Since we solve binary agreement, in the failure-free case there must be a value proposed by $t + 1$ different processes. Thus, the leader can use a threshold signature scheme to aggregate a quorum certificate on this proposed value.

As a second step, the leader sends this certificate to all processes and attempts to collect n different signatures on the value. If it succeeds, it broadcasts it. Every process that receives a signed-by-all certificate can safely decide upon its value. If a correct process does not decide, it broadcasts a fallback message. Every process that hears such a message, echoes it at most once, and execute $\mathcal{A}_{fallback}$ after 2δ time with 2δ -long rounds, as in Section 6. In Appendix C we prove the correctness of Algorithm 5. That is, we prove the following theorem:

▶ **Theorem 7.** *Algorithm 5 solves binary strong BA.*

Complexity

We show that if the run is failure-free, correct processes never perform the fallback algorithm.

▶ **Lemma 8.** *If $f = 0$, correct processes never perform the fallback algorithm.*

Proof. If all processes are correct then they all send their initial values to the leader at line 2. Since values are binary, and there are $n = 2t + 1$ processes, there must be a value v such that the leader receives $t + 1$ unique signatures on v . Then, the leader broadcasts a `propose` certificate on v (line 6). Every correct process that receives this certificate replies with a signed `decide` message at line 8. Since all processes are correct, the leader then receives n signatures and then broadcasts a `decide` certificate on v (line 12). All processes then receive this certificate and decide v at line 14. None of them sends a fallback message. ◀

18:14 Make Every Word Count: Adaptive Byzantine Agreement with Fewer Words

■ **Algorithm 5** strong BA algorithm: code for process p_i with initial value v_i .

Initially $decision, proof, bu_decision, bu_proof, fallback_val = \perp$
 $fallback_start \leftarrow \infty$

1: $leader \leftarrow p_1$
Round 1:
2: send $\langle v_i \rangle_{p_i}$ to $leader$
Round 2:
3: **if** $leader = p_i$ **then**
4: **if** received $t + 1$ messages of $\langle v \rangle_{p'}$ for some v **then**
5: batch these messages into $QC_{\text{propose}}(v)$ using a $(t + 1, n)$ -threshold signature scheme
6: broadcast the message $\langle \text{propose}, v, QC_{\text{propose}}(v) \rangle_{leader}$
Round 3:
7: **if** received valid $\langle \text{propose}, v, QC_{\text{propose}}(v) \rangle_{leader}$ **then**
8: send $\langle \text{decide}, v \rangle_{p_i}$ to $leader$
Round 4:
9: **if** $leader = p_i$ **then**
10: **if** received n messages of $\langle \text{decide}, v \rangle_{p'}$ **then**
11: batch these messages into $QC_{\text{decide}}(v)$ using a (n, n) -threshold signature scheme
12: broadcast the message $\langle \text{decide}, v, QC_{\text{decide}}(v) \rangle_{leader}$
Round 5:
13: **if** received valid $\langle \text{decide}, v, QC_{\text{decide}}(v) \rangle_{leader}$ and $decision = \perp$ **then**
14: $decision \leftarrow v$
15: $proof \leftarrow QC_{\text{decide}}(v)$
16: **else**
17: broadcast the message $\langle \text{fallback}, \perp, \perp \rangle_{p_i}$
18: $fallback_start \leftarrow now + 2\delta$
19: $bu_decision \leftarrow decision$
20: **while** $fallback_start > now$ **do**
21: **if** received $\langle \text{fallback}, v, proof_{p'} \rangle_{p'}$ **then**
22: **if** $decision = \perp$ and $proof_{p'} \neq \perp$ is a valid proof for a valid v **then**
23: $bu_decision \leftarrow v$
24: $bu_proof \leftarrow proof_{p'}$
25: **if** $fallback_start = \infty$ **then**
26: broadcast the message $\langle \text{fallback}, bu_decision, bu_proof \rangle_{p_i}$
27: $fallback_start \leftarrow now + 2\delta$
28: $fallback_val \leftarrow \mathcal{A}_{\text{fallback}}$ with $\delta' = 2\delta$ and initial value $bu_decision$
29: **if** $decision = \perp$ **then**
30: $decision \leftarrow fallback_val$

By Lemma 8, if all processes are correct then they never perform the fallback algorithm, and there are 4 all-to-leader and leader-to-all rounds, with a total of $O(n)$ words. Otherwise, the complexity is the complexity of the fallback algorithm, which is $O(n^2)$.

8 Conclusions and Future Directions

We have presented solutions for both Byzantine Broadcast and weak Byzantine Agreement with adaptive communication complexity of $O(n(f + 1))$ and resilience $n = 2t + 1$. To construct the weak BA algorithm, we utilized a threshold on the number of signatures such that on one hand, this number is sufficient to ensure a safe algorithm with adaptive communication in case there are not “many” Byzantine processes. On the other hand, failing to achieve this threshold indicates that there is a high number of failures, which allows the use of a quadratic fallback algorithm.

This weak BA algorithm is taken as a black box to construct our adaptive BB algorithm. Here, we carefully choose the predicate for the validity property, to allow us to reduce one problem to the other. Finally, for strong BA we propose a binary solution with optimal resilience. Our solution is linear in n in the practically common failure-free case, and quadratic in any other case. The question of whether a fully adaptive strong BA with optimal resilience exists or not remains open.

While $n = 2t + 1$ is optimal for strong BA, this is not the case for BB and weak BA, where any $t < n$ can be tolerated². Thus, another possible future direction is improving the resilience of an adaptive BB or adaptive weak BA to support any $t < n$. Our weak BA algorithm relies on the current resilience to satisfy that if $f > n - \lceil \frac{n+t+1}{2} \rceil$ then f is linear in t . Note that this remains true for any resilience of $n = \alpha t + \beta$, for $\alpha > 1, \beta > 0$ without compromising the intersection property required for safety. Should a quadratic solution for weak BA be developed, it could be used to improve the total resilience of our adaptive algorithm (instead of Momose and Ren’s algorithm [13]).

References

- 1 Ittai Abraham, Guy Golan-Gueta, and Dahlia Malkhi. Hot-stuff the linear, optimal-resilience, one-message bft devil. *CoRR*, abs/1803.05069, 2018. [arXiv:1803.05069](https://arxiv.org/abs/1803.05069).
- 2 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.
- 3 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.
- 4 Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *International conference on the theory and application of cryptography and information security*, pages 514–532. Springer, 2001.
- 5 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.
- 6 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.
- 7 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.
- 8 Yvo Desmedt. Society and group oriented cryptography: A new concept. In *Conference on the Theory and Application of Cryptographic Techniques*, pages 120–127. Springer, 1987.

² For weak BA, this stems from the resilience for external validity.

18:16 Make Every Word Count: Adaptive Byzantine Agreement with Fewer Words

- 9 Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for byzantine agreement. *Journal of the ACM (JACM)*, 32(1):191–204, 1985.
- 10 Danny Dolev, Ruediger Reischuk, and H Raymond Strong. Early stopping in byzantine agreement. *Journal of the ACM (JACM)*, 37(4):720–741, 1990.
- 11 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, SOSP '17*, New York, NY, USA, 2017. ACM. doi:10.1145/3132747.3132757.
- 12 Vassos Hadzilacos and Joseph Y Halpern. Message-optimal protocols for byzantine agreement. *Mathematical systems theory*, 26(1):41–102, 1993.
- 13 Atsuki Momose and Ling Ren. Optimal communication complexity of authenticated byzantine agreement. In *35th International Symposium on Distributed Computing (DISC 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.
- 14 Oded Naor and Idit Keidar. Expected linear round synchronization: The missing link for linear byzantine smr. In *34th International Symposium on Distributed Computing (DISC 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.
- 15 Victor Shoup. Practical threshold signatures. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 207–220. Springer, 2000.
- 16 Alexander Spiegelman. In search for an optimal authenticated byzantine agreement. In *35th International Symposium on Distributed Computing (DISC 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.
- 17 Zhuolun Xiang, Dahlia Malkhi, Kartik Nayak, and Ling Ren. Strengthened fault tolerance in byzantine fault tolerant replication. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 205–215. IEEE, 2021.

A Adaptive Byzantine Broadcast: Correctness

A.1 Correctness

We start by proving the phase’s requirements. First, immediately from lines 29 – 31 we get that all correct processes return either \perp or a valid v . Next, the following lemma shows that in non-silent phases with correct leaders all correct processes return a valid value.

► **Lemma 9.** *If a phase is non-silent and its leader is correct, then all correct processes return a valid value.*

Proof. If the leader is correct it broadcasts a `help_req` message at line 16. All correct processes then answer at round 2. If the leader receives a value signed by the sender at line 23, it broadcasts it at line 24. Otherwise, no correct processes received a value signed by the sender and sends an `idk` message at line 21. Since $n = 2t + 1$, the leader receives at least $t + 1$ `idk` messages (from the correct processes) and forms an `idk` certificate. It broadcasts this value at line 27. In both cases, all correct processes return a valid value at line 29. ◀

The next lemma proves that if all correct processes invoke a phase with a value other than \perp , then they can return only one type of a valid value – a value signed by the sender.

► **Lemma 10.** *If all correct processes invoke a phase with value $v \neq \perp$, there does not exist a value signed by $t + 1$ processes in the system.*

Proof. If all correct processes invoke a phase with value $v \neq \perp$, they reply to the `help_req` messages at line 19 and never send an `idk` message. Since there are at most t Byzantine processes, the leader cannot receive $t + 1$ `idk` messages and form an `idk` certificate signed by $t + 1$ different processes. ◀

We now prove the correctness of the BB algorithm. First, to be able to use the weak BA, all correct processes must execute it with valid initial values.

► **Lemma 11.** *All correct processes execute line 9 with a valid initial value.*

Proof. Let p_i be a correct process. In n phases, there is one phase with p_i as leader. If p_i has updated v_i prior to that phase, it happened either line 4 or at line 8. Immediately from the code we get that in both cases p_i updates a valid value. If p_i did not update a value, it initiates a non-silent phase, and by Lemma 9 returns a valid value. ◀

Note that agreement and termination stem immediately from the code and the correctness of the weak BA. The following lemma proves validity.

► **Lemma 12.** *If sender is correct, then all correct processes decide v_{sender} .*

Proof. If *sender* is correct then all correct process learn v_{sender} by the end of round 1 and update their values at line 4. By Lemma 10, in no phase can any process create a value signed by $t + 1$ processes. Thus, when executing the weak BA v_{sender} signed by the sender is the only valid value that exists in the run. By unique validity and since the *sender* does not sign more than one initial value, v_{sender} is the only possible BA output. It follows that all correct processes execute line 11 and return the sender's value. ◀

We conclude the following theorem:

► **Theorem 13.** *Algorithm 1 solves BB.*

B Weak BA: Correctness

We start by proving some lemmas about the phase's guarantees. First, we prove that if the *decision* is updated in a given phase, then its new value is valid.

► **Lemma 14.** *If a correct process updates decision during `invokePhase`, then v is a valid decision value.*

Proof. If a correct process updates its *decision* value at line 53 of `invokePhase` then it must have received a finalized certificate signed by $\lceil \frac{n+t+1}{2} \rceil$ processes. Hence, at least one correct process p' signed the decide message for v at line 44. By the code, p' signed the decide message for v if it received a commit certificate signed by $\lceil \frac{n+t+1}{2} \rceil$ processes. Hence, at least one correct process p'' signed the vote message for v at line 34. By the code, this is possible only if v is a valid value (line 33). ◀

Next, we prove that all correct processes that update their *decision* variable do so the same value. Moreover, at most one valid *decide_proof* can exist in the system. That is, a Byzantine process cannot devise a *decide_proof* that conflicts with any other *decide_proof* known by correct processes.

► **Lemma 15.** *All correct processes that update decision during `invokePhase` return the same decision. In addition, at most one finalize certificate can be formed in all phases.*

Proof. Note that every correct process that updates *decision* sets its at line 53. Assume that a correct process p_i sets its decision value to v in phase l and a correct process p_j sets its decision value to w in phase $k \geq l$.

18:18 Make Every Word Count: Adaptive Byzantine Agreement with Fewer Words

If $k = l$, then p_i and p_j set their decision value in the same round and they both receive a finalize certificate signed by $\lceil \frac{n+t+1}{2} \rceil$ different processes. At least one correct process signed both certificates and since correct processes sign at most one finalize message per phase, $v = w$.

For the case where $k > l$: in phase l , p_i receives a finalize certificate signed by $\lceil \frac{n+t+1}{2} \rceil$ different processes. Thus, at least $\lceil \frac{n+t+1}{2} \rceil - t \geq \frac{n-t+1}{2}$ correct processes updated their commit to v in that phase. These processes do not vote for any value in following rounds. Thus at most $n - t - \frac{n-t+1}{2} = \frac{n-t-1}{2}$ correct processes can sign a conflicting value. Since $\frac{n-t-1}{2} + t < \lceil \frac{n+t+1}{2} \rceil$, in any phase greater than l , no process can collect $\lceil \frac{n+t+1}{2} \rceil$ signatures on any value other than v . Hence, no process can send a commit message on $w \neq v$ with a valid commit certificate. For the same reason, no process can form and send a valid finalize certificate and decide upon any other value. Thus, $v = w$. ◀

We prove next that once a correct process is the leader of a non-silent phase, all correct processes return the same valid decision value by the end of that phase.

► **Lemma 16.** *If a correct leader invokes `invokePhase` in phase k and $f < \frac{n-t-1}{2}$, then all correct processes return the same valid decision by the end of the phase and this decision is a proposal of a correct process.*

Proof. The leader broadcasts its value v to all processes. If there is a correct process p for which `commit` $\neq \perp$, it sends the message $\langle \text{commit}, w, \text{proof}, j \rangle_p$ to the leader. If the leader receives $\langle \text{commit}, w, \text{proof}, j \rangle_{p'}$ (from any process), it broadcasts in round 3 a commit certificate for w . Otherwise, since $f < \frac{n-t-1}{2}$, leader receives $\lceil \frac{n+t+1}{2} \rceil$ messages voting for v and broadcasts a commit certificate for v . Then, all correct processes send the leader a finalize messages on v or w . Again, the leader receives $\lceil \frac{n+t+1}{2} \rceil$ messages finalizing v and broadcasts a finalize certificate for v . Correct processes receive this message and update their `decision` and `decide_proof` accordingly. Then, by the code they all return v . ◀

We now prove the correctness of the main BA algorithm. The following two lemmas prove that although some processes may start executing $\mathcal{A}_{\text{fallback}}$ at different times, they all successfully execute the fallback algorithm.

► **Lemma 17.** *If some correct process executes the fallback algorithm in Algorithm 3, all correct process do so and they all start at most δ time apart.*

Proof. Let p be the first correct process that executes the fallback algorithm at line 24 of Algorithm 3 at time t . This means that at time $t - 2\delta$, p broadcasts the fallback certificate to all other processes (line 22). By synchrony, this certificate is guaranteed to arrive at all correct processes by $t - \delta$, causing them to execute the fallback algorithm by $t + \delta$ if they have not done so earlier. ◀

► **Lemma 18.** *Consider a synchronous algorithm \mathcal{A} . Let σ be a synchronized run of \mathcal{A} defined as follows. Let t be the time that the first correct process starts executing \mathcal{A} in σ . All correct processes start executing \mathcal{A} by $t + \delta$. The round duration is 2δ . In round r that begins (locally) in t_r , round r messages are processed if they are received in the time window $[t_r - \delta, t_r + 2\delta]$. Then σ is a correct run of \mathcal{A} .*

Proof. Consider a process p that starts round r at time t_r^p . Let p' be another correct process that starts round r at time $t_r^{p'}$, and sends a message to p in round r . By assumption, $t_r^{p'} = t_r^p + \epsilon$ where $-\delta \leq \epsilon \leq \delta$, and a message sent by p' at $t_r^{p'}$ arrives at time t_a where $t_r^{p'} \leq t_a \leq t_r^{p'} + \delta$. Note that round r ends at p at time $t_{r+1}^p = t_r^p + 2\delta$. Hence, $t_r^p - \delta \leq t_a \leq t_r^p + 2\delta$, as needed. ◀

Next, we the following lemma states that if a correct process manages to reach a decision prior to the fallback algorithm, then this is the only possible decision. Moreover, this decision value must be a valid one.

► **Lemma 19.** *If some correct process decides v before executing the fallback algorithm, then all correct processes decide v and v is valid.*

Proof. If there exists a correct process p that decides at line 4, then by Lemma 15 and the code all processes that decide at line 4 decide v as well. Moreover, all other correct process that have not decided by line 5, send `help_req` messages. Process p answers them and they all decide at line 14. Otherwise, no correct process decides at line 4 and they all send `help_req` messages at line 6. Then, they all receive $t + 1$ `help` messages and by the code perform the fallback algorithm. In addition, by the lemma assumption, it must be that p decides v at line 14.

If correct processes execute the fallback algorithm, then by the code they all wait a time period of 2δ before the execution, during which they receive all decisions made by other correct processes and update `bu_decision` accordingly (line 24). Specifically, they receive v from p . It follows from Lemma 15 that `bu_decision` is updated with the same value at all correct processes. Thus, all correct processes execute $\mathcal{A}_{fallback}$ with the same input, and by strong unanimity they set `fallback_val` to v at line 24.

We now prove that v is valid. If p decides v at line 4, then it must have updated `decision` in the scope of the relevant phase. By Lemma 14 this value is valid. Otherwise, if p decides v at line 14, then the validity follows from the code. Hence, since v is valid, all correct processes decide it by line 27. ◀

Finally, we are ready to prove the required BA properties.

► **Lemma 20 (Agreement).** *In Algorithm 3 all correct process decide on the same value.*

Proof. First, by Lemma 15, all correct processes that decide in line 4 decide the same value v . In addition, it follows from the same lemma that every correct process that decides at line 14 after receiving a valid `finalize` certificate decides v , as at most one `finalize` certificate can be formed.

It is left to show that if not all correct processes decide before the fallback algorithm at line 24, they still decide upon the same value. If at least one correct process p receives a fallback certificate it follows from Lemma 17 that all correct processes receive the certificate within at most δ time of p . Then, by the code, all correct process execute the fallback algorithm at line 24 and by Lemma 18 and the fallback algorithm solves strong BA, providing agreement. By Lemma 19, we get that processes that decide before running the fallback decide on the same value. ◀

► **Lemma 21 (Termination).** *In Algorithm 3 all correct process decide.*

Proof. If not all correct processes decide before line 5 and no correct process receives a fallback certificate, it follows that less than $t + 1$ correct processes broadcast `help` messages at line 6. Hence, at least one correct process p has decided by line 5. Process p receives all of the correct `help` messages at line 7 and answers them at line 8. All correct processes that asked for `help` then decide at line 14.

It remains to examine the case that at least one correct process p receives a fallback certificate. It follows from Lemma 17 that all correct processes receive the certificate within at most δ time of p . Then, by the code, all correct process execute the fallback algorithm at line 24 and by Lemma 18 and the fallback algorithm solves BA, providing termination. ◀

18:20 Make Every Word Count: Adaptive Byzantine Agreement with Fewer Words

► **Lemma 22** (Unique Validity). *In Algorithm 3 if a correct process decides v then either $v = \perp$ or $\text{validate}(v) = \text{true}$, and if $v = \perp$ then more than one valid value exists in the run.*

Proof. Let v be the decision value of a correct process in Algorithm 3. First, by lines 27 – 29 $\text{validate}(v) = \text{true}$ or $v = \perp$. We prove that if $v = \perp$, then at least two valid values exist in the run.

By the code, all processes execute the fallback algorithm with valid inputs (either their initial valid values, or a valid value they adopt at line 19). By strong unanimity of $\mathcal{A}_{\text{fallback}}$, if all correct processes start with the same valid value v' , then v' must be the returned decision value. This contradicts the fact that \perp is returned at line 29. Therefore, not all correct processes execute $\mathcal{A}_{\text{fallback}}$ with the same value. As they all execute the fallback algorithm with valid inputs, it follows that at least two valid values exist in the run. ◀

In addition, we need to prove that every correct process updates its *decision* at most once.

► **Lemma 23.** *In Algorithm 3 all correct process decide at most once.*

Proof. Any correct process updates *decision* at line 4, line 14 or lines 27 – 29. In all cases, it only does so if *decision* = *undecided*. Since by the code it does not update *decision* to the value *undecided*, it follows that *decision* is updated at most once. ◀

From Lemmas 20, 21, 22, and 23 we conclude:

► **Theorem 24.** *Algorithm 3 solves weak BA.*

C Strong BA: Correctness

► **Lemma 25.** *If some correct process executes the fallback algorithm in Algorithm 5, all correct process do so and they all start at at most δ time apart.*

Proof is similar to Lemma 17 in Section 6.

► **Lemma 26** (Agreement). *In Algorithm 5 all correct process decide on the same value.*

Proof. First, as correct processes only sign one *decide* message, every process that receives $QC_{\text{decide}}(v)$ receives the same quorum certificate. Thus, all correct processes that decide at line 14 decide the same v . If at least one correct process receives a fallback message then by Lemma 25, they all execute the fallback algorithm at most δ time apart. Thus, if at least one correct process decides at line 14, then all correct processes that have not yet decided learn about v in the 2δ safety window, and adopt it as their initial value for the fallback (line 23). It follows that all correct processes decide with the same input value v and by strong unanimity this is the only possible decision. ◀

► **Lemma 27** (Termination). *In Algorithm 5 all correct process decide.*

Proof. If not all correct processes decide by line 14, then a correct process broadcasts a fallback message at line 17. It follows from Lemma 25 that all correct processes receive the certificate within at most δ time of p . Then, by the code, all correct process execute the fallback algorithm at line 28 and by Lemma 18 and the fallback algorithm solves strong BA, providing termination. ◀

► **Lemma 28** (Validity). *In Algorithm 5 if all correct processes propose the same value v , then the output is v .*

Proof. Correct processes only send **decide** messages on values with valid **propose** quorum certificates. Note that such a quorum certificate can only be formed with $t + 1$ unique signatures. Hence, if all correct processes propose the same value v , then the only possible **propose** quorum certificate is with v . As a result, the only possible **decide** quorum certificate is with v as well.

The fallback algorithm is executed with either the original initial values or with a value that has a corresponding **decide** quorum certificate. Thus, if correct processes execute the fallback algorithm, they all start with v and by strong unanimity of $\mathcal{A}_{fallback}$, the decision is v . ◀

Finally, we prove that every correct process updates its *decision* at most once.

► **Lemma 29.** *In Algorithm 5 all correct process decide at most once.*

Proof. Any correct process updates *decision* either at line 14 or at line 30. In both cases, it only does so if *decision* = \perp . Since it does not update *decision* to the value \perp at any step of the algorithm, it follows that *decision* is updated at most once. ◀

From Lemmas 26, 27, 28, and 29 we conclude:

► **Theorem 30.** *Algorithm 5 solves binary strong BA.*


Modeling Resources in Permissionless Longest-Chain Total-Order Broadcast

Sarah Azouvi ✉

Protocol Labs

Christian Cachin ✉ 

University of Bern, Switzerland

Duc V. Le ✉ 

University of Bern, Switzerland

Marko Vukolić ✉

Protocol Labs

Luca Zanolini ✉ 

University of Bern, Switzerland

Abstract

Blockchain protocols implement total-order broadcast in a permissionless setting, where processes can freely join and leave. In such a setting, to safeguard against Sybil attacks, correct processes rely on cryptographic proofs tied to a particular type of *resource* to make them eligible to order transactions. For example, in the case of Proof-of-Work (PoW), this resource is computation, and the proof is a solution to a computationally hard puzzle. Conversely, in Proof-of-Stake (PoS), the resource corresponds to the number of coins that every process in the system owns, and a secure lottery selects a process for participation proportionally to its coin holdings.

Although many resource-based blockchain protocols are formally proven secure in the literature, the existing security proofs fail to demonstrate why particular types of resources cause the blockchain protocols to be vulnerable to distinct classes of attacks. For instance, PoS systems are more vulnerable to long-range attacks, where an adversary corrupts past processes to re-write the history, than PoW and Proof-of-Storage systems. Proof-of-Storage-based and PoS-based protocols are both more susceptible to private double-spending attacks than PoW-based protocols; in this case, an adversary mines its chain in secret without sharing its blocks with the rest of the processes until the end of the attack.

In this paper, we formally characterize the properties of resources through an abstraction called *resource allocator* and give a framework for understanding longest-chain consensus protocols based on different underlying resources. In addition, we use this resource allocator to demonstrate security trade-offs between various resources focusing on well-known attacks (e.g., the long-range attack and nothing-at-stake attacks).

2012 ACM Subject Classification Theory of computation → Cryptographic protocols; Software and its engineering → Distributed systems organizing principles

Keywords and phrases blockchain, consensus, resource, broadcast

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2022.19

Related Version *Full Version*: <https://arxiv.org/abs/2211.12050>

Funding DVL has been supported by a grant from Protocol Labs to the University of Bern. LZ has been supported by the Swiss National Science Foundation (SNSF) under grant agreement Nr. 200021_188443 (Advanced Consensus Protocols).

Acknowledgements The authors thank anonymous reviewers for helpful feedback.



© Sarah Azouvi, Christian Cachin, Duc V. Le, Marko Vukolić, and Luca Zanolini; licensed under Creative Commons License CC-BY 4.0

26th International Conference on Principles of Distributed Systems (OPODIS 2022).

Editors: Eshcar Hillel, Roberto Palmieri, and Etienne Rivière; Article No. 19; pp. 19:1–19:23

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Permissionless consensus protocols are open for everyone to participate and often rely on a *resource* to protect against Sybil attacks. In the case of Proof-of-Work (PoW), this resource is computation: A computational puzzle must be solved in order to gain writing rights in the system. In contrast, in a Proof-of-Stake (PoS) system, writing access is granted using a form of lottery where participants are elected proportionally to the number of coins they own. Other resource-based systems, such as Proof-of-Storage, have also appeared. Participants are elected proportionally to the number of resources they commit to the system, and hence this commitment must be publicly verifiable. Different resources present different trade-offs. For example, PoS is much more energy-efficient than PoW but presents many additional vulnerabilities [5]. Comparing the security of protocols based on multiple resource types is a non-trivial task, as they use different assumptions and frameworks.

In this paper, we provide a common framework to formally compare consensus protocols based on different underlying resources. We only consider longest-chain protocols [11] that rely on an underlying resource, as we want to highlight the properties affected by varying the resource for the same consensus method. In future work, our framework could be used to model further approaches to ensure consensus, such as well-known BFT protocols [7], for instance. In longest-chain protocols, one participant is elected at each time step, on expectation, proportionally to their amount of resource and that participant gets to write to the append-only database by adding a *block* containing all the necessary data to the longest chain of blocks.

We also explore known attacks in this work. The first one is the long-range attack. In a long-range attack, an adversary corrupts processes that used to participate in the system but that no longer hold any resources. Moreover, we investigate nothing-at-stake attacks, where processes mine on multiple chains at the same time, and private attacks, where an adversary mines on its own chain without contributing to the honest chain. We are interested in quantifying gain or loss of security with different resources. It has already been shown that, when considering longest-chain protocols, PoS is less secure than PoW. We furthermore show that Proof-of-Storage stands in the middle, as storage is not virtual (like stake), but is reusable (unlike the computation of PoW).

We start the paper by providing a formal framework in which protocols based on different resources can meaningfully be compared. We differentiate between virtual and external resources to highlight which properties make longest-chain PoS and Proof-of-Storage less secure than PoW, although they both present trade-offs when it comes to their efficiency.

Contributions. Our contributions can be summarized as follows.

- We formally characterize the properties of resources through an abstraction called *resource allocator* and formally define properties for a *secure* resource allocator.
- We concretely define different resource allocator abstractions, each one for every type of resource used in popular blockchain protocols, namely, computation, stake, and storage.
- We present an algorithm that, when instantiated with different resource allocators, leads to a generalization of existing protocols such as Nakamoto consensus, Ouroboros Praos, and Filecoin's consensus protocol. We also show formally that this generalization implements total-ordered broadcast under a fixed total resource in a permissionless setting.
- We demonstrate how different resources lead to different security trade-offs by leveraging our model to explain long-range attacks against virtual resources and attacks related to the nothing-at-stake nature of reusable resources.

Related Work. Since the emergence of Bitcoin in 2008, the academic community has developed a number of frameworks [16, 22, 18, 11] for studying the safety and liveness properties of its Nakamoto consensus protocol. These studies also established a strong foundation for the development of blockchain protocols based on more eco-friendly types of resources, such as stake and storage. However, despite the fact that all resource-based blockchains have been formally proven to be secure, these results have failed to explain why certain properties of resources make some blockchain protocols more susceptible to particular types of attacks than others. To the best of our knowledge, no prior work has attempted to formally study the properties of underlying resources, and our work aims to fill this gap.

Lewis-Pye and Roughgarden [21] present the concept of a *resource pool* that reflects the resource balance of processes in the system at any time, and they use a *permitter* together with the resource pool to abstract away the leader selection procedure. Using this formalization, they demonstrate two crucial impossibility results for permissionless systems. Two main results of their work are: (i) no permissionless, deterministic, and decentralized protocol solves the Byzantine Agreement problem in a synchronous setting, and (ii) no permissionless and probabilistic protocols solve the Byzantine Agreement problem in the unsized setting (in which the total number of resources is unknown) with partially synchronous communication. However, their work could not capture several aspects of underlying resources used in blockchain protocols; therefore, their work did not demonstrate long-range attacks against virtual resources such as stake, and the cost of several other attacks on *reusable* resources. Our work takes a similar approach of abstracting away the leader selection process with a resource allocator (c.f., Section 3), and we further formalize the properties of resources through the interactions between the process and this allocator. With this formalization, we prove how permissionless and probabilistic blockchain protocols guarantee properties of a total-order broadcast in a synchronous setting and demonstrate various attacks against *virtual* or *reusable* resources.

Terner [25] also investigates how to abstract resources used in permissionless blockchains. While this work outlines several essential properties of resources and studies how the resource generation rate affects the standard properties (i.e., consistency and liveness) of robust transaction ledger, the study does not characterize the properties of the underlying resources used in permissionless blockchain protocols. Consequently, this model fails to explain why distinct types of resources render some protocols vulnerable to certain attacks (e.g., long-range attacks and private attacks).

2 Model and Definitions

2.1 System Model

Time. We assume that the protocol proceeds in *time steps* and define a time step to be a value in \mathbb{N} . Moreover, we consider 0 as starting time step of protocol execution.

Processes. We consider a system consisting of a set of *processes*, $\mathcal{P} = \{p_1, p_2, \dots\}$. Processes interact with each other through exchanging messages. A protocol for \mathcal{P} consists of a collection of programs with instructions for all processes. Moreover, to capture the permissionless nature of various blockchain protocols, processes can join the system at any time. We denote, $\mathcal{P}_{\leq t}$, the set of all processes that have participated in the protocol before the time step t . Hence, $\mathcal{P}_t \subseteq \mathcal{P}_{t'}$ for all $t \leq t'$. At the beginning of each time step, a process becomes *activated*, and it starts to follow a deterministic protocol. This includes processing any messages that may have arrived from other processes. Once done, it becomes *deactivated*. We assume that the activation period of a process p_i starts at the time step t and ends before time step $t + 1$.

Communication. We assume there is a low-level primitive for sending messages over point-to-point links between each pair of processes that know of each other, as well as a probabilistic broadcast primitive [7]. Point-to-point messages are authenticated and delivered reliably among correct processes. In probabilistic broadcast, correct processes *gossip-deliver* and *gossip-broadcast* messages with an overwhelming probability, no message is delivered more than once, and no message is created or corrupted by the network.

Network Delay. We denote by $\Delta \in \mathbb{N}$ with $\Delta \geq 1$ the maximum network delay [14]. Namely, if a correct process *gossip-broadcasts* a message m at a time step t , then other processes will have *gossip-delivered* or received over the message by the beginning of a time step $t + \Delta$ with an overwhelming probability.

Idealized Digital Signature. A digital signature scheme, Σ , consists of two operations, $\text{Sign}(\cdot, \cdot)$ and $\text{Verify}(\cdot, \cdot, \cdot)$. The operation $\text{Sign}(p_i, \cdot)$ invoked by p_i takes $m \in \{0, 1\}^*$ as input and returns a signature $\sigma \in \{0, 1\}^*$. Only p_i can invoke $\text{Sign}(p_i, \cdot)$. The operation $\text{Verify}(p_i, \cdot, \cdot)$ takes as input a signature, σ , and a message m ; $\text{Verify}(p_i, \cdot, \cdot)$ returns TRUE for any $p_i \in \mathcal{P}$ and $m \in \{0, 1\}^*$ if and only if p_i has invoked $\text{Sign}(p_i, m)$ and obtained σ before. Any process can invoke $\text{Verify}(\cdot, \cdot, \cdot)$.

Random Oracle. All hash functions are modeled as a random oracle, H , that can be queried by any process. H takes as input a bit string $x \in \{0, 1\}^*$ and returns a uniformly random string from $\{0, 1\}^\lambda$ where λ is the security parameter. Also, upon repeated queries, H always outputs the same answer.

2.2 Modeling Blockchain Data Structures

Blocks. We use tx to denote a *transaction*. We write $\overline{\text{tx}} = [\text{tx}_1, \dots, \text{tx}_m]$ to denote a list of transactions. A *block* is $B = (h, \overline{\text{tx}}, \pi, \sigma_i)$, where h is a hash value, $\overline{\text{tx}}$ is a list of transactions, π is a resource commitment proof (cf. Section 3) and σ_i is a signature on $(h, \overline{\text{tx}}, \pi)$. In this work, we assume that blocks are signed. In this way, we can abstract away the notion of *coinbase* transactions, i.e., the first transaction in a block, created by a miner, and used to collect the block reward. Finally, we denote with $B_0 = (\perp, \overline{\text{tx}}, \perp, \perp)$ the *genesis block*.

Blockchain. A *blockchain* $\mathcal{C} = [B_0, B_1, \dots]$ with respect to the genesis block B_0 is a chain of blocks forming a hash chain such that $h_j = H(B_{j-1})$ for $h_j \in B_j$ for $j = 1, 2, \dots$ with $B_j = (h_j, \overline{\text{tx}}_j, \pi_j, \sigma_j)$. For a blockchain \mathcal{C} , we use $\mathcal{C}[-k]$ to denote the last k -th block in \mathcal{C} , let $\mathcal{C}[k]$ to denote block B_k (i.e., block at height k), and write $\mathcal{C}[: -k]$ to denote the first $|\mathcal{C}| - k$ blocks. $|\mathcal{C}|$ denotes the length of \mathcal{C} . We write $\mathcal{C} \preceq \mathcal{C}'$ when \mathcal{C} is a prefix \mathcal{C}' . We use \mathcal{C}^t to denote the blockchain at time step t . For two time steps, t_1 and t_2 , $\mathcal{C}^{t_2}/\mathcal{C}^{t_1}$ is a set of blocks that is in \mathcal{C}^{t_2} but not in \mathcal{C}^{t_1} .

State. The blockchain *state* st specifies different information of the underlying blockchain protocol, e.g., the stake distribution of each process, the block information, such as timestamps, as well as contract local states. The blockchain state st can be reconstructed by executing transactions included in a blockchain \mathcal{C} . Without loss of generality, we define the state to be the blockchain, $st = \mathcal{C}$. Also, we write $st = (\mathcal{C}, B)$ to indicate that a block B is potentially appended to \mathcal{C} .

Validity. We introduce the notion of *validity* for transactions and blockchains to capture the fact that only “valid” transactions are delivered. More importantly, for all blockchain protocols, the decision on the validity is determined locally by all processes. Because of this, we define the validity as follows. A transaction x is *valid* with respect to \mathcal{C} if tx satisfies a *validation predicate* $\mathbb{P}(\mathcal{C}, \cdot)$ locally known to all processes (i.e., $\mathbb{P}(\mathcal{C}, [\text{tx}]) = \text{TRUE}$). We also use $\mathbb{P}(\mathcal{C}, \bar{\text{tx}}) = \text{TRUE}$ to indicate that the sequence of transactions in $\bar{\text{tx}}$ is valid (i.e., does not consume the same output in Bitcoin or the same nonce in Ethereum), and we define $\mathbb{P}(\mathcal{C}, [\])$ to be TRUE . Depending on the blockchain protocol, a *valid block* B issued by p_i should consist of a valid signature issued by p_i , a valid “proof” π for a so-called resource commitment that we introduce in Section 3 and valid transactions with respect to \mathcal{C} such that $\mathcal{C}[-1] = B$, (i.e., $\mathbb{P}(\mathcal{C}, \bar{\text{tx}}) = \text{TRUE}$ for $\bar{\text{tx}} \in B$). Finally, *valid blockchains* are chains that consist of only valid blocks and start from the genesis block B_0 .

2.3 Total-order Broadcast

We will show that the blockchain protocols considered here guarantee the following properties of total-order broadcast in a permissionless setting. In particular, total-order broadcast ensures that all processes deliver the same set of transactions in a common global order. In total-order broadcast, every process broadcasts a transaction by invoking $a\text{-broadcasts}(\text{tx})$. The broadcast primitive outputs a transaction tx through an $a\text{-deliver}(\text{tx})$ event. In this model, we do not distinguish between a process and a client. A client can be considered as a process that only broadcast transactions and does not participate in mining.

► **Definition 2.1** (Total-order Broadcast). *A protocol for total-order broadcast satisfies the following properties.*

Validity *If a correct process, p_i a -broadcasts a valid transaction tx according to $\mathbb{P}(\cdot, \cdot)$ (i.e., the validation predicate defined in Section 2), then p_i eventually a -delivers tx with an overwhelming probability.*

No duplication *No correct process a -delivers the same transaction tx more than once.*

Agreement *If a transaction tx is a -delivered by some correct process, then with an overwhelming probability tx is eventually a -delivered by every correct process.*

Total order *Let tx_1 and tx_2 be any two transactions, and suppose p_i and p_j are any two correct processes that a -deliver tx_1 and tx_2 . If p_i a -delivers tx_1 before tx_2 , then with an overwhelming probability, p_j a -delivers tx_1 before tx_2 .*

3 Modeling Resources in Blockchain

In this section, we model resources, formalize their properties through the abstraction of a *resource allocator*, and state our threat assumptions. The definition of a *resource allocator* in this section is only syntactic; security and liveness properties of the resource allocator are defined in Section 4.

► **Definition 3.1** (Resource Budget). *A resource budget r is a value in \mathbb{N} . At any given time, each process p_i has a resource budget r_i . In particular, there exists a function $\text{Alloc} : \mathcal{P} \times \mathbb{N} \rightarrow \mathbb{N}$ that takes as input a process p_i and a time step t , outputs the resource budget of a process at time step t . We define R to be the fixed resource budget existing in the system.*

The definition of a fixed resource budget and the resource allocation function can be viewed as the sized setting and the resource pool definition in Lewis-Pye and Roughgarden framework [21].

We note that the specification of the resource budget varies depending on protocols; e.g., for PoW, we define the budget to be a number of hash function evaluations per time step. We now define *resource allocator*, an abstraction that will allow us to reason about different resources.

► **Definition 3.2 (Resource Allocator).** *A resource allocator, RA, interacts with the processes through input events (RA-commit, RA-validate) and output events (RA-assign, RA-is-committed):*

- *RA-commit(p_i, st, r): At time step t , every process p_i may request a resource commitment π from the resource allocator by invoking RA-commit on inputs a state st and a resource budget $0 \leq r \leq \text{Alloc}(p_i, t)$, i.e., p_i does not RA-commit more resources than it possesses. At the end of the activation period of p_i , the resource allocator either assigns a resource commitment π and a resource budget r to process p_i through an RA-assign(p_i, st, r, π) event or assigns an empty value \perp and possibly a resource r to p_i through RA-assign(p_i, st, r, \perp).*
- *RA-validate(p_i, st, π): Every process p_i may validate a resource commitment π by invoking RA-validate on input a state, st , and a resource commitment π . The resource allocator validates the resource commitment π , through an event RA-is-committed(p_i, st, b) event, with $b = \text{TRUE}$ if the commitment π is a valid resource commitment for the state st or $b = \text{FALSE}$ otherwise.*

A process triggers RA-commit to pledge its resources to a system, and it can be assigned a resource commitment as a result to extend the blockchain. If the resource commitment π is included on-chain, then it must be *valid* (i.e., RA-validate returns TRUE) for the block to be accepted. Moreover, we assume that all the events to and from the resource allocator happen within the same time step. In particular, if a process p_i RA-commits some resource budget r at time step t , at the end of the activation period for p_i process p_i will receive either a resource commitment π and r or an empty resource commitment value \perp and r .

Resources can be classified into various types. In our model, these types can be described as the interactions between processes and the resource allocator. The following definition classifies different types of resources used in existing blockchain protocols.

► **Definition 3.3 (Types of resource).** *A resource can be classified as follows.*

Virtual *A resource is virtual when the resource allocator determines the resource budget of all processes from the given blockchain state st . For a virtual resource, we assume that there exists a function $\text{StateAlloc} : \mathcal{P} \times \mathbb{C} \rightarrow \mathbb{N}$ that takes as input a process p_i and a blockchain \mathbb{C} and outputs the resource budget of p_i , and p_i can invoke RA-commit(\cdot, \cdot, r) on an empty resource, $r = \perp$.*

External *A resource is external when a process must allocate the resource externally with a budget $r \geq 0$ to invoke RA-commit(\cdot). For an external resource, this commitment step is equivalent to giving RA access to the external resource with the budget r . Moreover, we assume that processes cannot lie about the resource budget r and commit more than r .*

Burnable *A resource is burnable when a process p can trigger multiple RA-commit(\cdot, \cdot, r) at a time step t , and it retrieves r through RA-assign(\cdot, \cdot, r, \cdot) at the end of the activation period for p_i . For all committing events RA-commit(p_i, \cdot, r_i) from the same process p_i that occur within a time step t , we require $\sum_{r_i > 0} r_i \leq \text{Alloc}(p_i, t)$.*

Reusable *A resource is reusable when a process p_i can use the same resource budget $r \leq \text{Alloc}(p_i, t)$ to trigger infinitely many RA-commit(\cdot, \cdot, r) at each time step t , and p_i does not need to retrieve r from the output event RA-assign. Hence, for reusable resources, we denote the value of r in the output event RA-assign(\cdot, \cdot, r, \cdot) to be \perp .*

► **Remark 3.4.** The assumption on external resources is natural because an *external* resource is inherently unforgeable; for instance, in PoW, processes cannot fake this budget as it is the physical limit of the mining hardware. For resources like storage, the resource is the physical hard drive, and r can be thought of as the capacity of the hard drive.

Failures. A process that follows its protocol during an execution is called *correct*. On the other hand, a *faulty* process may crash or deviate arbitrarily from its specification, such processes are also called *Byzantine*. We consider only Byzantine faults in this work. All Byzantine processes are controlled by a probabilistic polynomial-time adversary, \mathcal{A} ; we write $p_i \in \mathcal{A}$ to denote that a Byzantine process is controlled by \mathcal{A} . In this model, we require the adversary to go through the same process of committing resources and getting assigned resource commitments from the allocator. Since the allocator assigns the commitment at the end of the time step, we require a minimum delay between Byzantine processes to be one. We also note that this requirement is only for definitional reasons and can be relaxed by assuming the network delay to be zero for Byzantine processes. However, the concrete parameters on the probability of getting assigned resource commitments for Byzantine processes will need to be adjusted to reflect this assumption, and we leave this to future work.

► **Definition 3.5** (Adversarial Resource Budget). $R_{\mathcal{A}}$ is the maximum adversarial resource budget. For any time step t , it holds that: $\sum_{p_i \in \mathcal{A}} \text{Alloc}(p_i, t) \leq R_{\mathcal{A}}$.

► **Definition 3.6** (Corruption). At any time step t , an adversary \mathcal{A} can allocate a resource budget of $\text{Alloc}(p_i, t)$ from $R_{\mathcal{A}}$ to corrupt a process $p_i \in \mathcal{P}_{\leq t}$.

4 Resource-based Total-order Broadcast

In this section, we define an algorithm for the *resource-based longest-chain total-order broadcast* using a (*probabilistic*) *resource allocator* RA_{res} . We define various properties needed for a secure resource allocator so that the generic algorithm correctly guarantees properties of total-order broadcast. Then, we concretely define three different resource allocators based on three types of resource: computation, stake, and storage to inherently capture three popular (*probabilistic*) blockchain protocols, namely, Nakamoto consensus, Ouroboros Praos, and Filecoin's consensus protocol. However, due to space constraints, the description of the Proof-of-Storage allocator can be found in the full version of the paper.

unordered: set of transaction tx that has been received for execution and ordering
delivered: set of transaction tx that has been executed and ordered
 k : common prefix parameter
 \mathcal{B} : set of received blocks, $B = (h, \overline{\text{tx}}, \pi, \sigma)$, initially containing $B_0 = (\perp, \overline{\text{tx}}, \perp, \perp)$
 \mathcal{C} : set of valid blockchains derived from \mathcal{B} , initially contains one chain $\mathcal{C} = [B_0]$
 $\mathcal{C}_{\text{local}}$: local selected blockchain
 B_{com} : a RA -committed block for p_i
 At time step t , $r_i = \text{Alloc}(p_i, t)$ if r_i is *external*, $r_i \leftarrow \perp$ if r_i is *virtual*

■ **Figure 1** Initial state of a correct process.

4.1 Generic Resource-based Longest-chain Total-order Broadcast

A protocol for resource-based longest-chain total-order broadcast using RA_{res} allows any process p_i to *broadcast* transactions by invoking $a\text{-broadcast}(\text{tx})$ and to *deliver* those that are valid (according to a validation predicate $\mathbb{P}(\cdot, \cdot)$ and the local chain, $\mathcal{C}_{\text{local}}$) through an $a\text{-deliver}(\text{tx})$ event. Delivered transactions are *totally ordered* and stored in a list, *delivered*, by every process.

In particular, when a process p_i *a-broadcasts* a transaction, this is gossiped to every process, and eventually every correct process *gossip-delivers* it and stores it in a set *unordered*. Every stored transaction is then considered by p_i .

At any given time, a process may receive new blocks from other processes. Any process p_i can validate the block by invoking $RA\text{-validate}$ and $RA\text{-assigned}$ resource commitment to a process p_j by RA_{res} . Once the resource commitment is validated, the process verifies other components of the block such as signature and transactions and store new blocks in \mathcal{B} . Also, if a block B received by other processes does not have a parent (L22), the process can trigger a request message to pull the blockchain \mathcal{C} with B as the tip from other processes. Upon receiving this request message, other processes re-broadcast every blocks in \mathcal{C} with B as the tip (L13-L16). This step is an oversimplified and inefficient version of how blockchain nodes synchronize the chain with others. The goal is to demonstrate that it is feasible to obtain old blocks from other processes.

At any given time, a correct process adopts the longest chain to its knowledge as its local chain $\mathcal{C}_{\text{local}}$, and *extends* with a block B_{com} it wishes to order at the last block of its local chain $\mathcal{C}_{\text{local}}$. Observe that the *Extend* function in Algorithm 1 captures the operation of creating new blocks, usually called *mining*, in blockchain protocols and we refer to the processes in charge of creating blocks as *miners* or *validators*, interchangeably.

In our model, we abstract this *extending* operation as the interaction between the processes and the resource allocator RA_{res} . Namely, to start extending, process p_i needs to allocate a resource r along with the proposed state $(\mathcal{C}_{\text{local}}, B_{\text{com}})$ to the resource allocator RA_{res} through $RA\text{-commit}()$. Once RA_{res} *assigns* a resource commitment π , p_i attaches π to the block and gossips the block to other processes. The details of this interaction differ depending on the type of resource and are left for the next subsections. Figure 1 specifies all data structures maintained by a process, and the code for a process is presented in Algorithm 1.

For Algorithm 1 to satisfy the properties of total-order broadcast, the generic resource allocator needs to satisfy various properties, and we define those properties as follows.

► **Definition 4.1** (Secure Resource Allocator). *A resource allocator R is secure if it satisfies the following properties:*

Liveness *At a time step t , if a process p_i invokes $RA\text{-commit}(p_i, st, r)$ with a state st and a resource budget r then R issues either $RA\text{-assign}(p_i, st, r, \pi)$ or $RA\text{-assign}(p_i, st, r, \perp)$ during time step t .*

Validity *If resource commitment π is a valid resource commitment (i.e., $\pi \neq \perp$) contained in an output event $RA\text{-assign}(p_i, st, r, \pi)$, then any process p_j can invoke $RA\text{-validate}(p_j, st, \pi)$. The resource allocator R outputs $RA\text{-is-committed}(p_j, st, b)$ with $b = \text{TRUE}$.*

Use-Once *At any time step t , for any states st, st_1, st_2 , any resource budget $r, r_1, r_2 \in \mathbb{N}$ such that $r_1 + r_2 = r$, the probability that RA responds with $RA\text{-assign}(p_i, st, r, \pi)$ with $\pi \neq \perp$ after $RA\text{-commit}(p_i, st, r)$ is greater or equal to the probability that RA responds at least one $RA\text{-assign}(p_i, st_i, r_i, \pi)$ for $i \in \{1, 2\}$ with $\pi \neq \perp$ after two $RA\text{-commit}(p_i, st_1, r_1)$ and $RA\text{-commit}(p_i, st_2, r_2)$.*

■ **Algorithm 1** Resource-based longest-chain total-order broadcast (process p_i).

```

1: uses
2:   Resource allocator:  $RA_{\text{res}}$ 
3:   Probabilistic reliable broadcast: gossip
4:   Validation predicate:  $\mathbb{P}(\cdot, \cdot)$ 
5:   Random oracle:  $H : \{0, 1\}^* \rightarrow \{0, 1\}^\lambda$ 
6:   Signature scheme:  $\Sigma = (\text{Sign}, \text{Verify})$ 
7: init
8:    $\text{Extend}(\mathcal{C} = [B_0])$ 
9: upon a-broadcast(tx) do
10:   invoke gossip-broadcast([OP, tx])
11: upon gossip-deliver ([OP, tx]) do
12:    $\text{unordered} \leftarrow \text{unordered} \cup \{\text{tx}\}$ 
13: upon gossip-deliver ([REQUEST, B]) do ▷ Receive a request for parents of B
14:   if  $\exists \mathcal{C} \in \mathbb{C}$  s.t.  $\mathcal{C}[-1] = B$  then
15:     forall  $B' \in \mathcal{C}$  do ▷ Re-send all parents of B
16:       invoke gossip-broadcast ([BLK, B'])
17: upon gossip-deliver ([BLK, B]) s.t.  $B = (h, \bar{\text{tx}}, \pi, \sigma_i)$  do
18:   if  $\text{Verify}(p_j, h || \bar{\text{tx}} || \pi || s_{l_j}, \sigma_j) \wedge \exists \mathcal{C} \in \mathbb{C}$  s.t.  $H(\mathcal{C}[-1]) = h \wedge \mathbb{P}(\mathcal{C}, \bar{\text{tx}})$  then
19:      $st \leftarrow (\mathcal{C}, B)$ 
20:     invoke RA-validate( $p_i, st, \pi$ )
21:   else
22:     invoke gossip-broadcast([REQUEST, B]) ▷ Request for parents of B
23: upon RA-is-committed( $p_i, st, \text{TRUE}$ ) s.t.  $st = (\mathcal{C}, B)$  do
24:    $B \leftarrow \mathcal{B} \cup \{B\}$ 
25:   if  $|\mathcal{C}| > |\mathcal{C}_{\text{local}}|$  then
26:      $\mathcal{C}_{\text{local}} \leftarrow \mathcal{C}$  ▷ Update the local blockchain
27:      $\text{Extend}(\mathcal{C}_{\text{local}})$ 
28: upon RA-assign( $p_i, st, r, \pi$ ) s.t.  $st = (\mathcal{C}, B = (h, \bar{\text{tx}}, \pi, \perp)), \pi \neq \perp$  do
29:    $\sigma_i \leftarrow \text{Sign}(p_i, h || \bar{\text{tx}} || \pi)$ 
30:    $B \leftarrow (h, \bar{\text{tx}}, \pi, \sigma_i)$ 
31:   if  $r_i$  is burnable then
32:      $r_i \leftarrow r_i + r$ 
33:   invoke gossip-broadcast ([BLK, B])
34: upon RA-assign( $p_i, st, r, \pi$ ) s.t.  $\pi = \perp$  do
35:   if  $r$  is burnable then
36:      $r_i \leftarrow r$ 
37:    $\text{Extend}(\mathcal{C}_{\text{local}})$ 
38: upon a-deliver ([OP, tx]) do
39:    $\text{delivered} \leftarrow \text{delivered} \cup \{\text{tx}\}$ 
40: function  $\text{Extend}(\mathcal{C}_{\text{local}})$ 
41:   forall  $\text{tx} \in \mathcal{C}_{\text{local}}[: -k] \wedge \text{tx} \notin \text{delivered}$  do
42:     output a-deliver([OP, tx]) ▷ Deliver all transactions in the common prefix
43:      $\text{unordered} \leftarrow \text{unordered} \setminus \{\text{tx}\}$ 
44:    $h \leftarrow H(\mathcal{C}_{\text{local}}[-1])$ 
45:   select a list of transactions  $\bar{\text{tx}}$  from unordered such that  $\mathbb{P}(\mathcal{C}, \bar{\text{tx}}) = \text{TRUE}$ 
46:    $B_{\text{com}} \leftarrow (h, \bar{\text{tx}}, \perp, \perp)$ 
47:   invoke RA-commit( $p_i, (\mathcal{C}_{\text{local}}, B_{\text{com}}), r_i$ )
48:   if  $r$  is burnable then
49:      $r_i \leftarrow 0$ 

```

19:10 Modeling Resources

For a reusable resource, at any time step t , a resource budget r , a state st and upon potentially multiple repeated $RA\text{-commit}(p_i, st, r)$ from the same process p_i , if RA responds with $RA\text{-assigns}(p_i, st, r, \pi)$, then π is the same for every $RA\text{-commit}$ events output by RA .

Unforgeability No adversary can produce a resource commitment π such that π has not been previously $RA\text{-assigned}$ by RA and, upon $RA\text{-validate}(p_i, st, \pi)$, RA triggers $RA\text{-is-committed}(p_i, st, \text{TRUE})$, for some state st and some process p_i .

Honest-Majority Assignment At each time step, we denote with ϱ_H and ϱ_A the probabilities that at least one correct process and one Byzantine process, respectively, obtain a valid resource commitment for each $RA\text{-commit}$. More formally, for every time step t , we define:

$$\begin{aligned}\varrho_A &= \Pr[\exists RA\text{-assign}(p_i, st, r, \pi) \text{ such that } \pi \neq \perp \wedge p_i \in \mathcal{A}], \\ \varrho_H &= \Pr[\exists RA\text{-assign}(p_i, st, r, \pi) \text{ such that } \pi \neq \perp \wedge p_i \notin \mathcal{A}].\end{aligned}$$

Then we require that:

$$\varrho_A < \frac{1}{\Delta - 1 + 1/\varrho_H}. \quad (1)$$

The *liveness* property aims to capture the mining process in permissionless PoW blockchains and ensure that if processes keep committing resources, *eventually* one process will get assigned the resource commitment to extend the blockchain.

The *validity* property guarantees that a resource commitment can always be verified by any process p_i by triggering at any point $RA\text{-validate}$. This property captures the fact that any participant can efficiently verify, for example, the validity of the solution to the computational puzzle in PoW protocols or the evaluation of the verifiable random function in PoS protocols.

The *use-once* property prevents processes from increasing the probability of getting assigned the resource commitment either by committing several times, splitting the resource budget and then committing all the split amounts at different states or by committing a smaller resource budget. Intuitively, the *use-once* property also implies that the property holds for any integer partition of r (i.e., $r = \sum_{r_i > 0} r_i$). Moreover, the *use-once* property also implies that our model mainly focuses on probabilistic protocols as we do not aim to bypass the lower bound established in [21], namely, there is no deterministic protocol in *permissionless* setting that solves consensus. On the other hand, we believe that applying our model to *permissioned* blockchains with PoS, e.g., Tendermint [6], can be interesting future work.

The *unforgeability* property ensures that no process p_i can produce a valid resource commitment π that has not been previously $RA\text{-assigned}$ by the resource allocator.

Finally, the *honest-majority assignment* implies that despite the network delay, correct processes will have a higher probability of getting assigned the resource commitment at each time step. Equation (1) was established by Gaži *et al.* [18], and it takes into account that honest blocks may get discarded due to the network delay Δ .

Security Analysis. With the defined properties of a secure allocator, our model is equivalent to the idealized model introduced by Gaži *et al.* [18]. Therefore, their result also holds for our protocol, and we present them in our model as follows.

► **Lemma 4.2** ([18]). *Algorithm 1 implemented with a secure resource allocator RA_{res} satisfies the following properties:*

Safety *For any time steps t_1 and t_2 with $t_1 \leq t_2$, a common prefix parameter k and any local chain maintained by a correct process C_{local} , it holds that $C_{\text{local}}^{t_1}[: -k] \preceq C_{\text{local}}^{t_2}$ with an overwhelming probability.*

Liveness *For a parameter u and any time step t , let C_{local} be the local chain maintained by a correct process, then there is at least one new honest block in C^{t+u}/C^t with an overwhelming probability.*

Intuitively, *safety* implies that correct processes do not deliver different blocks at the same height, while *liveness* implies that every transaction is eventually delivered by all correct processes. Using Lemma 4.2 and properties of a secure resource allocator, we conclude the following.

► **Theorem 4.3.** *If RA_{res} is a secure resource allocator, then Algorithm 1 implements total-order broadcast.*

Proof. Observe that, since RA_{res} is a secure resource allocator, it satisfies *use-once* property. Therefore, Byzantine processes cannot amplify the probability $\varrho_{\mathcal{A}}$ by repeatedly triggering $RA\text{-commit}()$ on reusable resources at the same time step.

For the *validity* property, if a correct process p_i *a-broadcasts* a transaction tx (L9), tx is *gossip-broadcast* (L10) and, after Δ , every correct process *gossip-delivers* tx (L11) and adds it to *unordered* (L12). Eventually, transaction tx is selected by a correct process p_j as part of a block B (L45). Block B is then *gossip-broadcast* by p_j (L33) and eventually, after Δ , every correct process *gossip-delivers* B (L17), *validates* x (L18), and *validates* the resource commitment (L20). Observe that, because of the *unforgeability* property of RA_{res} , a valid resource commitment cannot be produced except by the resource allocator. Observe that this last step is possible through the *validity* property of RA_{res} . The proof then follows from Lemma 4.2.

No duplication property follows from the algorithm; if a correct process p_i *a-delivers* a transaction tx , p_i adds tx to *delivered* and condition at line L41 cannot be satisfied again.

For the *agreement* property, let us assume that a correct process p_i *a-delivered* a transaction tx buried at least k blocks deep in its adopted chain \mathcal{C} . Process p_i *a-delivers* a transaction tx when it *updates* the local blockchain with the longest chain \mathcal{C} (L26), tx has not been *a-delivered* yet and tx is part of the common prefix $\mathcal{C}[: -k]$ (L41). The property then follows from Lemma 4.2; eventually every correct process *a-delivers* transaction x , with an overwhelming probability.

Finally, for the *total order* property, from the *safety* property of Lemma 4.2, we know that correct processes do not deliver different blocks at the same height. This means that at a given height, if two correct processes p_i and p_j *a-delivered* a block, then this block is the same for p_i and p_j with an overwhelming probability. Moreover, since a block is identified by its hash, due to the collision-resistance property of $H(\cdot)$, it also implies that the set and order of transactions included in the block are the same for every correct process. So, if process p_i *a-delivers* transaction tx_1 before tx_2 , then either tx_1 and tx_2 are in the same block B with tx_1 appearing before tx_2 or they are in different blocks B_1 and B_2 such that B_2 appears in the chain after B_1 . The total-order property follows. ◀

4.2 Proof-of-Work Resource Allocator

In this part, we present the PoW allocator as a concrete instantiation of the resource allocator for *burnable* and *external* resources.

Proof-of-Work Resource Allocator. The PoW resource allocator RA_{pow} is parameterized by ϱ which is the default probability of getting assigned resource commitment for $r = 1$. RA_{pow} works as follows. Upon $RA\text{-commit}(p_i, st, r)$ by process p_i with a valid chain \mathcal{C} with respect to B_0 , RA_{pow} starts r concurrent threads of $Pow()$ function which acts as a biased coin with probability ϱ of assigning the resource commitment. Observe that, because computation is a *burnable* and *external* resource, processes cannot lie on about the *committed* resource budget r . In particular, Pow uniformly sample a value *nonce* in $\{0, 1\}^\lambda$ and either returns $\text{nonce} \in \{0, 1\}^\lambda$ or \perp . If *nonce* is the returned value in $\{0, 1\}^\lambda$, then RA_{pow} assigns it as the resource commitment to p_i , otherwise it $RA\text{-assigns}$ \perp to p_i . If the committed chain \mathcal{C} is not valid, then RA_{pow} $RA\text{-assigns}$ \perp to p_i . Validation of the resource commitment can be done by any process p_j through $RA\text{-validate}$; the resource allocator RA_{pow} returns either TRUE or FALSE, depending on the validity of the resource commitment. We implement the resource allocator RA_{pow} in Algorithm 2 and obtain the following lemma and theorem.

■ **Algorithm 2** Implementing PoW resource allocator, RA_{pow} .

```

50: state
51:    $B_0$ : Genesis block
52:    $\varrho$ : Default probability of getting assigned resource commitment on one resource
53: uses
54:   Random oracle:  $H : \{0, 1\}^* \rightarrow \{0, 1\}^\lambda$ 
55: upon  $RA\text{-commit}(p_i, st, r)$  s.t.  $st = (\mathcal{C}, B), B = (h, \bar{x}, \perp, \perp)$  do
56:   if  $\mathcal{C}$  is valid  $\wedge H(\mathcal{C}[-1]) = h$  then
57:     start  $r$  concurrent threads with  $\text{nonce}_j = Pow(p_i, st)$  for  $j \in \{0, \dots, r-1\}$ 
58:     wait for all  $r$  threads with  $Pow(p_i, st)$  for  $j \in \{0, \dots, r-1\}$  to finish
59:     if  $\exists \text{nonce}_j \neq \perp$  then
60:       output  $RA\text{-assign}(p_i, st, r, \text{nonce}_j)$ 
61:     else
62:       output  $RA\text{-assign}(p_i, st, r, \perp)$ 
63:   else
64:     output  $RA\text{-assign}(p_i, st, r, \perp)$ 
65: upon  $RA\text{-validate}(p_i, st, \pi)$  s.t.  $st = (\mathcal{C}, B), B = (h, \bar{x}, \pi, \sigma), \pi = \text{nonce}$  do
66:    $b \leftarrow H(h || \bar{x} || \text{nonce}) \stackrel{?}{\leq} \varrho \cdot 2^\lambda \wedge \mathcal{C}$  is valid  $\wedge H(\mathcal{C}[-1]) = h$ 
67:   output  $RA\text{-is-committed}(p_i, st, b)$ 
68: function  $Pow(p_i, st)$  ▷ With  $st = (\mathcal{C}, B)$  and  $B = (h, \bar{x}, \perp)$ 
69:    $\text{nonce} \xleftarrow{R} \{0, 1\}^\lambda$ 
70:   if  $H(h || \bar{x} || \text{nonce}) \leq \varrho \cdot 2^\lambda$  then
71:     return  $\text{nonce}$ 
72:   return  $\perp$ 

```

► **Lemma 4.4.** *Given the random oracle $H(\cdot)$, the default probability ϱ of getting assigned resource commitment on $r = 1$, and the network delay Δ , there exists a value $R_{\mathcal{A}}$ such that the resource allocator RA_{pow} implemented in Algorithm 2 is a secure resource allocator.*

Proof. *Liveness* property follows from Algorithm 2: upon $RA\text{-commit}(p_i, st, r)$ from process p_i , the resource allocator RA_{pow} either (i) has L56 satisfied and, eventually, outputs $RA\text{-assign}(p_i, st, r, \pi)$ with a resource commitment π to p_i or outputs $RA\text{-assign}(p_i, st, r, \perp)$ to p_i (L61) or (ii) if the chain is invalid (L63), and then the allocator outputs $RA\text{-assign}(p_i, st, r, \perp)$ to p_i .

For the *validity* property, observe that $\pi = \text{nonce}$ is valid if and only if there is a valid chain \mathcal{C} with respect to the genesis block B_0 such that the last block $B = \mathcal{C}[-1]$ contains π . Hence, π can be validated by any process p_j through $RA\text{-validate}(p_j, (\mathcal{C}, B^*), \pi)$; RA_{pow} then checks if $H(B^*) = h$ and $H(h||\bar{x}||\text{nonce}) \leq \varrho \cdot 2^\lambda$ outputting the same result at any process p_j .

The *use-once* property immediately follows because of the burnable property of the underlying resource (i.e., computation). Since RA_{pow} triggers $RA\text{-assigns}$ at the end of the the activation period for p_i , we claim that for multiple $RA\text{-commit}(p_i, \cdot, r_i)$ committed by p_i at t , it is equivalent to trigger $RA\text{-commit}(p_i, \cdot, r)$ once for $r = \text{Alloc}(p_i, t)$. In particular, at the time step t , let **Bad** be the event of not getting any resource commitment on all $RA\text{-commit}(p_i, \cdot, r_i)$ for $r_i \in \{r_1, r_2\}$ such that $r = r_1 + r_2$, then the probability of getting the resource commitment is $1 - \Pr[\text{Bad}] = 1 - (1 - 1 + (1 - \varrho)^{r_1})(1 - 1 + (1 - \varrho)^{r_2}) = 1 - (1 - \varrho)^r$.

Since the resource allocator RA_{pow} uses the random oracle H (i.e., idealized hash function with no exploitable weaknesses), the *unforgeability* property follows from the observation that, in order to produce a valid resource commitment, p_i has no better way to find the solution than trying many different queries to H . This implies that p_i has the same probability of obtaining a valid local resource commitment as it would have by $RA\text{-committing}$ to the resource allocator.

For the *honest-majority assignment* we recall that ϱ_H and ϱ_A are the probabilities that at least one correct process and one Byzantine process get the resource commitment after one $RA\text{-commit}$, respectively. In our model, it is not difficult to see that $\varrho_H = 1 - (1 - \varrho)^{R - R_A}$ and $\varrho_A = 1 - (1 - \varrho)^{R_A}$. Therefore, one can easily derive the amount of resource R_A such that $\varrho_A < \frac{1}{\Delta - 1 + 1/\varrho_H}$. ◀

► **Theorem 4.5.** *Algorithm 1 with the secure resource allocator RA_{pow} implements total-order broadcast.*

Proof. From Theorem 4.3 and Lemma 4.4, it follows that, since RA_{pow} is a secure resource allocator, then Algorithm 1 with RA_{pow} implements total-order broadcast. ◀

4.3 Proof-of-Stake Resource Allocator

The resource in PoS protocols is the stake of each process, and stake is a *virtual* and *reusable* resource. In those protocols, the probability of a process p_i being assigned a resource commitment is proportional to its stake in the system. In this part, we focus on Ouroboros Praos [9] for our formalization. Before presenting the PoS resource allocator as a concrete instance of a resource allocator for *reusable* and *virtual* resources, we need to introduce additional considerations and definitions.

Reusable Resources. By definition, a reusable resource allows processes to repeatedly trigger $RA\text{-commit}$ in the same time step using the same resource. Hence, if RA does not satisfy *use-once* property and assigns a resource commitment randomly, then every time a process p_i triggers $RA\text{-commit}$, process p_i might end up with a different result. For this reason, a naïve implementation of the resource allocator for reusable resources would allow an adversary to amplify the probability of getting assigned a resource commitment (i.e., ϱ_A) by repeatedly invoking $RA\text{-commit}$ using the same resource, i.e., in a grinding attack [3] at the same time step. Hence, for *probabilistic* blockchain protocols, to cope with this problem, one needs to ensure that the allocator satisfies *use-once* property. In particular, from designs of PoS protocols like Snow White [8] and Ouroboros Praos [9], three commonly used approaches to enforce *use-once* property are:

- (R1) **Explicit Time Slots** The first mechanism to enforcing *use-once* property is to index the resource by time slots. Protocols like Ouroboros Praos [9] and Snow White [8] require processes to have synchronized clocks to explicitly track time slots and epochs to ensure that each process derives a deterministic leader selection result from the same state.
- (R2) **Leader Selection from the Common Prefix** This mechanism requires correct processes to extract the set of potential leaders from the common prefix. In particular, the common prefix is a shortened local longest chain that is with overwhelming probability the same for all correct processes. This approach allows them to share the same view of potential leaders.
- (R3) **Deterministic and Trustworthy Source of Randomness** The source of randomness has to be trustworthy to ensure a fair leader election and to defend against an adaptive adversary that might corrupt processes predicted to be leaders for the upcoming time slots. In addition, the source of randomness has to be deterministic for each time slot and chain state in order to prevent the previously mentioned grinding attack. Hence, popular Proof-of-Stake blockchain protocols often rely on sophisticated protocols to produce randomness securely.

Slot and Epoch. An *epoch* e is a set of q adjacent time slot $S = \{sl_0, \dots, sl_{q-1}\}$. In practice, slot sl consists of a sufficient number of time steps so that discrepancies between processes' clocks are insignificant, and processes advance the slot at the same speed. In our model, we simplify this bookkeeping by requiring the allocator to maintain the slot and epoch.

Stake Distribution. The *stake distribution* at a time step t is $\mathbb{S}_{\text{stake}}^t = \{(p_1, r_1), \dots\}$ with $r_i \geq 0$, specifies the amount of stake owned by each process $p_i \in \mathcal{P}_{\leq t}$. We denote $\mathbb{S}_{\text{stake}}^e$ the stake distribution at the beginning of epoch e . The stake distribution $\mathbb{S}_{\text{stake}}^e$ can be obtained from $\text{StateAlloc}(\mathcal{C}[0 : sl], p_i)$ for $sl \leq e \cdot q$ for each $p_i \in \mathcal{P}_{\leq t}$.

► **Definition 4.6 (Leader Selection Process).** A leader selection process (\mathcal{D}, F) with respect to a stake distribution $\mathbb{S}_{\text{stake}} = \{(p_1, r_1), \dots\}$ is a pair consisting of a distribution \mathcal{D} and a deterministic function F . When $\rho \xleftarrow{R} \mathcal{D}$, for all $sl \in \mathbb{N}$, $F(\mathbb{S}_{\text{stake}}, sl; \rho)$ outputs process p_i with probability $1 - (1 - \rho)^{r_i}$ where ρ is the probability of assigning resource commitment for $r = 1$ for a given slot.

Proof-of-Stake Resource Allocator. The Proof-of-Stake resource allocator RA_{pos} with the leader selection process (\mathcal{D}, F) works as follows. First, we require that RA_{pos} keeps track of the current epoch and time slot to correctly assign the resource commitment to process p_i for the current slot. RA_{pos} keeps track of the slot through *Timeout* triggered by the *starttimer()* event. This approach is to enforce the first requirement of explicit time slots (R1). Secondly, upon $RA\text{-commit}(p_i, st, r)$ by process p_i with a valid state st in slot sl , RA_{pos} first checks if a random value $\rho \in \mathcal{D}$ for (p_i, st, sl) has been previously sampled; if so, then RA_{pos} picks it; otherwise, a fresh random value is sampled. This requirement ensures a deterministic and trustworthy source of randomness (R3). Then, RA_{pos} obtains the stake distribution of two epochs before, $\mathbb{S}_{\text{stake}}^{e-2}$ from st . This ensures a leader selection from the common prefix (R2). The resource allocator RA_{pos} uses $\mathbb{S}_{\text{stake}}^{e-2}$ together with the sampled randomness as input to the leader selection function F to check if p_i is selected for the slot sl . If this is the case, then RA_{pos} assigns the resource commitment to p_i , otherwise it assigns \perp . If the committed chain \mathcal{C} is not valid, then RA_{pos} assigns \perp to p_i . A validation of the resource commitment can be done by any process p_j through $RA\text{-validate}$; the resource allocator RA_{pos} returns

Algorithm 3 Implementing PoS Resource Allocator, RA_{pos} .

```

73: state
75:    $B_0$ : Genesis block
76:    $\mathcal{D}$ : Distribution
77:    $F$ : Leader selection function
78:    $sl$ : Current slot, initially  $sl = 0$ 
79:    $e$ : Current epoch, initially  $e = 0$ 
80:    $k$ : common prefix parameter
81:    $q$ : number of slots in an epoch, initially  $q = 16 \cdot k$ 
82:    $T$ : set of assigned resource commitments, initially empty
83: uses
84:   Random oracle:  $H : \{0, 1\}^* \rightarrow \{0, 1\}^\lambda$ 
85: upon  $RA\text{-commit}(p_i, st, r_i)$  s.t.  $st = (\mathcal{C}, B)$ ,  $B = (h, \bar{\text{tx}}, \perp, \perp)$  do
86:   if  $\mathcal{C}$  is valid  $\wedge H(\mathcal{C}[-1]) = h$  then
87:     obtain  $\mathcal{C}_{\text{prefix}}$  by pruning all blocks with slot  $> (e - 2) \cdot q$ , from  $\mathcal{C}$ 
88:     if  $\mathcal{C}_{\text{prefix}} = \emptyset$  do
89:        $\mathcal{C}_{\text{prefix}} \leftarrow [B_0]$ 
90:     if  $\exists (p_i, \mathcal{C}_{\text{prefix}}, \rho^*, sl) \in T$  then ▷ Queried before
91:        $\rho \leftarrow \rho^*$ 
92:     else
93:        $\rho \xleftarrow{R} \mathcal{D}$  ▷ Sample a fresh randomness
94:        $T \leftarrow T \cup \{(p_i, \mathcal{C}_{\text{prefix}}, \rho, sl)\}$  ▷ Update  $T$ 
95:       obtain the stake distribution  $\mathbb{S}_{\text{stake}}^{e-2}$  from  $\mathcal{C}_{\text{prefix}}$  ▷ Evaluate  $StateAlloc(\cdot, \cdot)$ 
96:        $p_j \leftarrow F(\mathbb{S}_{\text{stake}}^{e-2}, sl; \rho)$ 
97:       if  $p_i = p_j$  then
98:          $\pi \leftarrow (p_i, \rho, sl)$ 
99:         output  $RA\text{-assign}(p_i, st, \perp, \pi)$ 
100:      else
101:        output  $RA\text{-assign}(p_i, st, \perp, \perp)$ 
102:      else
103:        output  $RA\text{-assign}(p_i, st, \perp, \perp)$ 
104: upon  $RA\text{-validate}(p_i, st, \pi)$  s.t.  $st = (\mathcal{C}, B)$ ,  $B = (h, \bar{\text{tx}}, \pi, \sigma)$ ,  $\pi = (p_j, \rho, sl)$  do
105:   obtain  $\mathcal{C}_{\text{prefix}}$  by pruning all blocks with slot  $> (e - 2) \cdot q$ , from  $\mathcal{C}$ 
106:   if  $\mathcal{C}_{\text{prefix}} = \emptyset$  do
107:      $\mathcal{C}_{\text{prefix}} \leftarrow [B_0]$ 
108:   if  $\exists (p_i, \mathcal{C}_{\text{prefix}}, \rho, sl) \in T$  then ▷ Queried before
109:     obtain the stake distribution  $\mathbb{S}_{\text{stake}}^{e-2}$  from  $\mathcal{C}_{\text{prefix}}$ 
110:      $p_j^* \leftarrow F(\mathbb{S}_{\text{stake}}^{e-2}, sl; \rho)$ 
111:      $b \leftarrow p_i \stackrel{?}{=} p_j^* \wedge \mathcal{C}$  is valid  $\wedge H(\mathcal{C}[-1]) = h$ 
112:     output  $RA\text{-is-committed}(p_i, st, b)$ 
113:   else
114:     output  $RA\text{-is-committed}(p_i, st, \text{FALSE})$ 
115: upon  $Timeout$  do ▷ Increment slot
116:    $sl \leftarrow sl + 1$ 
117:   if  $sl \bmod q = 0$  then ▷ Increment epoch
118:      $e \leftarrow e + 1$ 
119:    $starttimer()$ 

```

either TRUE or FALSE, depending on the validity of the resource commitment. Finally, the PoS resource allocator is presented in Algorithm 3, and we conclude the following lemma and theorem.

► **Remark 4.7.** In practice, the randomness generation can be instantiated using verifiable random function [12], multiparty coin-tossing [20] protocol, or a random beacon [13]. However, Algorithm 3 aims to show the distinction between *external* and *virtual* resources.

► **Lemma 4.8.** *Given the random oracle $H(\cdot)$, the leader selection process (\mathcal{D}, F) parameterized by the default probability ϱ , and the network delay Δ , there exists a value R_A such that the resource allocator RA_{pos} implemented in Algorithm 3 is a secure resource allocator.*

Proof. *Liveness* property follows from the algorithm: upon $RA\text{-commit}(p_i, st, r)$ from process p_i , the resource allocator RA_{pos} either (i) has L86 satisfied and, eventually, outputs $RA\text{-assign}(p_i, st, \perp, \pi)$ with a resource commitment π to p_i , or outputs $RA\text{-assign}(p_i, st, \perp, \perp)$ to p_i (L100) or (ii) if the chain is invalid (L102), the allocator outputs $RA\text{-assign}(p_i, st, \perp, \perp)$ to p_i .

For the *validity* property, observe that $\pi = (p_i, \rho, sl)$ is valid if and only if p_i is a leader for sl . If p_i is a leader for sl , then in T there must be the random value ρ previously sampled for p_i (L94). This means that F evaluated on ρ will output again p_i . Hence, π can be validated by any process p_j through $RA\text{-validate}(p_j, st, \pi)$; RA_{pos} checks if $\pi \in T$ outputting the same result to p_j .

The *Use-once* property follows because, in our model, RA_{pos} keeps track of previous $RA\text{-commit}$ from p_i along with the time slots and states. Moreover, the choice of prob_i is stake-invariant, and it ensures that an adversary cannot increase its probability of being elected leader by dividing its stake into multiple identities. The proof for this is identical to the proof in Lemma 4.4. In practice, this property is enforced by the deterministic outputs of VRF and Hash function along with slot number and the common chain prefix as input.

The *unforgeability* property follows from the fact that any resource commitment produced by RA_{pos} is stored by the resource allocator in a set T of assigned resource commitments (L94). Hence, it is not possible for any process p_i to produce a valid resource commitment that is not in T . In practice, this property is guaranteed by the uniqueness property of verifiable random functions or the collision-resistant property of hash functions.

For the *honest-majority assignment* property, it is not difficult to see that we can derive ϱ_H and ϱ_A from R and R_A . In particular, $\varrho_H = 1 - (1 - \varrho)^{R-R_A}$ and $\varrho_A \approx 1 - (1 - \varrho)^{R_A}$. Here we note that the adversary can slightly increase ϱ_A by committing to shorter chains. However, it also means that the adversary will fall behind as it has to extend a much shorter chain than the current local chain maintained by correct processes, and we assume the adversary has no reason to do so. Hence, we consider $\varrho_A = 1 - (1 - \varrho)^{R_A}$. Thus, we can derive R_A so that $\varrho_A < \frac{1}{\Delta-1+1/\varrho_H}$. ◀

► **Theorem 4.9.** *Algorithm 1 with the secure resource allocator RA_{pos} implements total-order broadcast.*

Proof. From Theorem 4.3 and Lemma 4.8, it follows that, since RA_{pos} is a secure resource allocator, then Algorithm 1 with RA_{pos} implements total-order broadcast. ◀

5 Trade-offs Between Different Resources

In this section, we describe various attacks against the resource-based total-order broadcast. In particular, we demonstrate long-range attacks against *virtual* resources, and we discuss the incentive consideration that describes the cost of launching attacks against *burnable* and *reusable* resources.

5.1 Virtual Resource vs External Resource: Long-Range Attacks

Long-range Attacks on Virtual Resources. Long-range attacks [10] (LRAs), also sometimes called posterior-corruption attacks, can be mounted on any blockchain based on a virtual resource (such as PoS) if the majority of the set of active processes from an earlier slot becomes inactive in a later slot, as they no longer have any stake left in the system. Formally, they can be defined as follows:

► **Definition 5.1** (Virtual-Resource-Shifting Event). *A Virtual-Resource-Shifting Event happens when there exist two values h_0, h_1 , and a set of processes \mathcal{P}_{maj} such that:*

- **Active at h_0 :** *At height h_0 , processes in \mathcal{P}_{maj} control the majority of the total virtual resource (i.e., R), namely: $\sum_{p_i \in \mathcal{P}_{\text{maj}}} (\text{StateAlloc}(p_i, \mathcal{C}[0 : h_0])) > R - R_A$*
- **Inactive at h_1 :** *At height $h_1 > h_0$, processes in \mathcal{P}_{maj} control less virtual resources than the total number of resources controlled by the adversary (i.e., R_A): $\sum_{p_i \in \mathcal{P}_{\text{maj}}} (\text{StateAlloc}(p_i, \mathcal{C}[0 : h_1])) \leq R_A$*

If Definition 5.1 is satisfied, then most processes in \mathcal{P}_{maj} have released all or part of their resources by height h_1 , and the adversary has enough budget to corrupt all the active processes in \mathcal{P}_{maj} since they are all inactive in the present. The adversary could then use these processes to re-write the chain from $\mathcal{C}[h_0]$ since with a virtual resource as no external resource is needed to call the resource allocator. Furthermore, the *release of resource* from processes in \mathcal{P}_{maj} also happens on-chain, e.g., in the case of PoS for a process to move from active to inactive, it will spend its coins on-chain. An adversary re-writing the history of the chain could simply omit these transactions such that all processes satisfying definition 5.1 stay active in the alternative chain that the adversary is writing. The attack proceeds as follows:

1. When a virtual-resource-shifting event happens at the current height h_1 , \mathcal{A} corrupts all processes in \mathcal{P}_{maj} . Since the total of resources controlled by these processes is less than R_A , \mathcal{A} has enough budget to do so;
2. \mathcal{A} starts a new chain \mathcal{C}^* at $\mathcal{C}[h_0]$. At this height, \mathcal{A} controls the majority of the virtual resource, and because the resource allocator takes no further input apart from the state of $\mathcal{C}[0 : h_0]$, it assigns the resource commitment to \mathcal{A} with high probability;
3. \mathcal{A} now controls all processes in \mathcal{P}_{maj} and can alter the state of the chain such that the processes in \mathcal{P}_{maj} never release their resource;
4. The adversarial chain will grow at a faster rate and will *eventually* become longer than the honest chain because there is no network delay between corrupted processes.

Long-range Attacks on External Resources. The strategy above does not work with *external* resources. Even if Definition 5.1 holds, the adversary cannot call the resource allocator by simply corrupting the processes p_i as an *external* resources would be needed as input to the resource allocator (step 2 in the strategy above).

We formalize the implication of the long-range attack in the following lemma and theorem.

► **Lemma 5.2** (Long-range Attack). *In a virtual-resource-based total-order broadcast (Algorithm 1), let $\mathcal{C}_{\text{local}}$ be the longest chain maintained by a correct process, if a virtual-resource-shifting event occurs, then an adversary can eventually form a valid chain \mathcal{C}^* that is longer than $\mathcal{C}_{\text{local}}$.*

Proof. If a virtual-resource-shifting event (Definition 5.1) occurs, an adversary \mathcal{A} can corrupt all $p_i \in \mathcal{P}_{\text{maj}}$ at height h_1 . Notice that \mathcal{A} can do this because according to the threat model defined in definition 3.6, \mathcal{A} has enough resource budget to corrupt all $p_i \in \mathcal{P}_{\text{maj}}$.

Adversary \mathcal{A} can start a new chain \mathcal{C}^* at height h_0 by requiring all the corrupted processes $p_i \in \mathcal{P}_{\text{maj}}$ to commit old states to RA_{pos} . Since the Byzantine processes control the majority of the resources, the probability of Byzantine processes getting assigned commitment is strictly higher than the probability of correct processes getting assigned commitment; hence, the growth rate of \mathcal{C}^* is strictly higher than the growth rate of the honest chain $\mathcal{C}_{\text{local}}$; therefore, \mathcal{C}^* will eventually catch up and outgrow $\mathcal{C}_{\text{local}}$ in terms of the length.

More concretely, to simplify our analysis, we also assume the network delay to be 1 (i.e., $\Delta = 1$) between correct processes. We recall that ϱ_H is the probability that at least one correct process gets selected on the honest chain $\mathcal{C}_{\text{local}}$ at each time step. For any interval $[t_0, t_0 + t]$ and arbitrary $t_0, t \in \mathbb{N}$, we denote with X_0, \dots, X_{t-1} independent Poisson trails such that $\Pr[X_i = 1] = \varrho_H$, and we let $X_H = \sum_{i=0}^{t-1} X_i$. Using the Chernoff bound, one can show that for any $\epsilon \in (0, 1)$ it holds that $\Pr[X_H < (1 - \epsilon) \cdot \varrho_H \cdot t] \leq \exp(-\varrho_H \cdot t \cdot \epsilon^2/2)$. Intuitively, the Chernoff bound implies that the value of X_H cannot deviate too much from the mean; hence, for sufficiently large t and sufficiently small ϵ , the upper bound on the honest chain growth is approximately $\varrho_H \cdot t$, with an overwhelming probability.

Using the same argument for the growth of the malicious chain, one can show that for a sufficiently large time interval (i.e., t) and a sufficiently small ϵ , the lower bound of chain growth is approximately $\varrho_A \cdot t$ (i.e., $(1 + \epsilon) \cdot \varrho_A \cdot t$) with an overwhelming probability (i.e., $\exp(-\varrho_A \cdot t \cdot \epsilon^2/3)$), where ϱ_A is the probability that at least one Byzantine processes get selected on the honest chain \mathcal{C}^* at each time step.

So, if $\varrho_A > \varrho_H$, we can claim that \mathcal{C}^* grows at a faster rate than $\mathcal{C}_{\text{local}}$. This is the case for Algorithm 1 that uses RA_{pos} allocator. Due to Definition 5.1, the probability of getting assigned the resource commitment with \mathcal{C}^* , is $\varrho_A > 1 - (1 - \varrho)^{R-R_A} = \varrho_H$, where ϱ_H is the probability that at least one correct processes get assigned a resource commitment with $\mathcal{C}_{\text{local}}$. ◀

► **Remark 5.3.** Also, if we assume a $\Delta > 1$ network delay between correct processes, there will a non-zero probability that a fork can happen, and honest blocks can get discarded due to the network delay. On the other hand, we also assume a perfect synchrony ($\Delta = 1$) between Byzantine processes; therefore, there is no loss in the malicious growth rate. Therefore, even when correct processes and Byzantine processes control the same amount of resources on both chains, due to network delay, the chain growth rate of \mathcal{C}^* can still be higher than the chain growth rate of the honest chain $\mathcal{C}_{\text{local}}$.

► **Theorem 5.4.** *If a virtual-resource-shifting event occurs, a total-order broadcast based on virtual resources (Algorithm 1) does not implement total-order broadcast.*

Proof. Let \mathcal{C} be the honest chain adopted by every correct process and let us assume that all the transactions buried at least k blocks deep in \mathcal{C} have been *a-delivered* (Algorithm 1, L42) by every correct process. If a virtual-resource-shifting event occurs then, by Lemma 5.2, an adversary can eventually form a valid chain \mathcal{C}^* that is longer than \mathcal{C} .

For existing processes, the adversary can send this \mathcal{C}^* to a subset of correct processes. This implies that some correct processes will adopt \mathcal{C}^* as a valid chain; they will *a-deliver* all the transactions buried at least k blocks deep in \mathcal{C}^* . This implies that, eventually, the *total-order* property is violated. Also, due to the permissionless nature of our model, correct processes might join the system at any time. Hence, new processes will adopt the malicious chain as the local chain; therefore, *delivered* will be different among correct processes. Hence, the *total-order* property is violated. ◀

5.2 Incentives in Burnable and Reusable Resources

One of the vulnerabilities induced by *reusable* resources is that extending the blockchain is *costless* with respect to the resource considered. This is different from *burnable* resources, where creating a block consumes the resource; this consumption is captured in our model as the interaction between processes and the resource allocator. The use of reusable resources can result in two different types of adversarial behaviors. The first one consists in creating multiple blocks at the same time slots on different chains. The second one consists in keeping blocks created private from the rest of the processes. In both cases, we discuss how this *costless* property associated with block creation for longest-chain consensus protocols based on a reusable resource impacts their security compared to those based on burnable resource. In this section, we assume, as is traditional with any blockchain system, that some financial reward is associated with block creation, and we assume the cost of acquiring resources is the same for both reusable and burnable resources. With these assumptions and the use-once property of resource allocator, we define the chain extension cost as follows.

► **Definition 5.5** (Chain Extension Cost). *The cost of extending a valid chain for a process p_i between two time steps t_1 and t_2 such that $t_1 \leq t_2$ is defined to be the resource budgets committed and assigned back during this time interval. In particular, we have:*

- *For burnable resources: $Cost_{\text{burn}}(p_i, t_1, t_2) = \sum_{t=t_1}^{t_2} Alloc(p_i, t)$*
- *For reusable resources: $Cost_{\text{reuse}}(p_i, t_1, t_2) \leq \max_{t \in [t_1, \dots, t_2]} \{Alloc(p_i, t)\}$*

► **Proposition 5.6.** *For all time step $t_2 > t_1$ and a process p_i , the cost of extending a valid chain with a burnable resource is strictly more expensive than with a reusable resource, i.e., $Cost_{\text{burn}}(p_i, t_1, t_2) > Cost_{\text{reuse}}(p_i, t_1, t_2)$.*

Proposition 5.6 indicates that it is inherently more expensive to extend the blockchain for *burnable* resources; hence, it is more difficult to launch different types of attacks on blockchain based on *burnable* resources. In the following, we explain different types of attacks.

Private Attack. The private attack [11], sometimes called double-spending attack, is the most simple attack in longest-chain blockchains. The adversary creates a private chain, i.e., it mines on its own without broadcasting its blocks to the other processes and without accepting the blocks from other processes. In particular, the adversary runs Algorithm 1, except that it does not broadcast its blocks until the end of the attack. This means that two chains grow in parallel: the adversarial one, that only the adversary is aware of, and the honest one. The adversary is aware of the honest chain but chooses not to contribute to it and it wins the attack if it creates a chain longer than the honest chain. In the case of a *burnable* resource, this attack has a cost as every block created consumes a resource. If the adversary wins the attack, then the cost is recovered as the adversary wins the reward associated with block creation. Otherwise, it loses the cost associated with all the resources consumed. In the case of a *reusable* resource, the only cost of the attack is the *opportunity cost*, i.e., the adversary takes the risk of potentially not earning the rewards associated with block creation if the attack fails but does not lose any resources. The attack in this case is then much cheaper than in the case of a burnable resource. The cost of a private attack is higher if the resource allocator is based on a burnable resource than if it is on a reusable resource, thus creating a stronger disincentivisation for an adversary. The results follow from the fact that for a reusable resource, the resource allocator can be invoked on the same resource several times. From Proposition 5.6, it is not difficult to see that the expected return on performing a private attack is higher for a *reusable* resources as the probability of winning the attack (i.e. producing a longer chain) is the same in both cases, but the cost is higher for a *burnable* resource.

Resource-bleeding Attack. Stake-bleeding attacks [17] were proposed in the context of PoS blockchains and work, informally, as follows. An adversary starts creating a private chain (i.e., it does not broadcast its blocks to the rest of the network) but, differently from the private attack described previously, the adversary may continue creating blocks on the honest chain. In its private chain, the adversary includes all of the transactions it is aware of, harvesting the associated transaction fees. Furthermore, the adversary also receives the coinbase reward usually associated with block production. After a sufficient amount of time, the adversary will have bloated its amount of resources and will *eventually* be able to create a chain that becomes as long as the honest chain. This attack could be extended to the general-resource case, which we call this attack *resource-bleeding attack*, and note that in the case of an external resource, this attack is much easier to detect than in the Proof-of-Stake case. In order to understand this attack, we must extend the model from Section 4 to take into account *total resource adjustments* in the case of inactive processes. In Section 6, we describe the general case of resource-bleeding attacks and discuss how they are more detectable on an external resource and the most mitigated for burnable resources.

Nothing-at-Stake Attack. In a Nothing-at-Stake attack, instead of deciding to extend the longest chain (Algorithm 1, L25), a process decides to mine simultaneously on all of the chains it is aware of. In the case of a burnable resource, an adversary cannot reuse the same resource to mine on multiple chains (due to L49), hence in order to mount this attack, the adversary must decide how to commit its resources to multiple chains. In contrast, with a reusable resource, each resource can be fully committed to each chain. If there exists multiple forks of the same length, there is a risk that a process will mine on a chain that ends up being abandoned and thus will miss out on the associated reward. It thus becomes rational for a process to deviate from the protocol and mine on every chain since this reduces chance of losing reward because network may select different chain. If every process adopts this strategy, the protocol cannot achieve the common prefix property as every chain will keep on growing at the same pace.

6 Discussion

Resource-bleeding Attack in the Flexible Resource Setting. The resource-bleeding attack stems from this observation: in order to deal with inactive processes, if the protocol wants to maintain its block production rate, it needs to adjust its leader selection processes such that inactive processes are not selected anymore. In practice, this means increasing ϱ such that every active process has a higher chance of being selected and removing the inactive processes from the list of eligible block producers and hence maintaining a steady block rate. If an adversary starts a private attack, since no resource commitment from the other processes is included in the adversarial chain, after a sufficient time, ϱ will be updated to ensure that the adversarial chain block rate is maintained. On the other hand, with a reusable resource, the adversary could keep maintaining its resources on the honest chain to ensure that the leader selection probability is not adjusted on the honest chain. After enough time, all the honest processes will be removed from the power table in the adversarial chain. This means that when electing a leader on the adversarial chain, the adversary now represents the full power table and is guaranteed to be elected at each epoch. On the other hand, since the adversary maintain its resource on the honest chain, without contributing as many blocks as it could. This means that, after some time, the honest chain will grow at a slower rate than the adversarial chain and the adversary will be able to create a chain as long as the honest chain, breaking the safety of the protocol.

In practice in Bitcoin, the *target* value [26] is updated every two weeks (roughly) to ensure that blocks are created, on average, at the same pace. An adversary could fork the chain, wait for the difficulty adjustment to adjust and then be able to create a chain at the same pace as the honest chain. This is, however, easily detectable. In the PoW case, one can simply see that the difficulty has been adjusted and that one chain has much fewer resources than the other. Moreover, since the resource is burnable, it is not possible for an adversary to continue mining on the honest chain as the same burnable resource cannot be used twice, hence the adversary cannot maintain its full resource on the honest chain and the honest chain difficulty must be adapted accordingly.

For an external, but reusable, resource such as storage, the adversary could maintain its power in both chain, however, it is easy to detect the adversarial chain as it will have fewer resources committed to it and hence is distinguishable from the honest chain.

Mitigations against Different Attacks. In the following, we discuss various mitigations against attacks described in Section 5.

Long-range Attacks. In practice, many PoS systems deal with long-range attacks by using some form of checkpointing [1, 24, 23], requiring key-evolving cryptography [9, 19], or using multiple types of resources [15]. Others use more refined chain selection rules [9, 2] (i.e., chain density analysis or selecting the longest chain that fork less than k blocks) instead of the longest chain selection.

Resource-bleeding Attacks. In the case of PoS, mitigation has been proposed in Ouroboros Genesis [2] and it works as follows. When a process is presented with two forks, it differentiates between two cases. In the first case, the fork is smaller than the common prefix parameter k , i.e., the two chains differ for a number of slots smaller than k , in which case the usual longest-chain rule is applied. If on the other hand, the forks differ from more than k slots, then the processes look at the first k slots after the fork (i.e., the first k slots where the two chains diverge) and choose the chain with the most blocks in that period. Intuitively, this is because during the beginning of the fork, an adversary has not had the time to bloat its stake and hence the rate at which its chain grows will be smaller than that of the correct processes. In the case of an external resource, it suffices to look at the total power (which can be explicit in the case of a reusable resource, or implicit for a burnable resource, e.g., *target* value) at the tip (end) of the chains and pick the one with the most resource.

Nothing-at-stake Attacks. A process that performs a nothing-at-stake attack with a reusable resource is easily detectable as anyone can see that the same resource was used on different chains. One typical mitigation adopted by PoS systems is to *slash*, i.e., financially punish, processes who use their resource on concurrent chains. This is usually done by having processes deposit some money before gaining participation rights, and then burning some of this deposit if a proof of misbehavior is sent to the blockchain. The details of this mechanism are out of scope for this paper.

7 Conclusion

Resources are essential in ensuring the safety property of total-order broadcast protocols in a permissionless setting as it protects the protocol from Sybil attacks. However, there exist several attacks on protocols based on reusable and virtual resources that a formal specification would help understand and address.

In this work, we formalize properties of resources through a *resource allocator* abstraction, and identify crucial properties on how to make this resource allocator secure for blockchain protocols. Using a secure resource allocator, we demonstrate how to construct a generic

longest-chain total-order broadcast algorithm. Furthermore, we also illustrate how certain types of resources tend to make blockchain protocols more vulnerable to different types of attacks. We believe that this formalization will help blockchain protocol designers to select suitable types of resources for their protocols and understand and analyze the potential security trade-offs on those resources.

Outlook. For future work, we find the following research directions worth investigating:

- **Relaxed Assumptions.** Our analysis works with a setting where the total amount of active resources is known and fixed. Hence, it is natural to extend this model to a setting where the total amount of resources is unknown and potentially fluctuates.
- **Different Network Setting and Participation Models.** Our model focuses on *probabilistic longest-chain* protocols in a Δ -synchrony setting. However, we believe that our model can be applied to analyze properties of resource-based *deterministic* protocols in a permissioned and partially synchrony setting such as Tendermint [6] and HotStuff [27].
- **Different Types of Resources.** Finally, there are other resource-based protocols such as the Proof-of-Elapsed-Time (PoET) protocol [4] or multi-resources-based protocol [15] that have not been considered in this work. Hence, one can extend this model to analyze those protocols.

References

- 1 Sarah Azouvi, George Danezis, and Valeria Nikolaenko. Winkle: Foiling long-range attacks in proof-of-stake systems. In *AFT*, pages 189–201. ACM, 2020.
- 2 Christian Badertscher, Peter Gazi, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In *CCS*, pages 913–930. ACM, 2018.
- 3 Joseph Bonneau, Jeremy Clark, and Steven Goldfeder. On bitcoin as a public randomness source. *IACR Cryptol. ePrint Arch.*, page 1015, 2015.
- 4 Mic Bowman, Debajyoti Das, Avradip Mandal, and Hart Montgomery. On elapsed time consensus protocols. In *INDOCRYPT*, volume 13143 of *Lecture Notes in Computer Science*, pages 559–583. Springer, 2021.
- 5 Jonah Brown-Cohen, Arvind Narayanan, Alexandros Psomas, and S. Matthew Weinberg. Formal barriers to longest-chain proof-of-stake protocols. In *EC*, pages 459–473. ACM, 2019.
- 6 Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. *CoRR*, abs/1807.04938, 2018. [arXiv:1807.04938](https://arxiv.org/abs/1807.04938).
- 7 Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.
- 8 Phil Daian, Rafael Pass, and Elaine Shi. Snow white: Robustly reconfigurable consensus and applications to provably secure proof of stake. In *Financial Cryptography*, volume 11598 of *Lecture Notes in Computer Science*, pages 23–41. Springer, 2019.
- 9 Bernardo David, Peter Gazi, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In *EUROCRYPT (2)*, volume 10821 of *Lecture Notes in Computer Science*, pages 66–98. Springer, 2018.
- 10 Evangelos Deirmentzoglou, Georgios Papakyriakopoulos, and Constantinos Patsakis. A survey on long-range attacks for proof of stake protocols. *IEEE Access*, 7:28712–28725, 2019.
- 11 Amir Dembo, Sreeram Kannan, Ertem Nusret Tas, David Tse, Pramod Viswanath, Xuechao Wang, and Ofer Zeitouni. Everything is a race and nakamoto always wins. In *CCS*, pages 859–878. ACM, 2020.
- 12 Yevgeniy Dodis and Aleksandr Yampolskiy. A verifiable random function with short proofs and keys. In *Public Key Cryptography*, volume 3386 of *Lecture Notes in Computer Science*, pages 416–431. Springer, 2005.

- 13 drand: Distributed randomness beacon. URL: <https://drand.love/>.
- 14 Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.
- 15 Matthias Fitzi, Xuechao Wang, Sreeram Kannan, Aggelos Kiayias, Nikos Leonardos, Pramod Viswanath, and Gerui Wang. Minotaur: Multi-resource blockchain consensus. In *CCS*, pages 1095–1108. ACM, 2022.
- 16 Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *EUROCRYPT (2)*, volume 9057 of *Lecture Notes in Computer Science*, pages 281–310. Springer, 2015.
- 17 Peter Gazi, Aggelos Kiayias, and Alexander Russell. Stake-bleeding attacks on proof-of-stake blockchains. In *CVCBT*, pages 85–92. IEEE, 2018.
- 18 Peter Gazi, Aggelos Kiayias, and Alexander Russell. Tight consistency bounds for bitcoin. In *CCS*, pages 819–838. ACM, 2020.
- 19 Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nikolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *SOSP*, pages 51–68. ACM, 2017.
- 20 Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *CRYPTO (1)*, volume 10401 of *Lecture Notes in Computer Science*, pages 357–388. Springer, 2017.
- 21 Andrew Lewis-Pye. Byzantine generals in the permissionless setting. *CoRR*, abs/2101.07095, 2021. [arXiv:2101.07095](https://arxiv.org/abs/2101.07095).
- 22 Ling Ren. Analysis of nakamoto consensus. *IACR Cryptol. ePrint Arch.*, page 943, 2019.
- 23 Selma Steinhoff, Chrysoula Stathakopoulou, Matej Pavlovic, and Marko Vukolic. BMS: secure decentralized reconfiguration for blockchain and BFT systems. *CoRR*, abs/2109.03913, 2021. [arXiv:2109.03913](https://arxiv.org/abs/2109.03913).
- 24 Ertem Nusret Tas, David Tse, Fangyu Gai, Sreeram Kannan, Mohammad Ali Maddah-Ali, and Fisher Yu. Bitcoin-enhanced proof-of-stake security: Possibilities and impossibilities. *IACR Cryptol. ePrint Arch.*, page 932, 2022.
- 25 Benjamin Turner. Permissionless consensus in the resource model. In *Financial Cryptography*, volume 13411 of *Lecture Notes in Computer Science*, pages 577–593. Springer, 2022.
- 26 Bitcoin Wiki. Target. URL: <https://en.bitcoin.it/wiki/Target>.
- 27 Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan-Gueta, and Ittai Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In *PODC*, pages 347–356. ACM, 2019.

Computing Power of Hybrid Models in Synchronous Networks

Pierre Fraigniaud ✉

IRIF, Université Paris Cité and CNRS, France

Pedro Montealegre ✉

Faculty of Engineering and Science, Adolfo Ibáñez University, Santiago, Chile

Pablo Paredes ✉

Department of Mathematical Engineering, University of Chile, Santiago, Chile

Ivan Rapaport ✉

DIM-CMM (UMI 2807 CNRS), University of Chile, Santiago, Chile

Martín Ríos-Wilson ✉

Faculty of Engineering and Science, Adolfo Ibáñez University, Santiago, Chile

Ioan Todinca ✉

LIFO, Université d'Orléans and INSA Centre-Val de Loire, France

Abstract

During the last two decades, a small set of distributed computing models for networks have emerged, among which LOCAL, CONGEST, and Broadcast Congested Clique (BCC) play a prominent role. We consider *hybrid* models resulting from combining these three models. That is, we analyze the computing power of models allowing to, say, perform a constant number of rounds of CONGEST, then a constant number of rounds of LOCAL, then a constant number of rounds of BCC, possibly repeating this figure a constant number of times. We specifically focus on 2-round models, and we establish the complete picture of the relative powers of these models. That is, for every pair of such models, we determine whether one is (strictly) stronger than the other, or whether the two models are incomparable. The separation results are obtained by approaching communication complexity through an original angle, which may be of an independent interest. The two players are not bounded to compute the value of a binary function, but the *combined* outputs of the two players are constrained by this value. In particular, we introduce the XOR-Index problem, in which Alice is given a binary vector $x \in \{0, 1\}^n$ together with an index $i \in [n]$, Bob is given a binary vector $y \in \{0, 1\}^n$ together with an index $j \in [n]$, and, after a single round of 2-way communication, Alice must output a boolean out_A , and Bob must output a boolean out_B , such that $\text{out}_A \wedge \text{out}_B = x_j \oplus y_i$. We show that the communication complexity of XOR-Index is $\Omega(n)$ bits.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases hybrid model, synchronous networks, LOCAL, CONGEST, Broadcast Congested Clique

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2022.20

Related Version *Full Version*: <https://arxiv.org/abs/2208.02640>

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-PAI77170068.

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.

Martín Ríos-Wilson: Additional support from ANID-FONDECYT Postdoctorado 3220205.



© Pierre Fraigniaud, Pedro Montealegre, Pablo Paredes, Ivan Rapaport, Martín Ríos-Wilson, and Ioan Todinca;

licensed under Creative Commons License CC-BY 4.0

26th International Conference on Principles of Distributed Systems (OPODIS 2022).

Editors: Eshcar Hillel, Roberto Palmieri, and Etienne Rivière; Article No. 20; pp. 20:1–20:18



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

This paper analyzes the relative power of distributed computing models for networks, all resulting from the combination of standard synchronous models such as LOCAL and CONGEST [47], as well as Broadcast Congested Clique (BCC) [21]. Each of these three models has its strengths and limitations. In particular, CONGEST assumes the ability for each node to send a specific message to each of its neighbors at every round (even in a clique). However, the communication links have limited bandwidth. Specifically, at most $O(\log n)$ bits can be sent through any link during a round, in n -node networks. LOCAL assumes a link with unlimited bandwidth between any two neighboring nodes, but the information acquired by any node u after $t \geq 0$ rounds of communication is limited to the data available at nodes at distance at most t from u in the network. Finally, BCC supports all-to-all communications between the nodes, and thus does not suffer from the locality constraint of LOCAL and CONGEST. However, at each round, each node is bounded to send a *same* $O(\log n)$ -bit message to all the other nodes. In this paper, we investigate the power of models resulting from combining these three models, in order to take advantage of their positive aspects without suffering from their negative ones.

For the sake of comparing models, we focus on the standard framework of distributed *decision* problems on labeled graphs (see [27]). Such problems are defined by a collection \mathcal{L} of pairs (G, ℓ) , where $G = (V, E)$ is a graph, and $\ell : V \rightarrow \{0, 1\}^*$ is a function assigning a label $\ell(u) \in \{0, 1\}^*$ to every $u \in V$. Such a set \mathcal{L} is called a distributed *language*. For instance, deciding whether a certain set U of nodes in a graph G forms a vertex cover can be modeled by the language

$$\text{vertex-cover} = \{(G, \ell) : \forall \{u, v\} \in E(G), \ell(u) = 1 \vee \ell(v) = 1\},$$

by labeling 1 all the vertices in U , and 0 all the other vertices. Similarly, deciding C_4 -freeness can be modeled by the language $C_4\text{-freeness} = \{(G, \ell) : C_4 \not\preceq G\}$, where $H \preceq G$ denotes that H is a subgraph of G , and deciding whether a graph is planar can be captured by the language $\text{planarity} = \{(G, \ell) : G \text{ is planar}\}$. A distributed algorithm A *decides* \mathcal{L} if every node running A eventually accepts or rejects, and the following condition is satisfied: for every labeled graph (G, ℓ) ,

$$(G, \ell) \in \mathcal{L} \iff \text{all nodes accept.}$$

That is, every node should accept in a yes-instance (i.e., an instance $(G, \ell) \in \mathcal{L}$), and, in a no-instance (i.e., an instance $(G, \ell) \notin \mathcal{L}$), at least one node must reject.

For every $t \geq 0$, let us denote by \mathbf{L}^t the set of distributed languages \mathcal{L} for which there is a t -round algorithm in the LOCAL model deciding \mathcal{L} , with $\mathbf{L} = \mathbf{L}^1$. The sets \mathbf{C}^t and \mathbf{B}^t are defined similarly, for the CONGEST and BCC models, respectively. Note that while it is easy to show, using indistinguishability arguments, that, for every $t \geq 1$, $\mathbf{L}^t \setminus \mathbf{L}^{t-1} \neq \emptyset$ and $\mathbf{C}^t \setminus \mathbf{C}^{t-1} \neq \emptyset$, establishing that there is indeed a decision problem in $\mathbf{B}^t \setminus \mathbf{B}^{t-1}$ requires significantly more work [46]. Also, we define $\mathbf{L}^* = \cup_{t \geq 0} \mathbf{L}^t$, $\mathbf{C}^* = \cup_{t \geq 0} \mathbf{C}^t$, and $\mathbf{B}^* = \cup_{t \geq 0} \mathbf{B}^t$. So, in particular, \mathbf{L}^* is the class of distributed languages that can be decided in a constant number of rounds in the LOCAL model.

The three models under consideration, i.e., LOCAL, CONGEST, and BCC exhibit very different behaviors with respect to decision problems. For instance, it is known [22] that

$$C_4\text{-freeness} \in \mathbf{L} \setminus (\mathbf{B}^* \cup \mathbf{C}^*),$$

whenever one assumes, as we do in this paper, that, for all models under consideration, every node is initially aware of the identifiers¹ of its neighbors. On the other hand, it is also known [12] that

$$\text{planarity} \in \mathbf{B} \setminus \mathbf{L}^*.$$

This means that while no LOCAL algorithms can decide planarity in a constant number of rounds, there is a 1-round BCC algorithm deciding planarity, and while no BCC algorithms can decide C_4 -freeness in a constant number of rounds, there is a 1-round LOCAL algorithm deciding C_4 -freeness. So, if one allows LOCAL algorithms to do just a single round of all-to-all communication, as in BCC, then both C_4 -freeness and planarity can be solved in a constant number of rounds, hence increasing the computational power of LOCAL dramatically.

This observation led us to investigate scenarios such as the case in which the CONGEST model is enhanced by allowing nodes to perform few rounds in either LOCAL, or BCC. What would be the computing power of such a *hybrid* model? For answering this question, for a collection of non-negative integers $\alpha_1, \dots, \alpha_k$, β_1, \dots, β_k , and $\gamma_1, \dots, \gamma_k$, we define the set

$$\prod_{i=1}^k \mathbf{L}^{\alpha_i} \mathbf{B}^{\beta_i} \mathbf{C}^{\gamma_i}$$

as the class of decision languages \mathcal{L} which can be decided by an algorithm performing $\alpha_1 \geq 0$ rounds of LOCAL, followed by $\beta_1 \geq 0$ rounds of BCC, followed by $\gamma_1 \geq 0$ rounds of CONGEST, followed by $\alpha_2 \geq 0$ rounds of LOCAL, etc., up to $\gamma_k \geq 0$ rounds of CONGEST. For instance, we have

$$\{\text{planarity}, C_4\text{-freeness}\} \subseteq \mathbf{LB} \cap \mathbf{BL}.$$

However, how do \mathbf{LB} and \mathbf{BL} compare? And what about \mathbf{CB} vs. \mathbf{BC} , and \mathbf{LC} vs. \mathbf{CL} ? These are the kinds of questions that we are studying in this paper. In the long-term perspective, this line of research is motivated by the following question. Let \mathcal{L} be a fixed distributed language, and let us assume that a round of LOCAL costs a (say, for acquiring high-throughput channels), that a round of BCC costs b (say, for benefiting of facilities supporting all-to-all communications), and that a round of CONGEST costs c . The goal is to minimize the total cost of an algorithm deciding \mathcal{L} in a constant number of rounds, that is, to solve the following minimization problem:

$$\prod_{i=1}^k \mathbf{L}^{\alpha_i} \mathbf{B}^{\beta_i} \mathbf{C}^{\gamma_i} \ni \mathcal{L} \left(a \sum_{i=1}^k \alpha_i + b \sum_{i=1}^k \beta_i + c \sum_{i=1}^k \gamma_i \right). \quad (1)$$

Note that, for $a = b = c = 1$, Eq. (1) corresponds to minimizing the number of rounds for deciding \mathcal{L} when using a combination of the communication facilities provided by LOCAL, CONGEST, and BCC. For instance, deciding whether a graph is C_k -free can be achieved in $\lfloor \frac{k}{2} \rfloor$ rounds in LOCAL, that is, $C_k\text{-freeness} \in \mathbf{L}^{\lfloor k/2 \rfloor}$. Eq. (1) is asking whether deciding $C_k\text{-freeness}$ could be achieved at a lower cost by combining LOCAL, CONGEST, and BCC. For tackling Eq. (1), we need a better understanding of the fundamental effects resulting from combining these models.

¹ In each of the models, every node u of a n -node network $G = (V, E)$ is supposed to be provided with an identifier $\text{id}(u)$, where $\text{id} : V \rightarrow [1, N]$ is one-to-one, and $N(n) = \text{poly}(n)$, i.e., all identifiers can be stored on $O(\log n)$ bits in n -node networks. We also assume that all nodes are initially aware of the size n of the network, merely because this is the case in model BCC.

1.1 Our Results

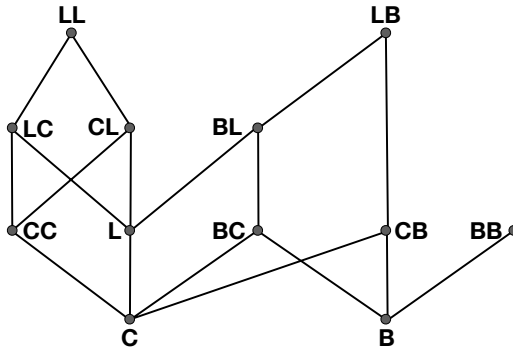
On the negative side, we provide a series of separation results between 2-round hybrid models. In particular, we show that **BC** and **CB** are incomparable. That is, there are languages in $\mathbf{BC} \setminus \mathbf{CB}$, and languages in $\mathbf{CB} \setminus \mathbf{BC}$. In fact, we show stronger separation results, by establishing that $\mathbf{BC} \setminus \mathbf{C}^*\mathbf{B} \neq \emptyset$, and $\mathbf{CB} \setminus \mathbf{BL}^* \neq \emptyset$. That is, in particular, there are languages that can be decided by a 2-round algorithm performing a single BCC round followed by one CONGEST round, which cannot be decided by any algorithm performing k CONGEST rounds followed by a single BCC round, for any $k \geq 1$.

On the positive side, we show that, for any non-negative integers $\alpha_1, \dots, \alpha_k, \beta_1, \dots, \beta_k$,

$$\prod_{i=1}^k \mathbf{L}^{\alpha_i} \mathbf{B}^{\beta_i} \subseteq \mathbf{L}^{\sum_{i=1}^k \alpha_i} \mathbf{B}^{\sum_{i=1}^k \beta_i}. \tag{2}$$

That is, if a language \mathcal{L} can be decided by a t -round algorithm alternating LOCAL and BCC rounds, then \mathcal{L} can be decided by a t -round algorithm performing all its LOCAL rounds first, and then all its BCC rounds – with the notations of Eq. (2), $t = \sum_{i=1}^k (\alpha_i + \beta_i)$. So, in particular $\mathbf{BL} \subseteq \mathbf{LB}$. This inclusion is strict, since, as said before, $\mathbf{CB} \setminus \mathbf{BL}^* \neq \emptyset$. In fact, this separation holds even if the number of LOCAL rounds depends on the number of nodes n in the network, as long as the algorithm performs $o(n)$ LOCAL rounds after its BCC round. Another consequence of Eq. (2) is that the largest class of languages among all the ones considered in this paper is $\mathbf{L}^*\mathbf{B}^*$, that is, languages that can be decided by algorithms performing k LOCAL rounds followed by k' BCC rounds, for some $k \geq 0$ and $k' \geq 0$. Thus, Eq. (1) should be studied for languages $\mathcal{L} \in \mathbf{L}^*\mathbf{B}^*$.

Interestingly, our separation results hold even for randomized protocols, which can err with probability at most $\epsilon \leq 1/5$. That is, in particular, there is a language $\mathcal{L} \in \mathbf{CB}$ (i.e., that can be decided by a 2-round algorithm performing a single BCC round followed by one CONGEST round) that cannot be decided by any algorithm performing k CONGEST rounds followed by a single BCC round, for any $k \geq 1$.



■ **Figure 1** The poset of 2-round hybrid models. An edge between a set of languages \mathbf{S}_1 and a set \mathbf{S}_2 , where \mathbf{S}_1 is at a level lower than \mathbf{S}_2 , indicates that $\mathbf{S}_1 \subseteq \mathbf{S}_2$. In fact, all inclusions are strict. Transitive edges are not displayed. Two sets that are not connected by a monotone path are incomparable. For instance, **CB** and **BL** are incomparable, while $\mathbf{BC} \subseteq \mathbf{LB}$.

Our Techniques. All our separation results are obtained by reductions from communication complexity lower bounds. However, we had to revisit several known communication complexity results for adapting them to the setting of distributed decision, in which no-instances may

be rejected by a single node, and not necessarily by all the nodes. In particular, we revisit the classical **Index** problem. Recall that, in this problem, Alice is given a binary vector $x \in \{0, 1\}^n$, Bob is given an index $i \in [n]$, and Bob must output x_i based on a single message received from Alice (1-way communication). We define the **XOR-Index** problem, in which Alice is given a binary vector $x \in \{0, 1\}^n$ together with an index $i \in [n]$, Bob is given a binary vector $y \in \{0, 1\}^n$ together with an index $j \in [n]$, and, after a single round of 2-way communication, Alice must output a boolean out_A and Bob must output a boolean out_B , such that

$$\text{out}_A \wedge \text{out}_B = x_j \oplus y_i.$$

That is, if $x_j \neq y_i$ then Alice and Bob must both accept (i.e., output *true*), and if $x_j = y_i$ then *at least* one of these two players must reject (i.e., output *false*). We show that the sum of the sizes of the message sent by Alice to Bob and the message sent by Bob to Alice is $\Omega(n)$ bits. This bound holds even if the communication protocol is randomized and may err with probability at most $1/5$, and even if the two players have access to shared random coins.

The fact that only one of the two players may reject a no-instance (i.e., an instance where $x_j \oplus y_i = 0$), and not necessarily both, while a yes-instance must be accepted by both players, yields an asymmetry which complicates the analysis. We use information theoretic tools for establishing our lower bound. Specifically, we identify a way to decorrelate the behaviors of Alice and Bob, so that to analyze separately the distribution of decisions taken by each player, and then to recombine them for lower bounding the probability of error in case the messages exchanged between the players are small, contradicting the fact that this error probability is supposed to be small. Roughly, given messages m_A and m_B exchanged by the two players, and given two indices i and j , we compute the value y_i maximizing the error probability for Alice, and the value x_j maximizing the error probability for Bob, conditioned to m_A, m_B, i, j . We then show that the combined pair (x_j, y_i) provide a sufficiently good lower bound on the probability of error for the whole protocol, which contradicts the fact that the error must be at most ϵ .

1.2 Related Work

The **LOCAL** model was introduced in [42] at the beginning of the 1990s, when the celebrated $\Omega(\log^* n)$ lower bound on the number of rounds for computing a 3-coloring or a maximal independent set (MIS) in the n -node cycle was proved. A few years later, the class of *locally checkable labeling* (LCL) problems was introduced and studied in [45]. This class essentially corresponds to the class \mathbf{L}^* , but restricted to graphs with constant maximum degrees. Given $\mathcal{L} \in \mathbf{L}^*$, and the family \mathcal{G}_Δ of graphs with maximum degree at most Δ , solving the LCL problem induced by \mathcal{L} and \mathcal{G}_Δ consists of designing a distributed algorithm which, given a graph $G \in \mathcal{G}_\Delta$, computes a labeling ℓ of the nodes such that $(G, \ell) \in \mathcal{L}$. It is known that many LCL problems can be solved in constant number of rounds in **LOCAL**. This is for instance the case of certain types of weak colorings problems [45]. Also, there is an $O(\sqrt{k}\Delta^{1/\sqrt{k}} \log \Delta)$ -approximation algorithm for minimum dominating set running in $O(k)$ rounds [38] (where $k \geq 1$ is a parameter), and there is an $O(n^\epsilon)$ -approximation algorithm for the minimum coloring problem running in $\exp(O(1/\epsilon))$ rounds [11]. In fact, it is undecidable, in general, whether a given LCL problem has a (construction) algorithm running in a constant number of rounds [45]. A plethora of papers have addressed graph problems in the **LOCAL** model, and we refer to the survey [52], but several significant results have been obtained since then, among which it is worth mentioning two fields in close connection to the topic of this paper, which emerged in the early 2010s. One is the systematic study of distributed

decision problems in various settings, including non-determinism [28, 31, 37] and interactive protocols [36, 44]. The other is a systematic study of the round-complexity of LCL problems (see, e.g., [9, 53], and the references therein).

The CONGEST model is a weaker variant of the LOCAL model in which the size of the messages exchanged at each round between neighbors is bounded to $O(\log n)$ bits, or B bits in the parametrized version of the model. This bound on the message size creates bottlenecks limiting the power of algorithms under this model. A fruitful line of research has established several non-trivial lower bounds on the round-complexity of CONGEST algorithms, by reduction from communication complexity problems (see for instance [1, 5, 24, 48, 50]). Nevertheless, several problems can still be solved in a constant number of rounds in CONGEST. This is for instance the case of computing a $(2 + \varepsilon)$ -approximation of minimum vertex cover which can be done in $O(\log \Delta / \log \log \Delta)$ rounds [10] in graphs with maximum degree Δ . Also, *testing* (a weaker variant of decision, a la property-testing) the presence of specific subgraphs like small cliques or short cycles can be done in a constant number of rounds in CONGEST (see, e.g., [14, 25, 29, 30, 41]).

The congested clique model [21, 43] has first been introduced in its *unicast* version (UCC), where every node is allowed to send potentially different $O(\log n)$ -bit messages to each of the other $n - 1$ nodes at every round. In the UCC model, many natural problems can be solved in a constant number of rounds [17, 35, 40]. The UCC model is very powerful, and it has actually been proved [21] that it can simulate powerful bounded-depth circuits classes, from which it follows that exhibiting non-trivial lower bounds for the UCC model is quite difficult. The *broadcast* variant of the congested clique, namely the BCC model, is significantly weaker than the unicast variant, and lower bounds on the round-complexity of problems in the BCC model have been established, again by reduction to communication complexity problems. This is the case of problems such as detecting the presence of particular subgraphs [21], detecting planted cliques [18], or approximating the diameter of the network [33]. Obviously, many fast, non-trivial BCC-algorithms have also been devised. As examples, we can mention the sub-logarithmic deterministic algorithm that finds a maximal spanning forest in $O(\log n / \log \log n)$ rounds [34], and algorithms for deciding and reconstructing several graph families (including bounded degeneracy graphs) performing in a constant number of rounds [13]. It is worth noticing that, for single round algorithms, the BCC model is also referred to using other terminologies, such as *simultaneous-messages* [8], or *sketches* [2, 54]. In these latter models though, the measure of complexity is the size of the messages, and therefore the restriction to $O(\log n)$ -bits messages is not enforced.

Hybrid distributed computing models have been investigated in the literature only recently, motivated by the various forms of modern communication technologies, from high-throughput optical links to global wireless communication facilities, to peer-to-peer long-distance logical connections. In particular, a hybrid model allowing nodes to perform in a *local* mode, and in a *global* mode at each round has been recently considered [7]. The local mode corresponds to perform a LOCAL round [47], while the global mode corresponds to perform a *node-capacitated clique* (NCC) round [6], which allows each node to exchange $O(\log n)$ -bit messages with $O(\log n)$ arbitrary nodes in the network. It is shown that, in the LOCAL+NCC hybrid model, SSSP can be approximated in $\tilde{O}(n^{1/3})$ rounds, and APSP can be approximated in $\tilde{O}(\sqrt{n})$ rounds. Several lower bounds are also presented in [7], including an $\tilde{\Omega}(\sqrt{n})$ -round lower bound for computing APSP, and an $\Omega(n^{1/3})$ -round lower bound for computing the diameter. In a subsequent work [39], it was shown that APSP can actually be solved exactly in $\tilde{O}(\sqrt{n})$ rounds in the LOCAL+NCC model. Some of these results were further improved in [15, 16] where it is shown how to solve multiple SSSP problems exactly in

$\tilde{O}(n^{1/3})$ rounds, and how to approximate SSSP in $\tilde{O}(n^{5/17})$ rounds. Other graph problems, such as spanning tree, maximal independent set (MIS) construction, and routing were also considered in the LOCAL+NCC model (see [20, 32]). In fact, it was very recently shown [3] that any problem on sparse graphs can be solved in $\tilde{O}(\sqrt{n})$ rounds in the LOCAL+NCC model. Efficient distributed algorithms for general graphs in this model can then be obtained using sparsification techniques. Finally, it is worth pointing out that the weaker hybrid model CONGEST+NCC was considered in [26] for restricted families of graphs.

As a final remark, it is interesting to notice that the XOR-Index problem is related to the EPR paradox [23], and especially the so-called CHSH game [19] whose objective is to demonstrate the existence of quantum (non-classical) correlations in physics (see [4]).

2 Hybrid Models Based on LOCAL and BCC

In this section, we consider the combination of LOCAL and BCC, and, in particular, we compare the two classed **LB** and **BL**. The section can be considered as a warmup section before stating more complex separation results further in the text.

First, we establish a general result concerning the hybridation of LOCAL and BCC. Recall that $\prod_{i=1}^k \mathbf{L}^{\alpha_i} \mathbf{B}^{\beta_i}$ is the class of distributed decision problems that can be decided by an algorithm performing α_1 rounds of LOCAL, then β_1 rounds of BCC, then α_2 rounds of LOCAL, etc., ending with β_k rounds of BCC. We show that every language in this class can be computed in the same number of rounds by performing first all LOCAL rounds, and then all BCC rounds.

► **Theorem 1.** *Let $k \geq 1$ be an integer, and let $\alpha_1, \dots, \alpha_k$ and β_1, \dots, β_k be non-negative integers. We have $\prod_{i=1}^k \mathbf{L}^{\alpha_i} \mathbf{B}^{\beta_i} \subseteq \mathbf{L}^{\sum_{i=1}^k \alpha_i} \mathbf{B}^{\sum_{i=1}^k \beta_i}$.*

Proof. Let $\mathcal{L} \in \prod_{i=1}^k \mathbf{L}^{\alpha_i} \mathbf{B}^{\beta_i}$, and let A be a distributed algorithm deciding \mathcal{L} in the corresponding hybrid model combining LOCAL and BCC. Let us consider the maximum integer $t < \sum_{i=1}^k (\alpha_i + \beta_i)$ such that A performs BCC at round t , and LOCAL at round $t + 1$. (If no such t exist, then A is already in the desired form.) We transform A into A' performing the same as A , excepted that rounds t and $t + 1$ are switched. Specifically, let us consider a run of A for an instance (G, ℓ) . Let B_u be the message broadcasted by u at round t of A , and, for every neighbor v of u , let $L_{u,v}$ be the message sent by u to v at round $t + 1$ of A . To define A' , let S_u be the state of every node u at the beginning of round t of A , and let $N_G(u)$ be the set of neighbors of u in G . In A' , every node u sends its state S_u to all its neighbors at round t , using LOCAL. At round $t + 1$ of A' , every node u broadcasts B_u to all nodes, using BCC (this is doable, as u was able to produce B_u based on S_u at round t). Finally, before completing round $t + 1$, every node u uses the collection $\{S_v : v \in N_G(u)\}$ and the collection $\{B_w : w \in V(G)\}$ to compute the messages $L_{v,u}$ for all $v \in N_G(u)$, by simulating what every such neighbor would have done v at round t of A . Indeed, $L_{v,u}$ depend solely on S_v and $\{B_w : w \in V(G)\}$. (We make the standard assumption that all nodes are running the same algorithm, but even if that was not the case, every node could also send the code of its algorithm to all its neighbors together with its state at round t .) It follows that, at the end of round $t + 1$ of A' , every node u can compute its state after $t + 1$ rounds of A . By repeating the same switch operation until no LOCAL rounds occur after a BCC round, we eventually obtain an algorithm deciding \mathcal{L} and establishing that $\mathcal{L} \in \mathbf{L}^{\sum_{i=1}^k \alpha_i} \mathbf{B}^{\sum_{i=1}^k \beta_i}$. ◀

► **Corollary 2.** $\mathbf{BL} \subsetneq \mathbf{LB}$.

Proof. The fact that $\mathbf{BL} \subseteq \mathbf{LB}$ is a direct consequence of Theorem 1. On the other hand, there is a distributed language in $\mathbf{LB} \setminus \mathbf{BL}$ since, as shown by Theorem 5, $\mathbf{CB} \setminus \mathbf{BL}^* \neq \emptyset$. ◀

We now show a separation between the class \mathbf{BL} and the class $\mathbf{B}^* \cup \mathbf{L}^*$ of languages that can be decided in a constant number of rounds either in **BCC** or **LOCAL**. The proof does not use communication complexity reduction, but a mere reduction to **triangle-freeness**.

► **Theorem 3.** $\mathbf{BL} \setminus (\mathbf{B}^* \cup \mathbf{L}^*) \neq \emptyset$

Proof. Let us consider the language **triangle-on-max-degree-freeness** (**TOMDF**) defined by the set of graphs G such that, for every triangle T in G , all nodes in T have a degree smaller than the maximum degree of G . Note that $\mathbf{TOMDF} \in \mathbf{BL}$. Indeed, during the **BCC** round, every node can broadcast its degree. Thus, during the **LOCAL** round, each node can learn all triangles it belongs to. Every node rejects if it is of maximum degree, and it is contained in a triangle. Otherwise, it accepts. Moreover, $\mathbf{TOMDF} \notin \mathbf{L}^*$ because, for every $k \geq 0$, in k **LOCAL** rounds a node cannot distinguish an instance G in which it has maximum degree from an instance G' in which there is a node with a larger degree. It remains to show that $\mathbf{TOMDF} \notin \mathbf{B}^*$.

Let us assume, for the purpose of contradiction, that there exists $k \geq 0$, such that **TOMDF** can be decided by an algorithm \mathcal{A} performing k **BCC** rounds, i.e., $\mathbf{TOMDF} \in \mathbf{B}^k$. We can use \mathcal{A} to decide **triangle-freeness** in $k + 1$ **BCC** rounds. Let G be a graph. In the first **BCC** round, every node v broadcasts its identifier $\text{id}(v)$ and its degree $d(v)$, and hence learns the maximum degree Δ of G . Then every node simulates \mathcal{A} on the virtual graph G' on $\frac{n\Delta}{2}$ nodes obtained from G by adding a set S_v of $\Delta - d(v)$ pending vertices to each vertex v of G . Every node v simulates \mathcal{A} in G' by simulating its execution on v and on all the nodes in S_v . Specifically, after the first **BCC** round, v knows the set of IDs used in G , and thus the rank of its ID in this set. Therefore, it can compute the set I composed of the smallest $\frac{n\Delta}{2} - n$ positive integers that are not used as IDs in G . Furthermore, it can assign IDs to its $\Delta - d(v)$ pending virtual neighbors in G' , using its rank and the degrees of all the nodes with lower rank in G , so that (1) the ID of each virtual node is unique in G' , and (2) every node of G knows the IDs assigned to the pending virtual neighbors of every other node in G . It follows that each node v does not need to simulate the messages broadcasted in \mathcal{A} by the nodes in S_v . In fact, every node v can simulate the behavior of all the virtual nodes in $S = \cup_{u \in V(G)} S_u$ at each round of \mathcal{A} . As a consequence, the simulation of \mathcal{A} in G' does not yield any overhead on the number of bits to be broadcasted by each (real) node v running \mathcal{A} . After the k **BCC** rounds of \mathcal{A} in G' have been simulated, every node v accepts (on G) if itself and all the nodes in S_v accept in \mathcal{A} on G' . Now, by construction, $G' \in \mathbf{TOMDF}$ if and only if G is triangle-free. Since \mathcal{A} decides **TOMDF**, we get that **triangle-freeness** $\in \mathbf{B}^{k+1}$, a contradiction. ◀

3 Hybrid Models Based on **BCC** and **CONGEST**

In this section, we consider the combination of **CONGEST** and **BCC**, and, in particular, we compare the two classes \mathbf{CB} and \mathbf{BC} . The separation of these two classes uses the communication complexity problem **XOR-Index**. In the next section, we will establish that the 2-ways 1-round communication complexity of **XOR-Index** is $\Omega(n)$ bits. We use this lower bounds in the proofs of this section.

We first show that not only $\mathbf{CB} \setminus \mathbf{BC} \neq \emptyset$ but also $\mathbf{CB} \setminus \mathbf{BL}^* \neq \emptyset$.

► **Theorem 4.** $\mathbf{CB} \setminus \mathbf{BL}^* \neq \emptyset$. This result holds even for randomized algorithms performing one BCC round followed by a constant number of LOCAL rounds, which may err with probability ϵ , for every $\epsilon < 1/5$.

Proof. Let us consider the distributed language denoted **one-marked-edge** defined as

$$\text{one-marked-edge} = \left\{ (G, \ell) : (\ell : V(G) \rightarrow \{0, 1\}) \right. \\ \left. \wedge (|\{\{u, v\} \in E(G) : \ell(u) = \ell(v) = 1\}| = 1) \right\}.$$

In words, the language corresponds to the graphs G with a potential mark on each node, satisfying that exactly one edge of G has its two endpoints marked. We have $\text{one-marked-edge} \in \mathbf{CB}$. Indeed, a simple algorithm consists, for each node, to learn which of its neighbors are marked, in one CONGEST round, and to broadcast its number of marked incident edges, in one BCC round. The nodes reject if the total sum of marked edges is different from 2 (i.e., exactly two nodes are incident to a unique marked edge). They accept otherwise.

We now prove that $\text{one-marked-edge} \notin \mathbf{BL}^*$. We show that this result holds even for a randomized algorithm which may err with probability $\epsilon < 1/5$. For the purpose of contradiction, let us assume that, for some $k \geq 0$, there exists an ϵ -error algorithm \mathcal{A} solving **one-marked-edge** using one BCC round followed by k consecutive LOCAL rounds. We show how to use \mathcal{A} for designing an ϵ -error 1-round protocol Π solving **XOR-index** by communicating only $\mathcal{O}(\sqrt{m})$ bits on m -bit instances, contradicting the fact that **XOR-index** has communication complexity $\Omega(m)$.

Let $(x, i) \in \{0, 1\}^m \times [m]$ and $(y, j) \in \{0, 1\}^m \times [m]$ be an instance of **XOR-index**. Without loss of generality, we assume that $m = \binom{n}{2}$ for some $n \in \mathbb{N}$. Let us consider a graph G on $2n + 4k$ nodes, composed of two disjoint copies of a clique of size n , plus a path P of $4k$ nodes. Let us denote by G^A and G^B the two cliques. The IDs assigned to the nodes of G^A are picked in $[1, n]$, while the IDs assigned to the nodes of G^B are picked in $[n + 1, 2n]$. One extremity of P is connected to all nodes in G^A , and the other extremity of P is connected to all nodes in G^B . Let us denote by P^A the $2k$ nodes of P closest to G^A , and by P^B the $2k$ nodes of P closest to G^B . These nodes are assigned IDs $2n + 1, \dots, 2n + 4k$, consecutively, starting from the extremity of P connected to G^A .

We enumerate the $m = \binom{n}{2}$ edges in G^A and G^B from 1 to m . Then, in Π , the players interpret their input vectors x and y as indicators of the edges of G^A and G^B respectively. We denote by G_{xy} the subgraph of G such that, for every $r \in [m]$, the r -th edge e of G^A (resp., G^B) is in G_{xy} if and only if $x_r = 1$ (resp., $y_r = 1$). Also, all edges incident to nodes of P are in G_{xy} . Let $\{u_A^i, v_A^i\}$ be the endpoints of the i -th edge of G_A , and let $\{u_B^j, v_B^j\}$ represent the endpoints of the j -th edge of G_B . (These edges may or may not be in G_{xy} depending on the values of x_j and y_i .) We define $\ell_{ij} : V(G) \rightarrow \{0, 1\}$ as the marking function such that $\ell_{ij}(w) = 1$ if and only if $w \in \{u_A^i, v_A^i, u_B^j, v_B^j\}$. By construction, we have that $(G_{xy}, \ell_{ij}) \in \text{one-marked-edge}$ if and only if $((x, i), (y, j))$ is a yes-instance of **XOR-index**, i.e., $x_j \neq y_i$. We say that Alice owns all nodes in $V(G^A) \cup V(P^A)$, and Bob owns all nodes in $V(G^B) \cup V(P^B)$. Observe that the edges of G_{xy} incident to nodes owned by Alice depend only on x , while the edges of G_{xy} incident to nodes owned by Bob only depend on y .

We are now ready to describe Π . First, Alice and Bob simulate the BCC round of algorithm \mathcal{A} on all the nodes of G_{xy} owned by them, respectively, considering that *no vertices are marked*. This simulation results in each player constructing a set of $n + 2k$ messages, one for each node of the clique owned by the player, plus one message for each of the $2k$ nodes in the sub-path owned by the player. We denote by M_0^A and M_0^B the set of messages

produced by Alice and Bob, respectively. Next, the players repeat the same procedure, but considering now that *all vertices are marked*, from which it results sets of messages denoted by M_1^A and M_1^B , respectively. Finally, Alice sends the pair (M_0^A, M_1^A) to Bob, as well as her input index i . Similarly, Bob sends the pair (M_0^B, M_1^B) to Alice, as well as his input index j . Observe that the size of these messages is $\mathcal{O}((n+k)\log n)$ bits.

After the communication, Alice and Bob decide their outputs as follows. First, each player extracts from M_1^A the messages produced by u_A^j and v_A^j , and extract from M_1^B the messages produced by u_B^i and v_B^i . Then, they extract from M_0^A and M_0^B the messages of every other node. Let us call M the resulting set of messages. Observe that M corresponds exactly to the set of messages communicated during the BCC round of \mathcal{A} on input (G_{xy}, ℓ_{ij}) . Then, Alice and Bob simulate the k LOCAL rounds of \mathcal{A} on all the vertices they own. This is possible as the nodes of P are not marked, for every instance of XOR-index. Each player accepts if all the nodes owned by this player accept. Since $(G_{xy}, \ell_{ij}) \in \text{one-marked-edge}$ if and only if $((x, i), (y, j))$ is a yes-instance of XOR-index, we get that Π is an ϵ -error protocol solving XOR-index on inputs of size m by communicating only $\mathcal{O}((n+k)\log n) = \mathcal{O}(\sqrt{m})$ bits, which is a contradiction with Theorem 7. \blacktriangleleft

We now show that $\mathbf{BC} \setminus \mathbf{CB} \neq \emptyset$.

\blacktriangleright **Theorem 5.** $\mathbf{BC} \setminus \mathbf{CB} \neq \emptyset$. *This result holds even for randomized algorithms performing one CONGEST round followed by one BCC round, which may err with probability ϵ , for every $\epsilon < 1/5$.*

Proof. For every $n \geq 2$, let us consider the path P_{2n+1} , i.e., the path with $2n+1$ nodes, denoted consecutively $a_1, \dots, a_n, c, b_n, \dots, b_1$. Let $x \in \{0, 1\}^n$, $y \in \{0, 1\}^n$, $i \in [n]$, and $j \in [n]$. We define the labeling $\ell_{x,y,i,j}$ of the nodes of P_n as follows:

$$\ell_{x,y,i,j}(a_1) = i, \quad \ell_{x,y,i,j}(a_n) = x, \quad \ell_{x,y,i,j}(b_n) = y, \quad \ell_{x,y,i,j}(b_1) = j,$$

and, for every $v \notin \{a_1, a_n, b_1, b_n\}$, $\ell_{x,y,i,j}(v) = \perp$. We define the distributed language

$$\text{XOR-index-path} = \{(P_{2n+1}, \ell_{x,y,i,j}) : (n \geq 2) \wedge (x, y \in \{0, 1\}^n) \wedge (i, j \in [n]) \wedge (x_j \neq y_i)\}.$$

First, we show that $\text{XOR-index-path} \in \mathbf{BC}$. During the BCC round, every node broadcasts its ID, and the IDs of its neighbors (a node with more than two neighbors simply rejects). Also, degree-1 nodes broadcasts their labels. Note that the $2n+1$ nodes can then check whether they are vertices of the path P_{2n+1} , and, if this is not the case, they reject. Let i and j be the labels broadcasted by the two extremities of the path. Based on the information broadcasted by all the nodes, each of the two nodes a_n and b_n adjacent to the middle node c of the path knows which of the two labels i or j correspond to the index broadcasted by its farthest extremity in the path, b_1 and a_1 , respectively. Thus, during the CONGEST round, a_n and b_n can send the bits x_j and y_i to the center c of the path, which checks whether $x_j \neq y_i$, and accepts or rejects accordingly.

Now, we show that $\text{XOR-index-path} \notin \mathbf{CB}$. Let us assume for the purpose of contradiction that there exists a 2-round algorithm \mathcal{A} deciding XOR-index-path by performing one CONGEST round followed by one BCC round. To solve an instance $((x, i), (y, j))$ of XOR-Index, Alice and Bob simulate \mathcal{A} on the path P_{2n+1} with consecutive IDs $1, \dots, 2n+1$. Specifically, Alice simulates the $n+1$ nodes a_1, \dots, a_n, c , while Bob simulates the $n+1$ nodes b_1, \dots, b_n, c , with the nodes labeled with $\ell_{x,y,i,j}$. For simulating the CONGEST round, Alice sends to Bob the message m_{a_n} sent from a_n to c during that round, and Bob sends to Alice the message m_{b_n} sent from b_n to c during that round. The BCC round is actually simulated

simultaneously. More precisely, Alice and Bob can both construct the messages broadcasted by all nodes a_3, \dots, a_{n-2} and b_3, \dots, b_{n-2} , merely because they know their IDs and their labels (equal to \perp), and they can therefore infer the messages these nodes receive during the CONGEST round. So, these messages do not need to be communicated between the players. Moreover, Alice knows a priori what messages m'_{a_1}, m'_{a_2} , and m'_{a_n} are to be broadcasted by a_1, a_2 and a_n during the BCC round, and can send them to Bob. Symmetrically, Bob knows a priori what messages m'_{b_1}, m'_{b_2} , and m'_{b_n} are to be broadcasted by b_1, b_2 and b_n during the BCC round, and can send them to Alice. As for node c , thanks to the messages m_{a_n} and m_{b_n} sent by Alice to Bob, and by Bob to Alice, respectively, both players can construct the message to be sent by c during the BCC round. So, in total, for simulating \mathcal{A} , Alice (resp., Bob) just needs to send the messages $m_{a_n}, m'_{a_1}, m'_{a_2}, m'_{a_n}$ to Bob (resp., the messages $m_{b_n}, m'_{b_1}, m'_{b_2}, m'_{b_n}$ to Alice), which consumes $O(\log n)$ bits of communication in total. Each player accepts if all the nodes he or she simulates accept, and rejects otherwise. Alice and Bob are thus able to solve XOR-index by exchanging $O(\log n)$ bits only, which contradicts Theorem 7. \blacktriangleleft

As a direct consequence of the previous two theorems, we get:

► **Corollary 6.** *The sets CB and BC are incomparable.*

4 The Communication Complexity of XOR-index

This section is dedicated to the analysis of the following communication problem.

XOR-index:

Input: Alice receives $x \in \{0, 1\}^n$ and $i \in [n]$; Bob receives $y \in \{0, 1\}^n$ and $j \in [n]$.

Task: Alice outputs a boolean out_A and Bob outputs a boolean out_B such that $\text{out}_A \wedge \text{out}_B = x_i \oplus y_i$.

We focus on 2-way 1-round protocols, that is, each player sends only one message to the other player, both players send their messages simultaneously, and each player must decide his or her output upon reception of the message sent by the other player. For every 2-player communication problem P , and for every $\epsilon > 0$, let us denote by $CC^1(P, \epsilon)$ the communication complexity of the best 2-way 1-round randomized protocol solving P with error probability at most ϵ .

► **Theorem 7.** *For every non-negative $\epsilon < 1/5$, $CC^1(\text{XOR-index}, \epsilon) = \Omega(n)$ bits.*

The rest of the section is entirely dedicated to the proof of Theorem 7. Let $0 \leq \epsilon < 1/5$, and let Π randomized protocol solving XOR-index with error probability at most ϵ , where Alice communicates k_A bits to Bob, and Bob communicates k_B bits to Alice. Without loss of generality, we can assume that, in Π , Alice (resp., Bob) sends explicitly the value of i (resp., j) to Bob (resp., Alice). Indeed, this merely increases the communication complexity of Π by an additive factor $O(\log n)$, which has no consequence, as we shall show that $k_A + k_B = \Omega(n)$.

Let us consider the probabilistic distribution over the inputs of Alice and Bob, where x and y are drawn uniformly at random from $\{0, 1\}^n$, and i and j are drawn uniformly at random from $[n]$. Let us denote X and I the random variables equal to the inputs of Alice, and Y and J the random variables equal to the inputs of Bob. Let M_A (resp., M_B) be the random variable equal to the message sent by Alice (resp., Bob) in Π on input (X, I) (resp., (Y, J)). Note that M_A and M_B have values in $\Omega_A = \{0, 1\}^{k_A}$ and $\Omega_B = \{0, 1\}^{k_B}$, respectively, of respective size 2^{k_A} and 2^{k_B} .

20:12 Computing Power of Hybrid Models in Synchronous Networks

Let us fix $i, j \in [n]$, $m_A \in \Omega_A$, and $m_B \in \Omega_B$. Let $\mathcal{E}_{m_A, j}^A$ be the event corresponding to Bob receiving $J = j$ as input, and Alice sending $M_A = m_A$ to Bob in the communication round. Similarly, let $\mathcal{E}_{m_B, i}^B$ be the event corresponding to Alice receiving $I = i$ as input, and Bob sending $M_B = m_B$ to Alice in the communication round. For $a, b \in \{0, 1\}$, we set:

$$p(a, m_A, j) = \Pr[X_J = a \mid \mathcal{E}_{m_A, j}^A], \text{ and } q(b, m_B, i) = \Pr[Y_I = b \mid \mathcal{E}_{m_B, i}^B],$$

and

$$p(a, j) = \Pr[X_J = a \mid J = j], \text{ and } q(b, i) = \Pr[Y_I = b \mid I = i].$$

Observe that $p(a, j) = q(b, j) = 1/2$. Let a^* and b^* be the most probable values of X_j given (m_A, j) , and of Y_i given (m_B, i) , respectively. Formally,

$$a^* = \operatorname{argmax}_{a \in \{0, 1\}} p(a, m_A, j), \text{ and } b^* = \operatorname{argmax}_{b \in \{0, 1\}} q(b, m_B, i).$$

Observe that $p(a^*, m_A, j) \geq 1/2$ and $q(b^*, m_B, i) \geq 1/2$. We first establish the following technical lemma.

► **Lemma 8.** *Let \mathcal{F} be the event that Π fails. We have*

$$\Pr[\mathcal{F} \mid \mathcal{E}_{m_A, j}^A, \mathcal{E}_{m_B, i}^B] \geq \Pr[a^* \neq X_J \mid \mathcal{E}_{m_A, j}^A, \mathcal{E}_{m_B, i}^B] \cdot \Pr[b^* \neq Y_I \mid \mathcal{E}_{m_A, j}^A, \mathcal{E}_{m_B, i}^B].$$

Proof. Without loss of generality, we assume that, in Π , after having communicated the pair (m_A, i) , Alice computes b^* , and decides her output as follows. If $b^* \neq x_j$, then Alice accepts with some fixed probability p_A , and if $b^* = x_j$ then Alice accepts with some fixed probability q_A . The probabilities p_A and q_A determines the actions of Alice. Similarly, we can assume that, after having communicated (m_B, j) , Bob computes a^* , and decides as follows. If $a^* \neq y_i$ then he accepts with some fixed probability p_B , and if $a^* = y_i$ then he accepts with some fixed probability q_B . Note that, in the case where the players do not take in account the value of a^* and b^* , then one can simply choose $p_A = q_A$ and $p_B = q_B$. Let us denote

$$R_A = \Pr[a^* = X_J \mid \mathcal{E}_{m_A, j}^A, \mathcal{E}_{m_B, i}^B], \text{ and } R_B = \Pr[b^* = Y_I \mid \mathcal{E}_{m_A, j}^A, \mathcal{E}_{m_B, i}^B].$$

Observe that

$$\Pr[\overline{\mathcal{F}} \mid \mathcal{E}_{m_A, j}^A, \mathcal{E}_{m_B, i}^B] = 1/2 \Pr[\overline{\mathcal{F}} \mid \mathcal{E}_{m_A, j}^A, \mathcal{E}_{m_B, i}^B, X_J \neq Y_I] + 1/2 \Pr[\overline{\mathcal{F}} \mid \mathcal{E}_{m_A, j}^A, \mathcal{E}_{m_B, i}^B, X_J = Y_I].$$

Now, conditioned on $X_J \neq Y_I$, the event $\overline{\mathcal{F}}$ corresponds to the event when Alice accepts and Bob accepts. Observe that, conditioned on $\mathcal{E}_{m_A, j}^A, \mathcal{E}_{m_B, i}^B$, these two latter events are independent. Moreover, conditioned on $X_J \neq Y_I$, the event $a^* \neq Y_I$ is equal to the event $a^* = X_J$. Similarly, conditioned on $X_J \neq Y_I$, the event $b^* \neq X_J$ is equal to the event $b^* = X_I$. It follows that

$$\begin{cases} \Pr[\text{Alice accepts} \mid \mathcal{E}_{m_A, j}^A, \mathcal{E}_{m_B, i}^B, X_J \neq Y_I] = R_B p_A + (1 - R_B) q_A; \\ \Pr[\text{Bob accepts} \mid \mathcal{E}_{m_A, j}^A, \mathcal{E}_{m_B, i}^B, X_J \neq Y_I] = R_A p_B + (1 - R_A) q_B. \end{cases}$$

This implies that

$$\begin{aligned} \Pr[\overline{\mathcal{F}} \mid \mathcal{E}_{m_A, j}^A, \mathcal{E}_{m_B, i}^B, X_J \neq Y_I] &= \Pr[\text{Alice accepts and Bob accepts} \mid \mathcal{E}_{m_A, j}^A, \mathcal{E}_{m_B, i}^B, X_J \neq Y_I] \\ &= \Pr[\text{Alice accepts} \mid \mathcal{E}_{m_A, j}^A, \mathcal{E}_{m_B, i}^B, X_J \neq Y_I] \cdot \Pr[\text{Bob accepts} \mid \mathcal{E}_{m_A, j}^A, \mathcal{E}_{m_B, i}^B, X_J \neq Y_I] \\ &= (R_B p_A + (1 - R_B) q_A) \cdot (R_A p_B + (1 - R_A) q_B). \end{aligned}$$

let us now consider the case when conditioning on $X_J = Y_I$. In this case, the event $\overline{\mathcal{F}}$ corresponds to the complement of the event when Alice accepts and Bob accepts. Observe that, conditioned on $X_J = Y_I$, the event $a^* \neq Y_I$ is equal to the event $a^* \neq X_J$, and the event $b^* \neq X_J$ is equal to the event $b^* \neq Y_I$. It follows that

$$\begin{cases} \Pr[\text{Alice accepts} \mid \mathcal{E}_{m_A,j}^A, \mathcal{E}_{m_B,i}^B, X_J = Y_I] = (1 - R_B)p_A + R_B q_A; \\ \Pr[\text{Bob accepts} \mid \mathcal{E}_{m_A,j}^A, \mathcal{E}_{m_B,i}^B, X_J = Y_I] = (1 - R_A)p_B + R_A q_B/ \end{cases}$$

This implies that

$$\begin{aligned} & \Pr[\overline{\mathcal{F}} \mid \mathcal{E}_{m_A,j}^A, \mathcal{E}_{m_B,i}^B, X_J = Y_I] \\ &= 1 - \Pr[\text{Alice accepts and Bob accepts} \mid \mathcal{E}_{m_A,j}^A, \mathcal{E}_{m_B,i}^B, X_J = Y_I] \\ &= 1 - \Pr[\text{Alice accepts} \mid \mathcal{E}_{m_A,j}^A, \mathcal{E}_{m_B,i}^B, X_J = Y_I] \\ & \quad \cdot \Pr[\text{Bob accepts} \mid \mathcal{E}_{m_A,j}^A, \mathcal{E}_{m_B,i}^B, X_J = Y_I] \\ &= 1 - ((1 - R_B)p_A + R_B q_A) \cdot ((1 - R_A)p_B + R_A q_B). \end{aligned}$$

Therefore, by combining the two cases, we get that

$$\begin{aligned} & \Pr[\overline{\mathcal{F}} \mid \mathcal{E}_{m_A,j}^A, \mathcal{E}_{m_B,i}^B] \\ &= \frac{1}{2} (R_A(p_A + q_A)(p_B - q_B) + R_B(p_A - q_A)(p_B + q_B) + 1 - p_A p_B + q_A q_B). \end{aligned}$$

Conditioned to the events $\mathcal{E}_{m_A,j}^A, \mathcal{E}_{m_B,i}^B$, the best protocol Π corresponds to the one that picks the values of p_A, q_A, p_B, q_B that maximize the previous quantity, restricted to the fact that p_A, q_A, p_B, q_B, R_A and R_B must be values in $[0, 1]$, and that R_A and R_B must be at least $1/2$. The maximum can be found using the Karush-Kuhn-Tucker (KKT) conditions [51]. In fact, as the restrictions are affine linear functions, the optimal value is one solution of the following system of equations:

$$\begin{aligned} (R_A + R_B - 1)p_B - (R_A - R_B)q_B - 2\mu_1 + 2\mu_5 &= 0 \\ (R_A + R_B - 1)p_A + (R_A - R_B)q_A - 2\mu_2 + 2\mu_6 &= 0 \\ (R_A - R_B)p_B - (R_A + R_B - 1)q_B - 2\mu_3 + 2\mu_7 &= 0 \\ -(R_A - R_B)p_A - (R_A + R_B - 1)q_A - 2\mu_4 + 2\mu_8 &= 0 \\ \mu_1(p_A - 1) &= 0 \\ \mu_2(p_B - 1) &= 0 \\ \mu_3(q_A - 1) &= 0 \\ \mu_4(q_B - 1) &= 0 \\ -\mu_5 p_A &= 0 \\ -\mu_6 p_B &= 0 \\ -\mu_7 q_A &= 0 \\ -\mu_8 q_B &= 0 \end{aligned}$$

From the set of solutions to this system, we obtain that $\Pr[\overline{\mathcal{F}} \mid \mathcal{E}_{m_A,j}^A, \mathcal{E}_{m_B,i}^B]$ is upper bounded by $R_A + R_B - R_A R_B$. Finally, observe that

$$(1 - R_A)(1 - R_B) = \Pr[a^* \neq X_J \mid \mathcal{E}_{m_A,j}^A, \mathcal{E}_{m_B,i}^B] \cdot \Pr[b^* \neq Y_I \mid \mathcal{E}_{m_A,j}^A, \mathcal{E}_{m_B,i}^B],$$

from which we get that

$$\Pr[\mathcal{F} \mid \mathcal{E}_{m_A,j}^A, \mathcal{E}_{m_B,i}^B] \geq \Pr[a^* \neq X_J \mid \mathcal{E}_{m_A,j}^A] \cdot \Pr[b^* \neq Y_I \mid \mathcal{E}_{m_B,i}^B],$$

as claimed. ◀

20:14 Computing Power of Hybrid Models in Synchronous Networks

We now show that, whenever the messages sent by Alice and Bob are too small, the distributions of a^* and of b^* is not far from the uniform. We make use of some basic definitions and tools on information complexity, and we refer to [49] for more details. Let (Ω, μ) be a discrete probability space. Given a random variable X we denote by $p_X : \Omega \mapsto \mathbb{R}$ the discrete density function of X , i.e., $p_X(\omega) = \Pr[X = \omega]$. We denote by $\mathbb{H} : \Omega \mapsto \mathbb{R}^+$ the entropy function, defined as $\mathbb{H}(X) = \sum_{\omega \in \Omega} p_X(\omega) \frac{1}{\log p_X(\omega)}$. Recall that, given two random variables X, Y on Ω , the entropy of X conditioned to Y is

$$\mathbb{H}(X | Y) = \mathbb{E}_{p_Y(y)}(H(X | Y = y)).$$

Moreover, let μ and ν be two probability measures on Ω . The total variation distance between μ and ν is defined as $|u - v|_{\text{TV}} = \sup_{E \subseteq \Omega} |\mu(E) - \nu(E)|$. It is known that $|u - v|_{\text{TV}} = \frac{1}{2} \sum_{\omega \in \Omega} |\mu(\omega) - \nu(\omega)|$. In addition, the Kullback-Liebler divergence between μ and ν is defined as $\mathbb{D}(\mu || \nu) = \sum_{\omega \in \Omega} \mu(\omega) \log \frac{\mu(\omega)}{\nu(\omega)}$. Given two random variables X and Y , their mutual information is defined as $\mathbb{I}(X; Y) = \mathbb{D}(p_{X,Y} || p_X p_Y)$. It is known that

$$\mathbb{I}(X; Y) = \mathbb{H}(X) - \mathbb{H}(X | Y) = \mathbb{H}(Y) - \mathbb{H}(Y | X) = \mathbb{I}(Y; X).$$

Finally, the mutual information of X, Y conditioned on a random variable Z is defined as the function $\mathbb{I}(X; Y | Z) = \mathbb{E}_{p_Z(z)}[\mathbb{I}(X; Y | Z = z)]$. Having all these notions at hand, we shall use the following technical lemmas:

► **Lemma 9** (Theorem 6.12 in [49]). *Let A_1, \dots, A_n be independent random variables, and let B be jointly distributed. We have $\sum_{i=1}^n \mathbb{I}(A_i; B) \leq \mathbb{I}(A_1, \dots, A_n; B)$.*

► **Lemma 10** (Pinsker's Inequality, Lemma 6.13 in [49]). *Let μ, ν be two probability measures over Ω . We have $|u - v|_{\text{TV}}^2 \leq \frac{2}{\ln 2} \mathbb{D}(\mu || \nu)$.*

Back into our problem, we observe that:

$$\begin{cases} \mathbb{I}(X_J; M_A | J) &= \frac{1}{n} \sum_{j \in [n]} \mathbb{I}(X_j; M_A) \leq \frac{\mathbb{I}(X; M_A)}{n} \leq \frac{\mathbb{H}(M_A)}{n} \leq \frac{k_A}{n} \\ \mathbb{I}(Y_I; M_B | I) &= \frac{1}{n} \sum_{i \in [n]} \mathbb{I}(Y_i; M_B) \leq \frac{\mathbb{I}(Y; M_B)}{n} \leq \frac{\mathbb{H}(M_B)}{n} \leq \frac{k_B}{n}. \end{cases}$$

By Pinsker's inequality, it follows that:

$$\begin{cases} \mathbb{E}_{(m_A, j)} (|p(\cdot, m_A, j) - p(\cdot, j)|_{\text{TV}}) &\leq \sqrt{\frac{k_A}{n}} \\ \mathbb{E}_{(m_B, i)} (|q(\cdot, m_B, i) - q(\cdot, i)|_{\text{TV}}) &\leq \sqrt{\frac{k_B}{n}} \end{cases}$$

These latter bounds imply that

$$\begin{cases} \mathbb{E}_{(m_A, j)} (p(a^*, m_A, j)) &\leq \frac{1}{2} + \sqrt{\frac{k_A}{n}} \\ \mathbb{E}_{(m_B, i)} (q(b^*, m_B, i)) &\leq \frac{1}{2} + \sqrt{\frac{k_B}{n}} \end{cases}$$

Now, from Lemma 8, we have that

$$\begin{aligned} \Pr[\mathcal{F} | \mathcal{E}_{m_A, j}^A, \mathcal{E}_{m_B, i}^B] &\geq p(1 - a^*, m_A, j) \cdot q(1 - b^*, m_B, i) \\ &\geq (1 - p(a^*, m_A, j)) \cdot (1 - q(b^*, m_B, i)). \end{aligned}$$

As a consequence, we have

$$\begin{aligned}
\Pr[\mathcal{F}] &= \mathbb{E}_{m_A, m_B, i, j} (\Pr[\mathcal{F} \mid \mathcal{E}_{m_A, j}^A, \mathcal{E}_{m_B, i}^B]) \\
&= \sum_{m_A, m_B, i, j} \Pr[\mathcal{F} \mid \mathcal{E}_{m_A, j}^A, \mathcal{E}_{m_B, i}^B] \cdot \Pr[\mathcal{E}_{m_A, j}^A, \mathcal{E}_{m_B, i}^B] \\
&\geq \sum_{m_A, m_B, i, j} \left(1 - p(a_{(m_A, j)}^*, m_A, j)\right) \left(1 - q(b_{(m_B, i)}^*, m_B, i)\right) \Pr[\mathcal{E}_{m_A, j}^A, \mathcal{E}_{m_B, i}^B] \\
&= \sum_{m_A, m_B, i, j} \left(1 - p(a_{(m_A, j)}^*, m_A, j)\right) \left(1 - q(b_{(m_B, i)}^*, m_B, i)\right) \Pr[\mathcal{E}_{m_A, j}^A] \cdot \Pr[\mathcal{E}_{m_B, i}^B] \\
&= \sum_{m_A, j} \left(1 - p(a_{(m_A, j)}^*, m_A, j)\right) \Pr[\mathcal{E}_{m_A, j}^A] \cdot \sum_{m_B, i} \left(1 - q(b_{(m_B, i)}^*, m_B, i)\right) \Pr[\mathcal{E}_{m_B, i}^B] \\
&= \left(1 - \mathbb{E}_{(m_A, j)}(p(a_{(m_A, j)}^*, m_A, j))\right) \cdot \left(1 - \mathbb{E}_{(m_B, i)}(q(b_{(m_B, i)}^*, m_B, i))\right) \\
&\geq \left(\frac{1}{2} - \sqrt{\frac{k_A}{n}}\right) \cdot \left(\frac{1}{2} - \sqrt{\frac{k_B}{n}}\right).
\end{aligned}$$

Since $\Pr[\mathcal{F}] \leq \epsilon$, we must have $\left(\frac{1}{2} - \sqrt{\frac{k_A}{n}}\right) \cdot \left(\frac{1}{2} - \sqrt{\frac{k_B}{n}}\right) \leq \epsilon \leq 1/5$, implying that $k_A = \Omega(n)$ or $k_B = \Omega(n)$.

5 Conclusion

In this paper, we have performed an extensive study of 2-round hybrid models resulting from mixing LOCAL, CONGEST, and BCC, and we obtained a complete picture of the relative power of these models (see Figure 1). This is a first step toward approaching the minimization problem expressed in Eq. (1), which asks for identifying the best combination of these three models for which there is an algorithm that solves a given distributed decision problem $\mathcal{L} \in \mathbf{L}^* \mathbf{B}^*$ with a minimum number of rounds, or at minimum cost. Solving this minimization problem appears to be currently out of reach, but this paper provides some knowledge about the computational power of hybrid models. Concretely, a step forward in the direction of solving the problem of Eq. (1) would be to determine whether most hybrid models remain incomparable when allowing t rounds for $t > 2$. In particular, in the case of hybrid models mixing LOCAL and BCC, we have shown that one can systematically assume that all LOCAL rounds are performed before all the BCC rounds. This does not hold for CONGEST and BCC, for 2-round algorithms. However, we do not know whether the classes $\prod_{i=1}^k \mathbf{B}^{\beta_i} \mathbf{C}^{\gamma_i}$ and $\prod_{i=1}^k \mathbf{B}^{\beta'_i} \mathbf{C}^{\gamma'_i}$ are systematically incomparable for all distinct sequences $((\beta_i, \gamma_i) : i = 1, \dots, k)$ and $((\beta'_i, \gamma'_i) : i = 1, \dots, k)$ such that $\sum_{i=1}^k (\beta_i + \gamma_i) = \sum_{i=1}^k (\beta'_i + \gamma'_i)$.

The line of research investigated in this paper could obviously be carried out by considering other models as well, in particular other congested clique models like UCC and NCC. It is easy to see that \mathbf{U} , the class of distributed languages that can be decided in one round in the unicast congested clique, is incomparable with the largest class of models considered in this paper. Namely, $\mathbf{U} \setminus \mathbf{L}^* \mathbf{B}^* \neq \emptyset$ and $\mathbf{L}^* \mathbf{B}^* \setminus \mathbf{U} \neq \emptyset$. Also, previous work on the hybrid model combining LOCAL and NCC reveals that computing the diameter of the network cannot be done in a constant number of rounds in this model. Taking this under consideration, it could be interesting to study the class $(\mathbf{LN})^*$ of distributed languages that can be decided in a constant number of rounds in the hybrid model combining LOCAL and NCC, where $(\mathbf{LN})^* = \cup_{t \geq 0} (\mathbf{LN})^t$, and \mathbf{N} denotes the class of languages decidable in one round in the node-capacitated clique.

References

- 1 Amir Abboud, Keren Censor-Hillel, Seri Khoury, and Ami Paz. Smaller cuts, higher lower bounds. *ACM Transactions on Algorithms (TALG)*, 17(4):1–40, 2021.
- 2 Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Analyzing graph structure via linear measurements. In *23rd ACM-SIAM symposium on Discrete Algorithms*, pages 459–467, 2012.
- 3 Ioannis Anagnostides and Themis Gouleakis. Deterministic distributed algorithms and lower bounds in the hybrid model. In *35th International Symposium on Distributed Computing (DISC)*, volume 209 of *LIPICs*, pages 5:1–5:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021.
- 4 Heger Arfaoui and Pierre Fraigniaud. What can be computed without communications? *SIGACT News*, 45(3):82–104, 2014.
- 5 Czumaj Artur and Christian Konrad. Detecting cliques in congest networks. *Distributed Computing*, 33(6):533–543, 2020.
- 6 John Augustine, Mohsen Ghaffari, Robert Gmyr, Kristian Hinnenthal, Christian Scheideler, Fabian Kuhn, and Jason Li. Distributed computation in node-capacitated networks. In *31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 69–79, 2019.
- 7 John Augustine, Kristian Hinnenthal, Fabian Kuhn, Christian Scheideler, and Philipp Schneider. Shortest paths in a hybrid network model. In *31st ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1280–1299, 2020.
- 8 László Babai, Anna Gál, Peter G Kimmel, and Satyanarayana V Lokam. Communication complexity of simultaneous messages. *SIAM Journal on Computing*, 33(1):137–166, 2003.
- 9 Alkida Balliu, Sebastian Brandt, Dennis Olivetti, Jan Studený, Jukka Suomela, and Aleksandr Tereshchenko. Locally checkable problems in rooted trees. In *40th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 263–272, 2021.
- 10 Reuven Bar-Yehuda, Keren Censor-Hillel, and Gregory Schwartzman. A distributed $(2 + \varepsilon)$ -approximation for vertex cover in $O(\log \Delta / \varepsilon \log \log \Delta)$ rounds. *Journal of the ACM*, 64(3):1–11, 2017.
- 11 Leonid Barenboim, Michael Elkin, and Cyril Gavoille. A fast network-decomposition algorithm and its applications to constant-time distributed computation. *Theoretical Computer Science*, 751:2–23, 2018.
- 12 Florent Becker, Adrian Kosowski, Martín Matamala, Nicolas Nisse, Ivan Rapaport, Karol Suchan, and Ioan Todinca. Allowing each node to communicate only once in a distributed system: shared whiteboard models. *Distributed Comput.*, 28(3):189–200, 2015.
- 13 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 *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 508–514, 2011.
- 14 Keren Censor-Hillel, Eldar Fischer, Gregory Schwartzman, and Yadu Vasudev. Fast distributed algorithms for testing graph properties. *Distributed Comput.*, 32(1):41–57, 2019.
- 15 Keren Censor-Hillel, Dean Leitersdorf, and Volodymyr Polosukhin. Distance computations in the hybrid network model via oracle simulations. In *38th International Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 187 of *LIPICs*, pages 21:1–21:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021.
- 16 Keren Censor-Hillel, Dean Leitersdorf, and Volodymyr Polosukhin. On sparsity awareness in distributed computations. In *33rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 151–161, 2021.
- 17 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 *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 471–480, 2019.

- 18 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.
- 19 John F. Clauser, Michael A. Horne, Abner Shimony, and Richard A. Holt. Proposed experiment to test local hidden-variable theories. *Phys. Rev. Lett.*, 23(15):880–884, 1969.
- 20 Sam Coy, Artur Czumaj, Michael Feldmann, Kristian Hinnenthal, Fabian Kuhn, Christian Scheideler, Philipp Schneider, and Martijn Struijs. Near-shortest path routing in hybrid communication networks. In *25th International Conference on Principles of Distributed Systems (OPODIS)*, volume 217 of *LIPICs*, pages 11:1–11:23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021.
- 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 Andrew Drucker, Fabian Kuhn, and Rotem Oshman. On the power of the congested clique model. In Magnús M. Halldórsson and Shlomi Dolev, editors, *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*, pages 367–376. ACM, 2014. doi:10.1145/2611462.2611493.
- 23 Albert Einstein, Boris Podolsky, and Nathan Rosen. Can quantum-mechanical description of physical reality be considered complete? *Physical Review*, 47(10):777–780, 1935.
- 24 Michael Elkin. An unconditional lower bound on the time-approximation trade-off for the distributed minimum spanning tree problem. *SIAM Journal on Computing*, 36(2):433–456, 2006.
- 25 Guy Even, Orr Fischer, Pierre Fraigniaud, Tzvil 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.
- 26 Michael Feldmann, Kristian Hinnenthal, and Christian Scheideler. Fast hybrid network algorithms for shortest paths in sparse graphs. In *24th International Conference on Principles of Distributed Systems (OPODIS)*, volume 184 of *LIPICs*, pages 31:1–31:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020.
- 27 Laurent Feuilloley and Pierre Fraigniaud. Survey of distributed decision. *Bull. EATCS*, 119, 2016.
- 28 Pierre Fraigniaud, Amos Korman, and David Peleg. Towards a complexity theory for local distributed computing. *J. ACM*, 60(5):35:1–35:26, 2013.
- 29 Pierre Fraigniaud and Dennis Olivetti. Distributed detection of cycles. *ACM Trans. Parallel Comput.*, 6(3):12:1–12:20, 2019.
- 30 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.
- 31 Mika Göös and Jukka Suomela. Locally checkable proofs in distributed computing. *Theory Comput.*, 12(1):1–33, 2016.
- 32 Thorsten Götte, Kristian Hinnenthal, Christian Scheideler, and Julian Werthmann. Time-optimal construction of overlay networks. In *40th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 457–468. ACM, 2021.
- 33 Stephan Holzer and Nathan Pinsker. Approximation of distances and shortest paths in the broadcast congest clique. In *19th International Conference On Principles Of Distributed Systems (OPODIS)*, 2016.
- 34 Tomasz Jurdziński and Krzysztof Nowicki. Connectivity and minimum cut approximation in the broadcast congested clique. In *International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, pages 331–344. Springer, 2018.

- 35 Tomasz Jurdziński and Krzysztof Nowicki. Mst in $O(1)$ rounds of congested clique. In *29th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2620–2632, 2018.
- 36 Gillat Kol, Rotem Oshman, and Raghuvansh R. Saxena. Interactive distributed proofs. In *37th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 255–264, 2018.
- 37 Amos Korman, Shay Kutten, and David Peleg. Proof labeling schemes. *Distributed Comput.*, 22(4):215–233, 2010.
- 38 Fabian Kuhn, Thomas Moscibroda, and Rogert Wattenhofer. What cannot be computed locally! In *23rd ACM Symposium on Principles of Distributed Computing (PODC)*, pages 300–309, 2004.
- 39 Fabian Kuhn and Philipp Schneider. Computing shortest paths and diameter in the hybrid network model. In *39th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 109–118, 2020.
- 40 Christoph Lenzen. Optimal deterministic routing and sorting on the congested clique. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 42–50, 2013.
- 41 Reut Levi, Moti Medina, and Dana Ron. Property testing of planarity in the CONGEST model. *Distributed Comput.*, 34(1):15–32, 2021.
- 42 Nathan Linial. Locality in distributed graph algorithms. *SIAM J. Comput.*, 21(1):193–201, 1992.
- 43 Zvi Lotker, Elan Pavlov, Boaz Patt-Shamir, and David Peleg. Mst construction in $o(\log \log n)$ communication rounds. In *15th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 94–100, 2003.
- 44 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 1096–1115, 2020.
- 45 Moni Naor and Larry Stockmeyer. What can be computed locally? *SIAM Journal on Computing*, 24(6):1259–1277, 1995.
- 46 Noam Nisan and Avi Wigderson. Rounds in communication complexity revisited. In *Proceedings of the twenty-third annual ACM symposium on Theory of computing*, pages 419–429, 1991.
- 47 David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. SIAM, 2000.
- 48 David Peleg and Vitaly Rubinfeld. A near-tight lower bound on the time complexity of distributed minimum-weight spanning tree construction. *SIAM Journal on Computing*, 30(5):1427–1442, 2000.
- 49 Anup Rao and Amir Yehudayoff. *Communication Complexity: and Applications*. Cambridge University Press, 2020.
- 50 Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. Distributed verification and hardness of distributed approximation. *SIAM Journal on Computing*, 41(5):1235–1265, 2012.
- 51 Rangarajan K Sundaram et al. *A first course in optimization theory*. Cambridge university press, 1996.
- 52 Jukka Suomela. Survey of local algorithms. *ACM Comput. Surv.*, 45(2):24:1–24:40, 2013.
- 53 Jukka Suomela. Landscape of locality. In *17th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT)*, volume 162 of *LIPICs*, pages 2:1–2:1, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.
- 54 Huacheng Yu. Tight distributed sketching lower bound for connectivity. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1856–1873, 2021.

Mending Partial Solutions with Few Changes

Darya Melnyk ✉

Aalto University, Finland

Jukka Suomela ✉

Aalto University, Finland

Neven Villani ✉

Aalto University, Finland

École Normale Supérieure Paris-Saclay, Université Paris-Saclay, France

Abstract

In this paper, we study the notion of mending: given a partial solution to a graph problem, how much effort is needed to take one step towards a proper solution? For example, if we have a partial coloring of a graph, how hard is it to properly color one more node?

In prior work (SIROCCO 2022), this question was formalized and studied from the perspective of *mending radius*: if there is a hole that we need to patch, how *far* do we need to modify the solution? In this work, we investigate a complementary notion of *mending volume*: how *many* nodes need to be modified to patch a hole?

We focus on the case of locally checkable labeling problems (LCLs) in trees, and show that already in this setting there are two infinite hierarchies of problems: for infinitely many values $0 < \alpha \leq 1$, there is an LCL problem with mending volume $\Theta(n^\alpha)$, and for infinitely many values $k \geq 1$, there is an LCL problem with mending volume $\Theta(\log^k n)$. Hence the mendability of LCL problems on trees is a much more fine-grained question than what one would expect based on the mending radius alone.

2012 ACM Subject Classification Theory of computation \rightarrow Distributed computing models; Theory of computation \rightarrow Parallel computing models

Keywords and phrases mending, LCL problems, volume model

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2022.21

Related Version *Full Version*: <https://arxiv.org/abs/2209.05363>

Funding This work was supported in part by the Academy of Finland, Grant 333837.

1 Introduction

If we have a partial solution to a graph problem, how much effort is needed to take one step towards a proper solution? For example, if we have a partial coloring of a graph, how hard is it to properly color one more node? In this work we present a formalism that captures the essence of this question, the *mending volume*: how many labels do we need to change to “patch a hole” in the solution?

We will define this concept formally in Definition 5, but for now the following informal description will suffice to understand what we mean by “patching a hole”. We are given a graph G , a partial solution λ for some graph problem Π , and some node v that is unlabeled in λ . We would like to find a new solution λ' such that node v is labeled in λ' , and also all nodes that were already labeled in λ remain labeled in λ' . We say that λ' is a *mend* of λ at node v ; we have “patched a hole” at v . Now the key complexity measure is the Hamming distance between λ and λ' , i.e., the number of nodes that we had to change. If for any G , λ , and v there is a mend λ' at node v that is within distance $T(n)$ from λ , where n is the number of nodes in G , we say that the mending volume of Π is at most $T(n)$.



© Darya Melnyk, Jukka Suomela, and Neven Villani;
licensed under Creative Commons License CC-BY 4.0

26th International Conference on Principles of Distributed Systems (OPODIS 2022).

Editors: Eshcar Hillel, Roberto Palmieri, and Etienne Rivière; Article No. 21; pp. 21:1–21:17

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1.1 Motivation

Mending volume is intimately connected with the analysis of *local search*. In particular, if the mending volume of problem Π is bounded by T , then we can start with any partial solution – including the trivially computable empty solution, which is a partial solution in which all vertices are unlabeled – and walk towards a valid solution by patching holes one at a time so that at each step we only need to consider modifications in which we change T labels.

Moreover, mending volume naturally captures the *reconfiguration effort* in computer systems. The system is initially in a valid state, but the physical structure of the system changes (e.g., a new component is installed), leading to an invalid state λ . Whenever we detect a change close to component v , we can consider v to be a hole (unlabeled in λ); now we need to find a new configuration λ' in which all components again function correctly. Further, in order to minimize service disruptions, we should also ensure that λ' is as close to λ as possible.

1.2 Contributions

It is easy to come up with graph problems where mending is trivial or very hard – these are problems with mending volume $O(1)$ or $\Theta(n)$. The work of Panconesi and Srinivasan [23] shows that the mending volume of Δ -coloring in a graph of maximum degree $\Delta \geq 3$ is $O(\log n)$. But is the mending volume for problems of this flavor always $O(1)$, $\Theta(\log n)$, or $\Theta(n)$?

We formalize this question by considering *locally checkable labeling problems* (LCLs), as defined by Naor and Stockmeyer [22]; these are problems in which we are given a graph with some maximum degree Δ , and the task is to label the nodes with labels from some finite set Σ , subject to some local constraints. Graph coloring with $k = O(1)$ colors in a graph of maximum degree $\Delta = O(1)$ is a model example of an LCL problem.

We show that already in the case of trees, it is possible to construct two infinite hierarchies of problems: for infinitely many values $0 < \alpha \leq 1$, there is an LCL problem with mending volume $\Theta(n^\alpha)$, and for infinitely many values $k \geq 1$, there is an LCL problem with mending volume $\Theta(\log^k n)$.

This shows that there is a striking difference between the mending volume that we study here and the *mending radius* that was defined recently in prior work [9]. In trees, the mending radius of any LCL problem is known to be $O(1)$, $\Theta(\log n)$, or $\Theta(n)$. Mending volume makes it possible to classify LCL problems into infinitely many classes, while mending radius only leads to three classes of problems. This is particularly relevant in balanced trees, as the diameter of a balanced tree is $\Theta(\log n)$ and hence knowing whether the mending radius is $\Theta(\log n)$ or $\Theta(n)$ does not tell us anything about the hardness of mending (both are essentially global). Mending volume, on the other hand, makes it possible to characterize the hardness in a much more fine-grained manner.

2 Related work

One of the first papers that make explicit use of the fact that some LCL problems have a logarithmic mending volume is by Panconesi and Srinivasan [23]. They compute a Δ -coloring of a graph by recoloring an augmenting path of length up to $O(\log n)$ whenever there is a conflict. However, their main interest is solving the problem in a distributed message passing model – in which the complexity of patching a hole is exactly the mending radius – and they therefore mainly focus on the mending radius instead of the mending volume of this problem.

■ **Table 1** An overview of the landscape of mending volume (MVol) for LCL problems on the classes of paths, trees and general graphs. Here ✓ denotes that LCL problems with this mending volume exist, ✗ denotes that such LCL problems cannot exist, and ? denotes an open question.

Setting	Possible mending volumes						
	$O(1)$...	$\Theta(\log n)$	$\Theta(\log^k n)$ $k > 1$...	$\Theta(n^\alpha)$ $0 < \alpha < 1$	$\Theta(n)$
Paths and cycles	✓	✗	✗	✗	✗	✗	✓
Rooted trees	✓	✗	✓	✓	✗	✓	✓
Trees	✓	✗	✓	✓	?	✓	✓
General graphs	✓	✗	✓	✓	?	✓	✓

The idea of refining the radius measure into a volume measure in the study of the landscape of LCL problems can be attributed to Rosenbaum and Suomela [25], who show similarities and differences between the models. The volume complexity for LCL problems has been further studied by Grunau et al. [11]. However, the focus of these papers is only on solvability (constructing a solution from nothing) rather than mendability (editing a partial solution to the closest complete solution) of a problem. They nevertheless highlight the fact that merely looking at the radius complexity does not capture all details of what information within that radius is actually necessary, and some problems that have the same radius complexity exhibit very different volume complexities.

Mending radius. Balliu et al. [9] introduced the first formal graph-theoretic notion of mending radius. The authors show how to use mending as a tool for algorithm design and analyze the complexities of mending on paths, rooted trees and grids.

In contrast to the definition of mending radius, our definition of mending volume captures more complexity classes of problems. A concrete example of the mending volume being more accurate than the mending radius is the three problems R_1 , R_2 , R_3 defined in Problem 1. Assume that we start with a partial solution where the root is uncolored and all other nodes are colored white. Naturally, any mending algorithm must color the root red, and start updating the descendants of the root down to the leaves. Assuming for now that this configuration is indeed the worst-case input (this is shown in Corollary 9 for a more general class of problems which includes all three of R_1 , R_2 and R_3), the mending radius of each of these problems is therefore $\Theta(\log_3 n)$. The mending volume, however, differs for all three problems R_i , and the corresponding complexities are discussed in Figure 1.

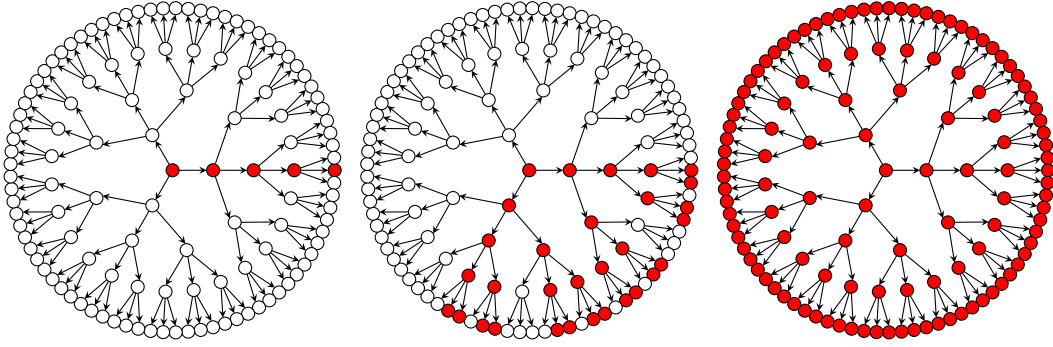
Also other papers have made use of mending radius, mainly as an algorithm design tool. Chechik and Mukhtar [14] design an algorithm for 6-coloring planar graphs using the observation that some small structures can be properly colored for any proper coloring of their surrounding vertices. Similar observations have been made for computing a Δ -coloring [23] and solving an edge-orientation with maximum out-degree $(1 + \varepsilon)a$ [18]. Recently, it has

■ **Problem 1** R_i .

Input: A balanced rooted ternary tree

Labels: red and white

Task: Color the vertices so that the root is red, and every red vertex has at least i red children.



■ **Figure 1** From left to right, solutions of R_1 , R_2 , R_3 (as defined in Problem 1) with the least number of red labels are visualized. In the case of R_2 (middle), each red vertex starting from the root in the center has two of its three children colored red, and this continues down to the leaves. The radius in all three of these examples is 4, and its growth rate as the graph gets larger is $\Theta(\log_3 n)$. The volume of R_2 is $2^{4+1} - 1$ which grows as $\Theta(n^{\log 2 / \log 3})$. On each of these three solutions the set of vertices recolored red has the same radius $\Theta(\log_3 n)$, yet the volume of the red zone is $\Theta(\log_3 n)$ for R_1 , $\Theta(n^{\log 2 / \log 3})$ for R_2 , and $\Theta(n)$ for R_3 .

been shown that mending algorithms with a constant radius can also be transformed into self-stabilizing algorithms in anonymous networks [15]. On the other hand, there are also papers that have introduced an explicit notion of mending, although using different definitions of partial solutions and complexity measures – this includes for example König and Wattenhofer [20] and Kutten and Peleg [21]. König and Wattenhofer [20] consider only faults that are an addition or deletion of a single vertex or edge at a time, and hence their definition of partial solutions features only at most a small number of unlabeled vertex. They also classify the cost of fixing a fault only in terms of local (constant-radius) or global (non-constant radius). Kutten and Peleg [21] are interested in the time needed to compute a complete solution as a function of the initial number of failures.

Local search. The idea of mending volume is closely related to local search in optimization problems (in the context of traditional centralized algorithms). Often one starts with a suboptimal solution and tries to converge to a better solution from there. Usually a problem is first solved by computing some possibly random initial variable assignment that satisfies the constraints, see e.g. [12, 16, 24]. Then, a local search algorithm is applied to find a better solution in the vicinity of the previous one.

A classic application of local search in combinatorial optimization is the traveling salesman problem; local search is often applied to hard problems in order to achieve a good approximation of the optimal solution [1, 3]. On the negative side, Johnson et al. [19] showed that an exponential number of iterations may be needed if the cost function can take exponential values. Ausiello and Protasi [5] later defined the class of guaranteed local optima (GLO) problems where the values of the cost function are bounded by a polynomial and showed that such problems can be solved in a polynomial number of iterations. Halldórsson [17] showed that local search can help to improve worst-case approximation guarantees by starting with a greedy solution and improving it locally using local search. He provides approximation results for various problems, such as the independent set, k -dimensional matching and k -set packing in nearly-linear sequential time. Chandra and Halldórsson [13] later showed a $2(k+1)/3$ -approximation algorithm for the weighted k -set packing problem, thus improving a previous result from Bafna et al. [8] and Arkin and Hassin [4].

Self-stabilization. Mending can also be seen as an approach for the correction step of self-stabilizing algorithms. Self-stabilization is typically implemented by the conjunction of a failure detector [10] and a recovery procedure [6]. Similarly to prior work in [7] and [2], our approach applies to problems where inconsistencies are locally detectable, but recovering from failures usually requires global reconfiguration. Indeed the restriction of our study to the class of LCL problems guarantees that the consistency of the state can be checked with only a local view, but since the mending radius in [9] already covers the case of problems that can be corrected with constant locality, most of our new results apply to problems that are not locally correctable. Compared to [26], our setting is more restricted since for the problems we study the history of a vertex is not relevant, but we are also interested in optimality of the final solution in terms of its Hamming distance to the initial solution. We also have the same point of view as [26] that if the complexity is chosen to be the execution time, then it measures more properties related to the implementation or the model than inherent properties of the problem and hence we aim for a model-independent notion. Unlike [26] however, in cases where several failures occur simultaneously and are repaired sequentially through successive mending procedures each resulting in an intermediate partial solution, our measure of complexity considers only the distance from one partial solution to the next and it does not guarantee minimizing the total distance from the initial solution to the final complete solution. The other main difference is that we classify problems in terms of the distance to the nearest solution whereas the measure of locality in [26] is the radius of the view – i.e. the amount of information – that is required to compute said nearest solution.

3 Preliminaries

Our definition of the mending volume is built along the lines of the definition of the mending radius in [9]: we define the mending volume as a measure entirely independent of any distributed computing model and we place ourselves in the context of Locally Checkable Labeling problems (LCLs) first introduced in [22]. We use the same definition of partial solutions as [9] in order to make our results comparable. A reader who is familiar with the notions of graph labeling problems – and LCLs in particular – as well as with the specific definition of partial solutions from [9] may skip directly to Section 4 in which we introduce a formal definition of the mending volume.

3.1 Locally checkable labelings

LCLs are labeling problems on bounded-degree graphs. In these problems, an input graph with maximum degree $\Delta = O(1)$ is given and the task is to produce an assignment of labels to vertices in a way that satisfies some predetermined local constraints. The specification of an LCL problem is done by means of a local verifier.

► **Definition 1** (Local verifier). *A verifier ϕ is a function that maps tuples (G, λ, v) to $\{\text{happy}, \text{unhappy}\}$, where v is a vertex and λ a labeling of G . We say that the verifier ϕ accepts λ if $\phi(G, \lambda, v) = \text{happy}$ for all v , otherwise it rejects λ .*

In addition, ϕ is local if, for some constant radius r , whenever (G_1, λ_1) and (G_2, λ_2) coincide over the radius- r neighborhood of v_1 and v_2 then they have the same image according to ϕ .

We write $\mathcal{N}_r(v)$ for the radius- r neighborhood of v and $(G, \lambda)|_V$ for the restriction of G and λ to the subgraph and labeling with only vertices from V , this constraint can be expressed more formally as: $(G_1, \lambda_1)|_{\mathcal{N}_r(v_1)} \simeq (G_2, \lambda_2)|_{\mathcal{N}_r(v_2)}$ implies $\phi(G_1, \lambda_1, v_1) = \phi(G_2, \lambda_2, v_2)$.

An LCL problem is entirely characterized by a finite set of labels and a local verifier.

► **Definition 2** (Locally Checkable Labeling). *A Locally Checkable Labeling problem Π is represented by a finite set of labels Σ , a class of input graphs \mathcal{G} , and a local verifier ϕ . An instance of Π is a graph $G \in \mathcal{G}$. A solution is a labeling λ of G over Σ that is accepted by ϕ .*

3.2 Partial solutions

Mending takes as input an incomplete labeling and extends it into one that is one step closer to being complete. Since graph labelings were defined to be complete over all vertices, the most natural way to define partial solutions is to extend the set of labels with one fresh label \perp that is interpreted as “unlabeled”, and adapt the local constraints to allow labelings that involve this new label. We will often refer to vertices that are labeled \perp simply as “unlabeled vertices” or “holes”.

A desirable definition of partial solutions should satisfy the following three properties:

1. A partial solution without any hole is a complete solution.
2. The empty labeling (the constant function $\lambda_{\perp} : x \mapsto \perp$) is a partial solution.
3. A sub-solution of a partial solution is also a partial solution. That is, if λ is a partial solution then any labeling

$$\lambda_S : x \mapsto \begin{cases} \lambda(x) & \text{if } x \in S \\ \perp & \text{otherwise} \end{cases}$$

is a partial solution.

As stated in [9], the following is a simple way to satisfy all of these constraints: extend the verifier ϕ' to be **happy** whenever an unlabeled vertex is visible in the radius- r neighborhood, otherwise fall back to the same rules as ϕ .

► **Definition 3** (Partial solution). *For $\Pi = (\Sigma, \mathcal{G}, \phi)$, where ϕ has radius r , let $\Sigma^* = \Sigma \sqcup \{\perp\}$, and define a relaxation $\Pi^* = (\Sigma^*, \mathcal{G}, \phi^*)$ of Π to allow empty labels.*

For a labeling λ' over Σ^ , define $\phi^*(G, \lambda', v)$ as follows: if there exists a node u_{\perp} within distance r of v such that $\lambda'(u_{\perp}) = \perp$, then $\phi^*(G, \lambda', v) := \mathbf{happy}$; otherwise, let λ be any labeling over Σ that agrees with λ' on $G_{|\mathcal{N}_v(r)}$ and set $\phi^*(G, \lambda', v) := \phi(G, \lambda, v)$.*

We define $\text{dom}_{\Sigma}(\lambda')$ to be the set of vertices that λ' labels with labels from Σ . A labeling (resp. solution) of Π^ is called a partial labeling (resp. partial solution) of Π .*

One can easily check that all the desirable properties stated above are satisfied by Definition 3; this fact is also proven in [9]. Note that this definition of partial solutions has a notion of locality that is consistent between labelings and partial labelings: the verifiers ϕ and ϕ^* have the same locality radius.

We can now define what it means to mend a partial solution: a mend of λ is a new partial solution with one specific vertex no longer labeled \perp , and no additional \perp labels.

► **Definition 4** (Mend). *For a partial solution λ of Π on an instance G , we say that λ' is a mend of λ at $v \in G$ if the following hold:*

Validity: λ' is a partial solution.

Progress: $\text{dom}_{\Sigma}(\lambda) \cup \{v\} \subseteq \text{dom}_{\Sigma}(\lambda')$, that is, no \perp was added and v is no longer labeled \perp .

The mending problem $\text{Mend}(\Pi)$ associated with an LCL Π is the following task: given $G \in \mathcal{G}$, λ solution of Π^* and v hole of λ , produce λ' a mend of λ at v . We call such a tuple (G, λ, v) an instance of $\text{Mend}(\Pi)$, and λ' a solution.

4 Complexity landscape of mending volume

Having defined LCLs and partial solutions, we can now introduce mending volume. This definition (see Section 4.1) is a purely graph-theoretic measure of the optimal solution for a worst-case instance of a mending problem. Later, in Section 4.2, we develop a technique for designing LCLs that have a specific mending volume on infinite rooted trees. In Section 4.3, we show that these problems can be transferred to finite and non-oriented trees while keeping the same mending volume complexity. Finally, in Sections 4.4 and 4.5, we apply these design techniques to obtain problems that have mending volume $\Theta(n^\alpha)$, $0 < \alpha < 1$ or $\Theta(\log^k n)$, $k \in \mathbb{N}^*$, thereby providing examples of complexities that the mending volume exhibits that were not observed previously in the study of the mending radius.

4.1 Mending volume: Definition

For two labelings λ and λ' , we define $\text{diff}(\lambda, \lambda') := \{v: \lambda(v) \neq \lambda'(v)\}$ such that $|\text{diff}(\lambda, \lambda')|$ is the Hamming distance between two partial solutions. We define the mending volume of a problem Π as the distance between the partial solution and the optimal mend for the worst-case instance (G, λ, v) of $\text{Mend}(\Pi)$. Here, G is an input graph from the family on which Π is defined, λ is a partial solution, and v is a hole s.t. $\lambda(v) = \perp$ at which λ must be mended.

► **Definition 5** (Mending volume). *The mending volume of problem Π is*

$$\text{MVol}(\Pi) := \max_{G, \lambda, v} \min \{|\text{diff}(\lambda, \lambda')| : \lambda' \text{ is a mend of } \lambda \text{ at } v\}.$$

4.2 Mending in infinite rooted trees

In what follows, we will establish the landscape of possible mending volumes. For a summary, please refer to Table 1 that was introduced earlier. To this end, we give examples of LCL problems that have logarithmic, polylogarithmic and polynomial mending volumes. The definitions of these problems are made easier by the fact that all of them are of a specific type that we call *propagation problems*.

The complexity analysis of problems in this class is straightforward for two reasons: (1) they admit a simple matrix description by an encoding shown in Section 4.2.2, and (2) we only need to study their behavior in infinite regular trees thanks to results from Section 4.3 which allow us to transfer complexity results from infinite regular trees to finite graphs. The advantage of infinite regular trees is that the complexity analysis is simplified by the absence of high-degree nodes, leaves, and other irregularities of the input graph. This restriction of only considering infinite regular rooted trees also has the complementary effect of illustrating that even simple problems already exhibit a rich variety of mending volume complexities. Since any propagation problem with mending volume T can be transformed into a problem on general trees or graphs with the same mending volume T , as proven in Section 4.3, our choice does not restrict the generality of our results when it comes to existence results. This does however makes the impossibility results not directly applicable to the case of general graphs. In fact, we do not yet know if these results still hold for general graphs.

4.2.1 Propagation problems

In this section, we define propagation problems on infinite rooted trees, with the goal to use them as a design tool for LCLs that exhibit specific mending volume complexities.

► **Definition 6** (Infinite Δ -regular rooted trees). *An infinite Δ -regular rooted tree (or simply infinite rooted trees when Δ is clear from the context) is a tree with the following properties:*

- *exactly one vertex is distinguished as the root;*
- *each vertex admits a unique directed path to the root;*
- *each vertex has exactly Δ incoming edges.*

Note that this class of graphs only consists of a single graph for a fixed Δ . On this class of input graphs, we define propagation problems as any LCL problem that is constructed according to the procedure explained in Definition 7.

► **Definition 7** (Construction of a propagation problem). *In the label set Σ , distinguish two special labels – the initial label l_0 , and the wildcard label l_- . Let $\Sigma' := \Sigma \setminus \{l_-\}$. Choose some $\mu : \Sigma' \times \Sigma' \rightarrow \mathbb{N}$ and some $\Delta \geq \max_{l \in \Sigma'} \sum_{l' \in \Sigma'} \mu(l, l')$. This defines an LCL on infinite Δ -regular rooted trees, with locality 1, where the radius-1 neighborhood of v labeled by λ is accepted if all of the following constraints are satisfied:*

- *$\lambda(v) = l_0$ if v is the root;*
- *if $\lambda(v) = l \neq l_-$ then, for every $l' \in \Sigma'$, there are at least $\mu(l, l')$ children of v labeled l' .*

In other words, we only allow labeling constraints of the form “any vertex labeled l must have at least $\mu(l, l')$ children labeled l' ” and “the root must be labeled l_0 ”. The requirement $\Delta \geq \max_{l \in \Sigma'} \sum_{l' \in \Sigma'} \mu(l, l')$ is chosen such that all constraints are compatible with each other. Note that there are no constraints involving the wildcard label l_- in this definition: it may appear as a child of any other label, and it may have any labels as its own children. In particular, the labeling where the root is unlabeled and all non-root vertices are labeled l_- is always a valid partial solution. We will show in Corollary 9 that this labeling is the worst-case instance for most propagation problems.

Since the input graphs on which these problems are defined are infinite, and since these problems often produce mends that have infinite volume, it will be useful to study the volume in terms of the number of modified labels at distance at most d_{\max} from the hole.

4.2.2 Matrix representation

Let M be a matrix of size $|\Sigma'| \times |\Sigma'|$ defined as $M[l, l'] = \mu(l, l')$. This means that the coefficient $M[l, l']$ indicates how many children labeled l' each vertex labeled l must have. Observe that the coefficient $M^d[l, l']$ of the d -th power of M is the tightest lower bound on how many children labeled l' a vertex labeled l must have at distance d for a complete solution to be accepted. By induction, a vertex labeled l must have at least $M[l, l']$ children labeled l' at distance 1; each of its children labeled l'' at distance d (of which there are $M^d[l, l'']$ by inductive hypothesis) must then have at least $M[l'', l']$ children labeled l' , which makes for a total of

$$\sum_{l'' \in \Sigma'} M^d[l, l''] M[l'', l'] = M^{d+1}[l, l']$$

children labeled l' at distance $d + 1$. We argue in Theorem 8 that this provides bounds for MVol .

We write $\|L_l M^d\| := \sum_{l' \in \Sigma'} M^d[l, l']$, where L_l is the vector with a 1 only in position l . This corresponds to counting the total number of children of l at distance d that have any label among Σ' , which is also the total number of modified labels at distance d when starting from the initial labeling that consists of an unlabeled root and all other vertices labeled l_- . We show in Theorem 8 that this quantity expresses both upper and lower bounds on the mending volume up to a fixed distance d_{\max} of the propagation problem described by M .

► **Theorem 8** (Mending complexity of a propagation problem). *The mending volume up to distance d_{\max} of a propagation problem represented by M is between $\sum_{d=0}^{d_{\max}} \|L_{l_0} M^d\|$ and $\max_{l \in \Sigma'} \sum_{d=0}^{d_{\max}} \|L_l M^d\|$.*

Proof. We start with the lower bound. Recall that all vertices in the input graph have degree exactly Δ . Consider an initial partial labeling λ in which the root is initially unlabeled, and all other vertices are labeled l_- . A mend λ' of λ at the root will have to be a complete solution, and thus require the root to be labeled l_0 . By the previous observation, at distance d from the root, there must be at least $M^d[l_0, l']$ vertices in λ' labeled l' that must have been modified during the mending. Therefore $\sum_{d=0}^{d_{\max}} \|L_{l_0} M^d\|$ is a lower bound for how many labels were modified at distance at most d_{\max} .

We can now show the upper bound. An important characteristic of the family of propagation problems is that the output of the verifier depends only on a portion of the labels of the children. Once sufficiently many children are labeled correctly, the remaining ones have no impact. This means that no initial configuration can force more than $M^d[l, l']$ labels l' to be added at distance d from a vertex v labeled l : in the worst case, it suffices to arbitrarily choose $M[l, l']$ children at each level and color them accordingly while ignoring all the other children. Thus the worst-case instance has mending cost at distance d_{\max} no more than $\max_{l \in \Sigma'} \sum_{d=0}^{d_{\max}} \|L_l M^d\|$. ◀

► **Corollary 9** (Worst-case instance of a propagation problem). *If l_0 is such that $\|L_{l_0} M^d\| = \Omega(\max_{l \in \Sigma'} \|L_l M^d\|)$ then the initial instance where the root is unlabeled and all other vertices are labeled l_- is the worst-case instance.*

The condition for Corollary 9 is satisfied at least for problems where all labels are reachable from l_0 in the sense that any complete solution must contain every label at least once. Put otherwise, for every $l' \in \Sigma'$ there must exist some $d_{l'}$ for which $M^{d_{l'}}[l_0, l'] \neq 0$. This is also the case for every propagation problem we will define in the rest of this paper.

4.2.3 Landscape of the growth rate of matrix exponentiation

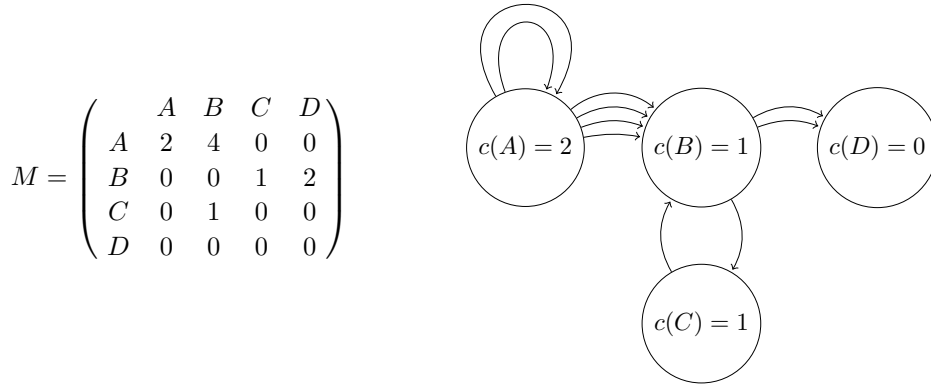
In this section, we turn to a study of possible growth rates of the quantity $\|L_l M^d\|$ introduced in Section 4.2.2. This quantity is upper bounded by $|\Sigma'| \times \max_{l' \in \Sigma'} M^d[l, l']$ (approximate a sum by the number of elements multiplied by the greatest element). In order to determine the mending volume up to a multiplicative constant of a propagation problem in which all labels are reachable from the root label, we have thus established that it is sufficient to look at the growth rate as a function of d of $\max_{l, l' \in \Sigma'} M^d[l, l']$ – the greatest coefficient of M^d . We will denote it as $\max M^d$.

In the following analysis, we make use of the interpretation of M as the adjacency matrix of a graph G_M . Here G_M is a directed graph with one vertex for each element of Σ' . For every pair $(v_l, v_{l'})$ there are exactly $M[l, l']$ directed edges from v_l to $v_{l'}$. Further, there are $M^d[l, l']$ walks of length exactly d from v_l to $v_{l'}$ in G_M . Let $c(l)$ be the number of cycles in G that contain v_l . We say that a vertex v_l is of type 0 if $c(l) = 0$, type 1 if $c(l) = 1$, and type 2 if $c(l) \geq 2$. Figure 2 shows an example of a matrix M and the corresponding graph G_M .

We will show that the types of the vertices fully determine the growth rate of $|M^p|$: if some vertex is part of several cycles, then there are exponentially many paths of length d from that vertex to itself. Otherwise, if all vertices are part of at most one cycle, then there are only polynomially many paths of length d from one vertex to another.

► **Lemma 10.** *Assume that v_l is a vertex of type 2. It holds that $M^d[l, l] = \Omega((1 + \beta)^d)$ for some $\beta > 0$.*

21:10 Mending Partial Solutions with Few Changes



■ **Figure 2** An example of matrix M describing a propagation problem over the label set $\Sigma' = \{A, B, C, D\}$. The corresponding G_M is shown on the right, where A is of type 2, B and C are of type 1, and D is of type 0.

Proof. Let C_1, C_2, \dots be the $c(l)$ distinct cycles that contain v_l . Let L_1, L_2, \dots denote their lengths respectively, and let $L := \text{lcm}(L_1, L_2, \dots)$. There are at least $c(l)$ walks of length L from v_l to itself, each following only one of the cycles C_j L/L_j number of times. Hence, for all k , there are at least $c(l)^k$ walks of length $d := kL$ from v_l to itself and therefore $M^d[l, l] \geq c(l)^{d/L}$ walks for infinitely many values of d . Thus $M^d[l, l] = \Omega((c(l)^{1/L})^d)$. ◀

► **Corollary 11.** *Let vertex v_l be of type 2 and reachable from v_{l_0} . Then $M^d[l_0, l] = \Omega((1 + \beta)^d)$ for some $\beta > 0$.*

In terms of the corresponding propagation problem, the interpretation of this fact is that any label l corresponding to a vertex of type 2 must have at least two descendants with the same label l at a certain fixed distance, each of which must also satisfy the same property. This produces an exponentially increasing amount of vertices labeled l as we move further away from the root.

► **Lemma 12.** *If there is no vertex of type 2 reachable from v_{l_0} , then for all labels l , $M^d[l_0, l] = O(d^k)$ for some constant k .*

Proof. For each vertex v_l , we denote $C(l)$ to be the cluster of v_l , defined as follows: $C(l) := \{l\}$ if $c(l) = 0$; otherwise, $C(l) := \{l' \mid v_l \rightarrow^* v_{l'} \rightarrow^* v_l\}$ describes the vertices in the same cycle as v_l . Since $c(l) \leq 1$, the clusters form a partition of Σ' . We use K to denote the number of clusters. We now consider all possible walks from l_0 to some label l by separating each step of the walk between those that stay in the same cluster and those that change cluster. The constraints on the cycles of the graph obtained from the assumption that every vertex is of type at most 1 give us information on how many times a step of the walk can change cluster.

Construct G'_M whose vertices are the clusters of G_M , by contracting each cluster into a single vertex while keeping duplicate edges between different clusters, and removing edges inside a cluster. The resulting graph G'_M is acyclic as, otherwise, some vertex would be part of several cycles and would not be of type 0 or 1 as we have assumed. Any walk W of length exactly d in G_M from v_l to $v_{l'}$ is uniquely defined by

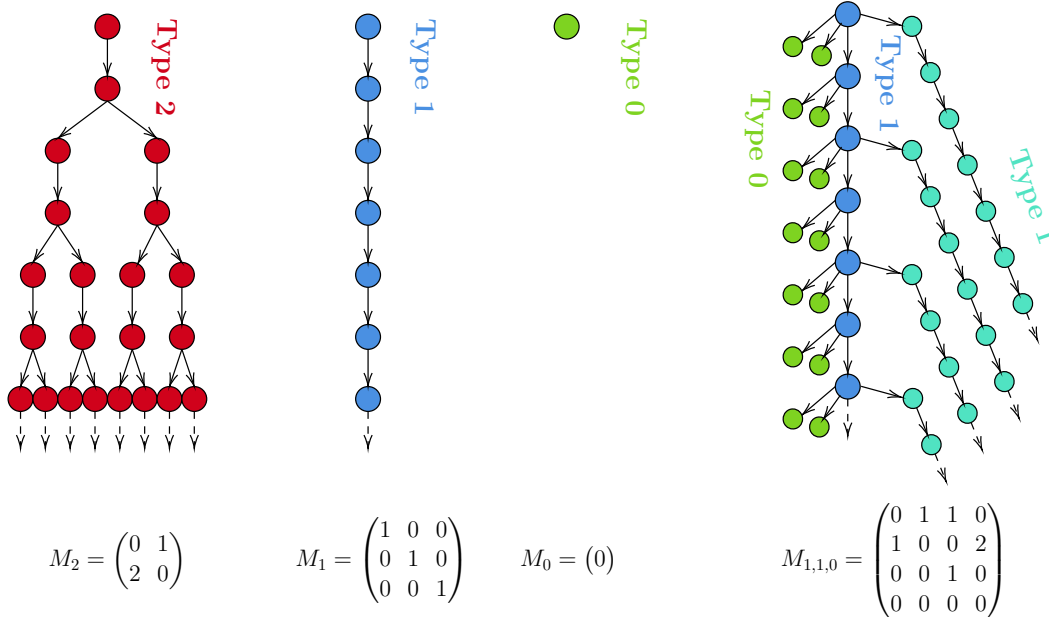


Figure 3 Examples of subtrees obtained by considering the vertices with a label from the same cluster, for clusters of type 2 (left, red), type 1 (center left, blue) and type 0 (center right, green). Below each tree is an example matrix M that exhibits the behavior in question. As stated in Lemma 10, a cluster of type 2 produces an exponentially growing tree where some vertices have several children within the same cluster. This is different to a cluster of type 1, in which each vertex has exactly one child within the same cluster, and cluster of type 0 where there are no cycles and thus also no children from the same cluster. The last tree on the right illustrates Lemma 12, where, no matter how we combine clusters of types 1 and 0 (in this specific case the clusters are $\{1, 2\}$ of type 1, $\{3\}$ of type 1, and $\{4\}$ of type 0), the clusters can never achieve an exponential growth the same way a type 2 cluster can. They can therefore produce at most polynomial growth with a larger polynomial degree as we combine more clusters.

- a walk W' in G'_M from $C(l)$ to $C(l')$, let $C(l) = C_1 \rightarrow C_2 \rightarrow \dots \rightarrow C_{|W|} = C(l')$ be this walk;
- the length d_i of the walk within C_i , for each $1 \leq i \leq |W|$ (because no vertex is of type 2, there is only one such walk for a given length).

Note that $d_1 + \dots + d_{|W|} + (|W| - 1) \leq d$ since the total length of the walk is more than the sum of each walk inside a cluster, and $|W| \leq K \leq |\Sigma'|$ since each cluster is of size at least 1. There are finitely many walks W' in G'_M , since it is a finite acyclic graph, and for each of them the number of possible tuples $(d_1, \dots, d_{|W|})$ is bounded by d^K . Thus the number of walks W of length d is polynomially bounded by $O(d^K)$. ◀

In contrast to Lemma 10, the interpretation of this situation in terms of the underlying propagation problem is as follows: if there are only vertices of type 0 or 1 then, from each vertex labeled l , there is at most one path that contains other vertices labeled l . Figure 3 shows what we can expect solutions to typical propagation problems satisfying the conditions from Lemmas 10 and 12 to look like.

We observe further that if there is a walk in G'_M that goes through two or more cycles, then there are at least $\Omega(d)$ paths of length d . Whereas if G'_M contains only isolated cycles, then there are at most $O(1)$ paths of length d . Thus Theorem 13 holds.

21:12 Mending Partial Solutions with Few Changes

► **Theorem 13** (Landscape of $\max M^d$). *The growth rate of $d \mapsto \max |M^d|$ is either*

1. *eventually zero;*
2. *or $\Theta(1)$;*
3. *or $O(d^p)$ for some value $p \geq 1$;*
4. *or $\Omega((1 + \beta)^d)$ for some $\beta > 0$.*

By comparing Theorem 13 to Figure 3, we can see that all these families of growth rates are represented there: from left to right, $2^{d/2} = \Omega((1 + \beta)^d)$, constant 1 which is of course in $\Theta(1)$, eventually zero, and $d/2 + 3 = O(d^p)$.

We can combine Theorem 13 with the previously established bounds from Theorem 8 stating how to relate the mending volume to the growth of $\max M^d$. We thus obtain the following corollary:

► **Corollary 14** (Landscape of the mending volume on infinite rooted trees). *The mending volume up to distance $d := \log_\Delta n$ of a propagation problem is either*

- *$O(1)$ if $\max M^d$ is eventually zero;*
- *or $\Theta(\log n)$ if $\max M^d = \Theta(1)$;*
- *or $O(\log^k n)$ for some $k > 1$ if $\max M^d = O(d^p)$;*
- *or $\Omega(n^\alpha)$ for some $0 < \alpha < 1$ if $\max M^d = \Omega((1 + \beta)^d)$.*

This concludes the survey of the landscape of propagation problems on infinite rooted trees. We found that there are complexity classes $O(1)$, $\text{poly}(\log n)$ and $\Omega(n^\alpha)$, with a gap between $\omega(\log n)$ and $o(\log^2 n)$. In Sections 4.4 and 4.5 we will look more closely at the classes of growths $\Omega(n^\alpha)$ and $\Theta(\log^k n)$ to show that infinitely many values of α and k can appear.

4.3 From infinite regular rooted trees to general trees

Some of the results from Section 4.2.3 are applicable to trees even if they are no longer infinite rooted and regular. Indeed there is a straightforward translation that transforms a propagation problem on infinite Δ -regular rooted trees into one that has the same growth properties, but can be defined on finite rooted trees where some vertices are of degree lower than Δ . This construction is detailed in Problem 2, which shows how for any generic problem Π we can remove the constraints of finiteness and Δ -regularity while preserving the mending volume complexity.

■ **Problem 2** Generalization of Π to finite and non- Δ -regular trees.

Input: Any tree

Labels: Same as Π

Task: Any vertex of degree exactly Δ must satisfy the labeling constraints from Π

We now argue that the fact that these trees are finite does not affect the conclusions made earlier about the possible complexities. The worst-case instance can namely still be constructed as a balanced finite Δ -regular tree. We prove that the mending process is just as efficient in the case that the tree is unbalanced as it is in a balanced tree, by proving the following intermediate result: a labeling picked at random among a set of possible labelings of an unbalanced tree is on average asymptotically as efficient as the optimal labeling of a balanced tree. This implies that the optimal labeling of an unbalanced tree is asymptotically as efficient as the optimal labeling of a balanced tree. To show this, we use the fact that the optimal labeling must be at least as efficient as the average performance of several valid labelings.

► **Theorem 15** (Generalization to finite unbalanced trees). *If Π is a propagation problem, it has the same mending volume complexity on unbalanced trees as it does on balanced Δ -regular trees.*

Proof. Assume that we wish to label a tree of size $n + 1$ rooted in v . Each of the Δ subtrees that are children of v have size $n/\Delta + d_i$ for $1 \leq i \leq \Delta$ where $\sum_{i=1}^{\Delta} d_i = 0$, and we wish to assign a new labeling to each of them. The growth rate f_j^{bal} ($1 \leq j \leq \delta$) of the number of labels that would need to be modified for a balanced tree is uniquely determined by its assigned root label l_i .

A key observation is that as f_j^{bal} is defined by some $\sum_{d=0}^{d_{\max}} \|L_l M^d\|$, it is eventually concave in n (which possibly includes eventually constant). This is because there are at most Δ as many modified labels at any distance $d + 1$ as at the previous distance thus $\|L_l M^{d+1}\| \leq \Delta \|L_l M^d\|$, yet from d to $d + 1$ the total number of vertices grows by a factor exactly Δ .

The total number of modified labels in a tree of size $n + 1$ is thus

$$f^{\text{bal}}(n + 1) = 1 + \sum_{j=1}^{\Delta} f_j^{\text{bal}}(n/\Delta).$$

Assume inductively that the true number of modified labels in any non-balanced tree of size n' is less than $K \cdot f_j^{\text{bal}}(n')$, i.e. that a balanced tree is asymptotically the worst-case input. Let $c_{i,j}$ denote this number for the subtree i if it were assigned root label j . The average performance of an algorithm that distributes the required labels randomly among all children with equal probability is then

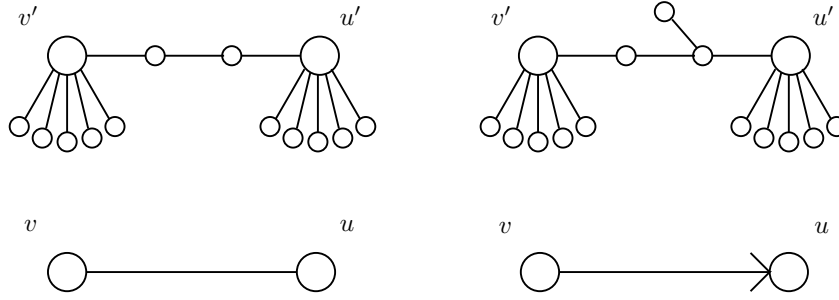
$$\begin{aligned} c' &= \text{avg}_{\sigma \in \mathfrak{S}_{\Delta}} 1 + \sum_{i=1}^{\Delta} c_{i,\sigma(i)} \\ &\leq \text{avg}_{\sigma \in \mathfrak{S}_{\Delta}} 1 + \sum_{i=1}^{\Delta} K \cdot f_{\sigma(i)}^{\text{bal}}(n/\Delta + d_i) && \text{induction hypothesis} \\ &\leq 1 + K \cdot \sum_{j=1}^{\Delta} \text{avg}_{\sigma \in \mathfrak{S}_{\Delta}} f_j^{\text{bal}}(n/\Delta + d_{\sigma(j)}) && \text{reassign indices} \\ &\leq 1 + K \cdot \sum_{j=1}^{\Delta} f_j^{\text{bal}}(n/\Delta) && \text{by concavity of all } f_j^{\text{bal}} \\ &\leq K \cdot f^{\text{bal}}(n + 1). \end{aligned}$$

The induction hypothesis trivially holds for some big enough K once given n that satisfies the main requirement of functions f_j^{bal} reaching their eventual concavity, hence the result for all big enough sizes of trees.

For now this result is implicitly restricted by the condition that n be divisible by Δ , indeed $f_j^{\text{bal}}(n/\Delta)$ is not defined in general for fractional n/Δ . The result extends to all sizes of trees – still under the condition that they are big enough – from the following observation: between n and the next larger integer divisible by Δ there is at most a constant multiplicative factor Δ which translates to at most a multiplicative Δ additional labels being modified. This multiplicative factor has no impact on the asymptotic behavior. ◀

We further show that if the labeling assumes an orientation of the edges then a variant of the problem with the same mending volume complexity can be defined on unoriented graphs by encoding the orientation in the graph structure. One possible encoding of the orientation

21:14 Mending Partial Solutions with Few Changes



■ **Figure 4** Oriented interpretation of an unoriented graph. Top: pairs of vertices in G ; bottom: their interpretation in \bar{G} . Note that G is a tree if and only if \bar{G} is a tree.

is the following: given an unoriented graph G , let *leaves* be the vertices with degree exactly 1. To interpret G as an oriented graph \bar{G} with maximum degree Δ , extract the subset \bar{V} of vertices that have at least $\Delta + 1$ leaves as neighbors. This provides the vertex set of \bar{G} . An edge between \bar{u} and \bar{v} is added if between the corresponding vertices u and v in G , there is a path of length exactly 2, connected to at most one leaf. If there is such a leaf on the vertex of the path closest to u , consider the edge oriented from \bar{v} to \bar{u} . Otherwise, if there is no leaf then the edge is not oriented. Figure 4 visualizes this transformation.

A labeling λ of G is interpreted as a labeling $\bar{\lambda}$ of \bar{G} by defining $\bar{\lambda}(\bar{v}) := \lambda(v)$.

Any local property that $\bar{\lambda}$ should satisfy can be verified locally on λ : a property of $(\bar{G}, \bar{\lambda})$ that can be computed from the restriction $(\bar{G}, \bar{\lambda})|_{\bar{V}}$ to a neighborhood \bar{V} can also be computed from $(G, \lambda)|_{\mathcal{N}_2(V)}$, where $V = \{u \mid \bar{u} \in \bar{V}\}$.

4.4 Application: $\text{MVol} = n^{\Theta(1)}$

In this section and the next, we show examples of problems that exhibit polynomial and polylogarithmic complexities, with a particular focus on showing which values of $0 < \alpha < 1$ and $k \geq 1$ can appear for complexities $\Theta(n^\alpha)$ and $\Theta(\log^k n)$.

The prior analysis resulting in Corollary 14 suggests that, in order to construct a problem with volume $n^{\Theta(1)}$, we should consider a propagation problem whose matrix M has exponential growth for $d \mapsto \max M^d$. A good candidate is the problem described by $M = (2)$ for $\Delta = 3$. It describes the following problem:

■ **Problem 3** Polynomial propagation.

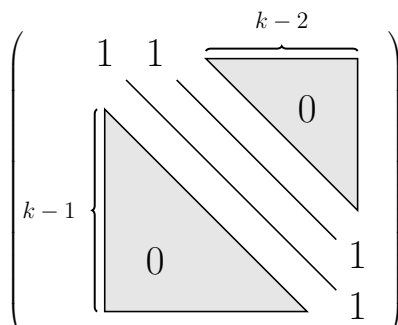
Input: An infinite rooted Δ -regular tree

Labels: red and white

Task: Color the vertices according to the following rules, by order of precedence:

- any labeling is valid for a vertex that does not have exactly 3 children;
- any labels are valid for the children of a vertex labeled white;
- if a vertex is red, it needs at least two of its children to be red;
- the root has to be red.

One can notice that this problem happens to be the same as one previously introduced in Definition 1 and for which an example solution is shown in Figure 1 (middle). Using the terminology of Definition 7, red is the initial label, and white is the wildcard label. From the initial labeling consisting of the root being unlabeled and all other vertices being white, a mend needs to recolor 2^d vertices at layer d from white to red. For a balanced ternary tree, this will produce a total of $2^{\log_3 n + 1} - 1$ recolored vertices, i.e. $\Theta(n^{\ln 2 / \ln 3})$.



■ **Figure 5** M_k of size $k \times k$ exhibits growth $\Theta(p^k)$ for the root label l_0 corresponding to line 1.

By slightly adjusting the parameters, we can engineer any rational power of n : the problem described by $M = (2^p)$ for $\Delta = 2^q$, where $q > p > 1$, will exhibit complexity $\Theta(n^{\ln 2^p / \ln 2^q}) = \Theta(n^{p/q})$.

4.5 Application: $MVol = (\log n)^{\Theta(1)}$

We now show how to construct a problem that has mending volume $\Theta(\log^k n)$ for any chosen $k \geq 1$. This time, Corollary 14 suggests to look for a matrix for which $d \mapsto \max M^d$ has growth rate $\Theta(p^k)$. This is satisfied by the matrix illustrated in Figure 5: its size is k , and it has entries 1 along and immediately above the diagonal, and entries 0 everywhere else. A solution to the problem described by this matrix has the following form: a path from a leaf to the root is labeled l_0 including both ends. From each vertex labeled l_0 there is a path labeled l_1 from another leaf, and so on until each vertex labeled l_{k-2} being the endpoint of a path labeled l_{k-1} from a leaf.

5 Conclusions and discussion

In this work, we have defined the mending volume and shown that it is able to distinguish more complexity classes of problems than the mending radius from [9], which is the most closely related previously studied notion.

Cost of reading vs. cost of writing. Observe that the mending volume as defined here does not yet determine the size of the neighborhood that needs to be *explored* in order to decide what to change. Indeed the definition of the mending volume as the optimum over all possible mends implies that in order to compute a volume-optimal mend one would need to explore all possible mends and thus would have to gather information on the entire neighborhood up to the mending radius. This suggests that the mending volume is advantageous primarily in situations in which the cost of gathering information is negligible compared to the cost of actually performing the modifications.

In the full version of this work we also explore *algorithmic* versions of mending volume, which capture the idea of how many nodes need to be explored in order to find a mend.

Patching one hole vs. patching all holes. While our measure guarantees optimality of a single mend at a specific vertex, it does not guarantee optimality of the final solution when executing several mends sequentially. It also does not prevent the fact that some labels

might be overwritten repeatedly each time a new mend is computed to patch a different hole. Compare this to [26] for example, which provides strong guarantees of monotonicity (each node changes label at most once) and goal optimality (the final solution is the nearest one in terms of Hamming distance).

This shortcoming is not unexpected: a closer look at the problems we study here reveals that they fall under Condition 1 of Theorem 1 of [26]: the optimal configuration can depend on the initial configuration at an arbitrary distance of the vertex of interest. This means that finding the optimal solution could require global information. Thus, the kind of problems that the mending volume is best at classifying is precisely a class of problems to which the results of [26] do not apply.

Open questions. Although our encoding from propagation problems to general graphs in Section 4.3 makes our existential results applicable to the case of general trees and general graphs, the impossibility results do not work analogously. One major unsolved question is whether there are problems that have a mending volume strictly between polylogarithmic and polynomial, either in general trees or in general graphs.

References

- 1 Emile Aarts and Jan Karel Lenstra, editors. *Local Search in Combinatorial Optimization*. Princeton University Press, 2003. doi:10.2307/j.ctv346t9c.
- 2 Yehuda Afek, Shay Kutten, and Moti Yung. Memory-efficient self stabilizing protocols for general networks. In Jan van Leeuwen and Nicola Santoro, editors, *Distributed Algorithms*, pages 15–28, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.
- 3 Eric Angel. *A Survey of Approximation Results for Local Search Algorithms*, pages 30–73. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. doi:10.1007/11671541_2.
- 4 Esther M. Arkin and Refael Hassin. On Local Search for Weighted k-Set Packing. *Mathematics of Operations Research*, 23(3):640–648, 1998. doi:10.1287/moor.23.3.640.
- 5 Giorgio Ausiello and Marco Protasi. Local search, reducibility and approximability of NP-optimization problems. *Information Processing Letters*, 54(2):73–79, 1995. doi:10.1016/0020-0190(95)00006-X.
- 6 Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-stabilization by local checking and correction (extended abstract). In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science, SFCS '91*, pages 268–277, USA, 1991. IEEE Computer Society. doi:10.1109/SFCS.1991.185378.
- 7 Baruch Awerbuch, Boaz Patt-Shamir, George Varghese, and Shlomi Dolev. Self-stabilization by local checking and global reset. In Gerard Tel and Paul Vitányi, editors, *Distributed Algorithms*, pages 326–339, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.
- 8 Vineet Bafna, Babu Narayanan, and R. Ravi. Nonoverlapping local alignments (weighted independent sets of axis-parallel rectangles). *Discrete Applied Mathematics*, 71(1):41–53, 1996. doi:10.1016/S0166-218X(96)00063-7.
- 9 Alkida Balliu, Juho Hirvonen, Darya Melnyk, Dennis Olivetti, Joel Rybicki, and Jukka Suomela. Local Mending. In *Structural Information and Communication Complexity*, 2022. doi:10.1007/978-3-031-09993-9_1.
- 10 Joffroy Beauquier, Sylvie Delaët, Shlomi Dolev, and Sébastien Tixeuil. Transient fault detectors. *Distrib. Comput.*, 20(1):39–51, July 2007. doi:10.1007/s00446-007-0029-x.
- 11 Sebastian Brandt, Christoph Grunau, and Václav Rozhoň. The Randomized Local Computation Complexity of the Lovász Local Lemma. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, 2021. doi:10.1145/3465084.3467931.
- 12 M. Chams, A. Hertz, and D. de Werra. Some experiments with simulated annealing for coloring graphs. *European Journal of Operational Research*, 32(2):260–266, 1987. Third EURO Summer Institute Special Issue Decision Making in an Uncertain World. doi:10.1016/S0377-2217(87)80148-0.

- 13 Barun Chandra and Magnús M Halldórsson. Greedy Local Improvement and Weighted Set Packing Approximation. *Journal of Algorithms*, 39(2):223–240, 2001. doi:10.1006/jagm.2000.1155.
- 14 Shiri Chechik and Doron Mukhtar. Optimal Distributed Coloring Algorithms for Planar Graphs in the LOCAL Model. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, 2019.
- 15 Johanne Cohen, Laurence Pilard, Mikaël Rabie, and Jonas Sénizergues. Making Self-Stabilizing any Locally Greedy Problem, 2022. doi:10.48550/arXiv.2208.14700.
- 16 Philippe Galinier and Alain Hertz. A survey of local search methods for graph coloring. *Computers & Operations Research*, 33(9):2547–2562, 2006. Part Special Issue: Anniversary Focused Issue of Computers & Operations Research on Tabu Search. doi:10.1016/j.cor.2005.07.028.
- 17 Magnús M. Halldórsson. Approximating Discrete Collections via Local Improvements. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '95, pages 160–169, 1995. doi:10.5555/313651.313687.
- 18 David G. Harris, Hsin-Hao Su, and Hoa T. Vu. On the Locality of Nash-Williams Forest Decomposition and Star-Forest Decomposition. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, 2021. doi:10.1145/3465084.3467908.
- 19 David S. Johnson, Christos H. Papadimitriou, and Mihalis Yannakakis. How easy is local search? *Journal of Computer and System Sciences*, 37(1):79–100, 1988. doi:10.1016/0022-0000(88)90046-3.
- 20 Michael König and Roger Wattenhofer. On Local Fixing. In *Principles of Distributed Systems*, 2013. doi:10.1007/978-3-319-03850-6_14.
- 21 S. Kutten and D. Peleg. Tight fault locality. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, 1995. doi:10.1109/SFCS.1995.492672.
- 22 Moni Naor and Larry Stockmeyer. What Can be Computed Locally? *SIAM Journal on Computing*, 24(6):1259–1277, 1995. doi:10.1137/S0097539793254571.
- 23 Alessandro Panconesi and Aravind Srinivasan. The local nature of Δ -coloring and its algorithmic applications. *Combinatorica*, 15(2):255–280, 1995. doi:10.1007/BF01200759.
- 24 Silvia Richter, Malte Helmert, and Charles Gretton. A Stochastic Local Search Approach to Vertex Cover. In *KI 2007: Advances in Artificial Intelligence*, pages 412–426, 2007. doi:10.1007/978-3-540-74565-5_31.
- 25 Will Rosenbaum and Jukka Suomela. Seeing Far vs. Seeing Wide: Volume Complexity of Local Graph Problems. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, PODC '20, 2020. doi:10.1145/3382734.3405721.
- 26 Yukiko Yamauchi and Sébastien Tixeuil. Monotonic stabilization. In Chenyang Lu, Toshimitsu Masuzawa, and Mohamed Mosbah, editors, *Principles of Distributed Systems*, pages 475–490, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

The Impossibility of Approximate Agreement on a Larger Class of Graphs

Shihao Liu ✉

Department of Computer Science, University of Toronto, Canada

Abstract

Approximate agreement is a variant of consensus in which processes receive input values from a domain and must output values in that domain that are sufficiently close to one another. We study the problem when the input domain is the vertex set of a connected graph. In asynchronous systems where processes communicate using shared registers, there are wait-free approximate agreement algorithms when the graph is a path or a tree, but not when the graph is a cycle of length at least 4. For many graphs, it is unknown whether a wait-free solution for approximate agreement exists.

We introduce a set of impossibility conditions and prove that approximate agreement on graphs satisfying these conditions cannot be solved in a wait-free manner. In particular, the graphs of all triangulated d -dimensional spheres that are not cliques, satisfy these conditions. The vertices and edges of an octahedron is an example of such a graph. We also present a family of reductions from approximate agreement on one graph to another graph. This allows us to extend known impossibility results to even more graphs.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Theory of computation → Computability

Keywords and phrases Approximate agreement on graph, wait-free solvability, triangulated sphere

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2022.22

Funding This work is supported by the Natural Science and Engineering Research Council of Canada.

Acknowledgements I want to thank my advisor Faith Ellen for her advice and many helpful discussions. I would also like to thank the anonymous reviewers for their helpful comments.

1 Introduction

Agreement problems have been extensively studied in the field of distributed computing. In particular, the *consensus* problem [19] requires that processes agree on a single input value. When the system is asynchronous and processes may crash, Fischer, Lynch, and Paterson [12] showed that consensus is unsolvable. Since then, many variants of consensus with milder agreement requirements have been studied in asynchronous systems.

One such variant is *approximate agreement*, where instead of agreeing on a single value, processes must output values that are sufficiently close to one another. This problem was introduced by Dolev, Lynch, Pinter, Stark and Weihl [11] and is related to synchronizing clocks in a distributed system. Attiya, Lynch, and Shavit [8] considered the approximate agreement problem where the domain is \mathbb{R} and processes are required to output values that are within distance ε of one another. They showed that this problem has step complexity $\Theta(\log n)$ using single-writer registers in the asynchronous shared-memory setting. Their bound does not depend on ε nor the size of the input domain. Using multi-writer registers, Schenk [21] showed that this problem has step complexity $O(\log(M/\varepsilon))$, where M is the largest magnitude of any input value. Here, the complexity does not depend on the number of processes in the system. Mendes, Herlihy, Vaidya and Garg [17] considered approximate agreement on \mathbb{R}^d . They showed that this problem has a solution that tolerates up to f Byzantine failures in an asynchronous completely-connected message-passing system if



© Shihao Liu;

licensed under Creative Commons License CC-BY 4.0

26th International Conference on Principles of Distributed Systems (OPODIS 2022).

Editors: Eshcar Hillel, Roberto Palmieri, and Etienne Rivière; Article No. 22; pp. 22:1–22:20

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

22:2 The Impossibility of Approximate Agreement on a Larger Class of Graphs

and only if $n > \max\{3f, (d+1)f\}$. More recently, Attiya and Ellen [7] gave a wait-free algorithm with $O(\log n(\log n + \log(S/\varepsilon)))$ step complexity solving approximate agreement on \mathbb{R}^d using multi-writer registers, where S is the maximum distance between any two inputs. They also proved two lower bounds for this problem, $\Omega\left(\min\left\{\frac{\log M}{\log \log M}, \frac{\sqrt{\log n}}{\log \log n}\right\}\right)$ and $\frac{1}{2} \log_{\sqrt{2}+1}(S/\varepsilon)$.

Approximate agreement on graphs is the natural discrete variant of approximate agreement. Here the domain is the vertex set of some connected graph. Herlihy, Kozlov, and Rajsbaum [13] viewed approximate agreement on graphs as robot convergence tasks.

Approximate agreement on a path is equivalent to approximate agreement on a closed interval in \mathbb{R} . For some other graphs, such as trees, there are wait-free algorithms to solve approximate agreement using registers [1]. For other graphs, such as cycles of length at least 4, no wait-free algorithms using only registers exist [3, 10]. However, for many graphs it is unknown whether a wait-free solution for approximate agreement exists using only registers.

Approximate agreement on graphs

On a connected undirected graph $G = (V, E)$ (known to all processes), each process p_i begins the *approximate agreement* problem with an input value $x_i \in V$. At the end of the computation, each process outputs a value $y_i \in V$ such that the following two conditions are satisfied:

- agreement: different output values are adjacent in G , and
- (clique) validity: if the inputs form a clique in G , then the set of outputs is a subset of the set of inputs.

Different validity conditions for approximate agreement on graphs have been considered. The *shortest path validity* condition [3] requires each output value to lie on some shortest path between two input values. It ensures that each output is a vertex in the smallest convex subgraph containing all input vertices. A subgraph $H \subseteq G$ is *convex* if, for any two vertices u and v in H , all shortest paths (in G) between u and v are contained in H . The *minimal path validity* condition [18] requires output values to lie on some chordless path between two input values. Since every shortest path is also a chordless path, minimal path validity generalizes shortest path validity. The *clique validity* condition [1] is also known as *1-gathering validity*, it is a generalization of both minimal path validity and shortest path validity. Thus, a proof that there is no wait-free algorithm for approximate agreement on a graph with clique validity implies that there is no wait-free algorithm for approximate agreement on that graph with the other two validity conditions. Likewise, an algorithm solving approximate agreement on a graph with shortest path validity also guarantees minimal path validity and clique validity.

Positive results

Alistarh, Ellen, and Rybicki [3] showed that approximate agreement with shortest path validity has a wait-free solution on any graph of radius one or any *nicey bridged* graph in a shared memory system where processes communicate using registers. The family of nicely bridged graphs includes all chordal graphs. They also presented an approximate agreement algorithm on any connected graph that tolerates at most one process crash failure in the same model. This shows that approximate agreement has a wait-free solution among two processes on any connected graph.

Nowak and Rybicki [18] gave an approximate agreement algorithm on chordal graphs in an asynchronous message-passing model with at most f Byzantine processes, provided the number of processes is greater than $(\omega(G) + 1)f$, where $\omega(G)$ is the size of the largest clique in G . However, their algorithm only guarantees minimal path validity.

With clique validity, Alcántara, Castañeda, Flores-Peñaloza, and Rajsbaum [1] showed that approximate agreement has a wait-free solution using only registers when the graph G is a tree or, more generally, when the *clique graph* of G is a tree. The clique graph of G is the graph (V', E') , where vertices in V' are cliques in the graph G , and $(v', u') \in E'$ if and only if $v' \cap u' \neq \emptyset$ in the graph G .

Negative results

Approximate agreement has no wait-free solution using only registers on a cycle of length at least 4. Castañeda, Rajsbaum, and Roy [10] showed this by giving a reduction from 2-set agreement. Later, Alistarh, Ellen, and Rybicki [3] gave a direct combinatorial proof using Sperner's lemma. They also used a reduction to show that approximate agreement has no wait-free solution on a graph G , if the vertices of G can be labelled using the set $\{0, 1, 2\}$, such that

- G contains no triangle with three different labels, and
- G contains a cycle C in which exactly one node has label 1 and its two neighbours in C have labels 0 and 2.

We call such a labelling an *AER impossibility labelling*. In particular, any cycle of length $c \geq 4$ has an AER impossibility labelling.

Ledent [16] conjectured that approximate agreement is not solvable in a wait-free manner on any graph whose complex of cliques is not contractible. This includes graphs consisting of the nodes and edges of an octahedron and an icosahedron and, more generally, any triangulated d -dimensional sphere, for $d \geq 1$, that is not a clique. Note that a triangulated 1-dimensional sphere is a cycle and a triangulated 2-dimensional sphere is a connected planar graph in which every edge is shared by exactly two triangles.

Our contribution

In Section 4.1, we show that approximate agreement on the octahedron graph has no wait-free solution using only registers, for $n \geq 4$ processes. In Section 4.2, we extend this result to any graph that satisfies a new set of impossibility conditions provided there are sufficiently many processes. Any cycle of length at least 4 satisfies these impossibility conditions. More generally, these impossibility conditions are satisfied by any graph (except a clique) consisting of the nodes and edges of a triangulated d -dimensional sphere, for $d \geq 1$. This includes the octahedron graph, which we show does not have an AER impossibility labelling.

In Section 5, we describe a simple reduction from approximate agreement on one graph to approximate agreement on another graph. As an application of this reduction, we show that the impossibility of wait-free approximate agreement on the stellated octahedron graph can be derived from the impossibility of wait-free approximate agreement on the octahedron graph.

Alistarh, Ellen and Rybicki [4] showed that extension-based proofs cannot be used to prove the impossibility of approximate agreement on a 4-cycle. In Section 6, we briefly discuss our generalization showing that extension-based proofs cannot be used to prove the impossibility of wait-free approximate agreement on any connected graph.

2 Iterated immediate snapshot model

We focus our attention on computation in the full-information (non-uniform) iterated immediate snapshot model [15].

In this model, a set P of n processes communicate by accessing an infinite sequence of shared single-writer atomic snapshot objects. Each *single-writer atomic snapshot* object has n components and supports two atomic operations, **update** and **scan**. Initially, each component of each snapshot object contains the value $-$. An **update**(x) performed by process p_i on a snapshot object changes the value of its i -th component to x , while a **scan** returns the current value of each component.

At any given time, the *state* of a process p_i consists of its process identifier, its current *view* of the system, and a bit indicating whether it has just performed an update or a scan. Initially, the view of process p_i is just its input value and p_i is poised to access the first snapshot object. Each process accesses each snapshot object in the sequence at most twice. The first time process p_i accesses a snapshot object, it performs an **update**. At its next step, p_i performs a **scan** on the same snapshot object and changes its state depending on the result it received from the **scan**. In the full-information setting, whenever process p_i performs an **update**, it writes its entire computation history into the i -th component of the snapshot object, and whenever it performs a **scan**, it changes its view to be the result of the scan. After changing its state, process p_i applies a *decision map* δ to its current state. If the state of p_i is mapped to \perp , then p_i is poised to access the next snapshot object in the sequence. Otherwise, the state of p_i is mapped to an output value y , which p_i outputs, and p_i cannot take any more steps. A *protocol* is specified by the decision map δ used by every process. A protocol is *wait-free* if each process takes a finite number of its own steps before its state is mapped to an output value by δ .

A *configuration* consists of the state of each process. Note that, in the full-information setting, we can determine the contents of the snapshot objects from the states of every process. An *initial configuration* is a configuration where processes are in their initial states (and the snapshot objects have their initial values). A process is *active* in a configuration if δ maps the state of that process to \perp . Likewise, a process is *terminated* in a configuration if δ maps the state of that process to an output value. A *terminal configuration* is a configuration where all processes are terminated.

An *execution* from a configuration C is defined by an alternating sequence $C_0, Q_1, C_1, Q_2, C_2, \dots$ of configurations and subsets of processes, beginning with the configuration $C_0 = C$, such that, for each $k \geq 0$, Q_{k+1} is a non-empty subset of processes that are poised to access the same snapshot object in configuration C_k . Configuration C_{k+1} is the result of the processes in Q_{k+1} each performing an **update**, and then each performing a **scan**, starting from configuration C_k . Each execution from C induces a *schedule* from C , which is the sequence Q_1, Q_2, \dots of subsets of processes in the execution. Since each process only updates its corresponding component of each snapshot object, an execution is completely specified by its starting configuration and its schedule. If α is a finite schedule from configuration C , we use $C\alpha$ to denote the configuration at the end of the execution that induces α . In this case, we say that $C\alpha$ is *reachable* from C via the schedule α . Note that if C is a terminal configuration, then the empty schedule is the only possible schedule from C .

An execution is *Q-only* if its schedule consists of subsets of processes in Q . The schedule of a Q -only execution is called a Q -only schedule. If each active process in Q is poised to perform **update** on the same snapshot object in C , then a *1-round Q-only schedule* from C is a Q -only schedule where each active process in Q appears exactly once. When $Q = P$, we simply call this schedule a *1-round schedule*. From a configuration where all active processes are poised to perform **update** on the same snapshot object, every 1-round Q -only schedule can be extended to a 1-round schedule by appending a 1-round $(P \setminus Q)$ -only schedule. The resulting schedule is called a *1-round Q-first schedule* (i.e. all occurrences of processes in Q

occur before any occurrence of a process in $P \setminus Q$). For $k \geq 1$, a k -round Q -first schedule β starting from C is a sequence of 1-round Q -first schedules β_1, \dots, β_k , where β_1 starts from C and β_i starts from $C\beta_1 \dots \beta_{i-1}$, for $2 \leq i \leq k$. Note that each terminal configuration reachable from an initial configuration C_0 is also reachable from C_0 via a k -round schedule, for some $k \geq 0$. (See Lemma 4.4 from [2].) Thus, it suffices to only consider such schedules.

Let C be a configuration reachable from some initial configuration via a k -round Q -first schedule, where $\emptyset \neq Q \subseteq P$. Then the *partial configuration* C' of C induced by Q consists of the of states of the processes in Q . We use $\pi(C')$ to denote the set of processes Q whose states appear in the partial configuration C' . The partial configuration C' can be viewed as a configuration in a system with a smaller set of processes: Suppose β is a k -round Q -first schedule starting from configuration C . Let β' be the restriction of β to the processes in Q . Then β' is a schedule starting from the partial configuration C' of C induced by Q and each process in Q has the same state in $C'\beta'$ and $C\beta$. Note that, for each process in Q , components corresponding to processes in $P \setminus Q$ all have value $-$ in both $C'\beta'$ and $C\beta$.

Two (partial) configurations C and C' are *indistinguishable to a set of processes* Q if $Q \subseteq \pi(C) \cap \pi(C')$ and the state of each process in Q is the same in both configurations. Consider any 1-round Q -first schedule β from both C and C' . If C and C' are indistinguishable to Q , then $C\beta$ and $C'\beta$ are indistinguishable to Q . Since we are restricting attention to full-information protocols, the converse is also true. It follows that, if $C\beta = C'\beta$, then $\beta = \beta'$.

If \mathbb{K} is a collection of (partial) configurations (not necessarily induced by the same set of processes), $C \in \mathbb{K}$ is a (partial) configuration, and $\emptyset \subseteq Q \subseteq \pi(C)$, then we say Q *identifies* C in \mathbb{K} if, for every other (partial) configuration $C' \in \mathbb{K}$ such that $Q \subseteq \pi(C')$, at least one process in Q has a different state in C and C' . When the collection \mathbb{K} is clear from context, we simply say Q identifies C .

For any $r \geq 0$ and for any (partial) configuration C reachable from some (partial) initial configuration via some r -round $\pi(C)$ -only schedule, let $\chi(C, \delta)$ denote the set of all possible (partial) configurations reachable via 1-round $\pi(C)$ -only schedules starting from C and let $\chi^k(C, \delta)$ denote the set of all possible (partial) configurations reachable via k -round $\pi(C)$ -only schedules starting from C . For a collection \mathbb{K} of (partial) configurations, define $\chi(\mathbb{K}, \delta) = \bigcup_{C \in \mathbb{K}} \chi(C, \delta)$ and $\chi^k(\mathbb{K}, \delta) = \bigcup_{C \in \mathbb{K}} \chi^k(C, \delta)$.

3 A Computational Version of Sperner's Lemma

Our main results in Section 4 rely on a variant of a classical combinatorial tool known as Sperner's lemma. The original topological proofs for the impossibility of wait-free set agreement [14, 20, 9] all used Sperner's lemma or equivalent formulations. Later, Attiya and Castañeda [5] proved the impossibility of set agreement using purely combinatorial techniques, without using topology. Their argument implicitly applied elements of Sperner's lemma directly on executions. More recently, Alistarh, Ellen, and Rybicki [3] gave a combinatorial proof for the impossibility of approximate agreement on cycles (of length at least 4) using a generalization of Sperner's lemma to convex polygons.

In this section, we generalize Sperner's lemma and rephrase it as a self-contained statement about executions in the iterated immediate snapshot model. It makes no explicit mention of topology. However, we note that it is equivalent to a formulation of Sperner's lemma for manifolds, phrased in terms of simplicial complexes and subdivisions, that appears in [13] as Lemma 9.3.4.

Consider a protocol among $n \geq 2$ processes in the iterated immediate snapshot model. Let $2 \leq m \leq n$ and let \mathbb{H} be a collection of (partial) initial configurations such that $|\pi(C)| = m$ for each $C \in \mathbb{H}$. For each $C \in \mathbb{H}$ and each subset of processes $Q \subseteq \pi(C)$, denote by $I(C, Q)$

the set of input values of processes in Q in the (partial) configuration C . The *boundary* of \mathbb{H} , denoted $\mathcal{B}(\mathbb{H})$, is the collection of all pairs (C, Q) , where $Q \subsetneq \pi(C)$ is a subset of $m - 1$ processes and C is identified by Q in \mathbb{H} . In other words, for each pair $(C, Q) \in \mathcal{B}(\mathbb{H})$, there is no other (partial) configuration $C' \in \mathbb{H}$ such that $Q \subsetneq \pi(C')$ and each process in Q has the same state in C and C' .

Throughout the remainder of this section, we let t be the maximum number of non-empty rounds taken by the protocol in executions starting from (partial) initial configurations in \mathbb{H} . Let $\mathbb{T} = \chi^t(\mathbb{H}, \delta)$ be the collection of all (partial) terminal configurations reachable from (partial) initial configurations in \mathbb{H} .

Definition 1 is a computational analogue of a Sperner labelling. Immediately afterwards, we give an example with $n = m = 3$ to help explain the definition.

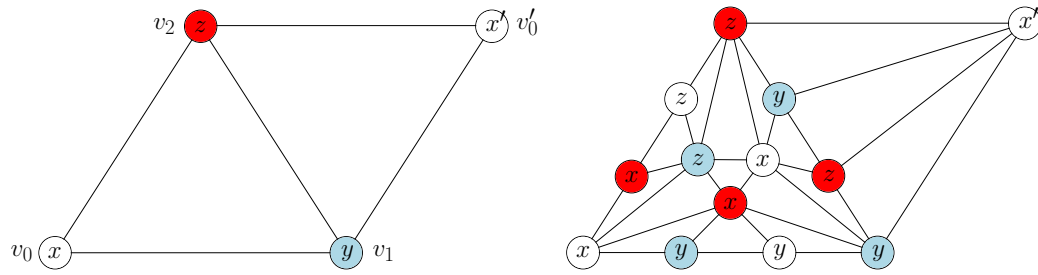
► **Definition 1.** *A collection \mathbb{H} of (partial) initial configurations, each of which consists of the states of the same number of processes $m \geq 2$, satisfies the computational Sperner conditions (CSC) for the protocol if:*

- *CSC1: For each $(C, Q) \in \mathcal{B}(\mathbb{H})$, the processes in Q have different input values in C . In other words, $|I(C, Q)| = m - 1$.*
- *CSC2: For each $(C, Q) \in \mathcal{B}(\mathbb{H})$, there are an odd number of pairs $(C', Q') \in \mathcal{B}(\mathbb{H})$ such that $I(C', Q') = I(C, Q)$.*
- *CSC3: For each $(C, Q) \in \mathcal{B}(\mathbb{H})$, for each subset $S \subseteq Q$, and for each S -first $\pi(C)$ -only t -round schedule β starting from C , each process in S has output a value in $I(C, S)$ in the (partial) configuration $C\beta$.*
- *CSC4: For any $C \in \mathbb{H}$ and any subset $Q \subsetneq \pi(C)$ of $m - 1$ processes, there is at most one other configuration $C' \in \mathbb{H}$ such that $Q \subsetneq \pi(C')$ and the configurations C and C' are indistinguishable to all processes in Q .*

Let \mathbb{H} be a collection consisting of two initial configurations C and C' , where $\pi(C) = \pi(C') = \{p_0, p_1, p_2\}$ and C and C' are indistinguishable only to processes p_1 and p_2 . Since p_0 has a different initial state in C and C' , any subset of $\{p_0, p_1, p_2\}$ containing p_0 identifies C and C' in \mathbb{H} . Thus, $\mathcal{B}(\mathbb{H}) = \{(C, \{p_0, p_1\}), (C, \{p_0, p_2\}), (C', \{p_0, p_1\}), (C', \{p_0, p_2\})\}$. Let $x \neq x'$ be the input of process p_0 in configurations C and C' . Let v_0 and v'_0 be its initial states in these two configurations. Let y and z be the inputs of processes p_1 and p_2 in both configurations and let v_1 and v_2 be their initial states. This is illustrated in Figure 1a, where the colors white, blue, and red represent the processes p_0 , p_1 , and p_2 , respectively. The configurations $C = \{v_0, v_1, v_2\}$ and $C' = \{v'_0, v_1, v_2\}$ are the two triangles. Since C and C' are indistinguishable to p_1 and p_2 , these triangles share the edge $\{v_1, v_2\}$. The edges $\{v_0, v_1\}$ and $\{v_0, v_2\}$ in C and the edges $\{v_2, v'_0\}$ and $\{v_1, v'_0\}$ in C' form the boundary of this polygon.

CSC1 says that every edge on the boundary of the polygon has endpoints with different inputs, so, in the example, x, x', y , and z are all different. CSC2 says that each pair of inputs that labels an edge on the boundary labels an odd number of such edges. In the example, each such pair labels exactly one edge on the boundary. CSC4 is a technical requirement (analogous to the pseudomanifold property of a simplicial complex) that says each edge occurs in at most two triangles.

The set of partial terminal configurations \mathbb{T} reachable from C and C' by a protocol in the iterated immediate snapshot model can be represented by a subdivision of the two triangles [15]. This is illustrated in Figure 1b, where each triangle in the subdivision represents a reachable terminal configuration. The subdivision of a vertex v_i , which is just a vertex, corresponds to the $\{p_i\}$ -only execution from v_i . The subdivision of an edge $\{v_i, v_j\}$



(a) Triangles representing configurations C and C' . (b) A subdivision of the two triangles representing terminal configurations reachable from C and C' by a protocol.

corresponds to the $\{p_i, p_j\}$ -only executions from the partial configuration $\{v_i, v_j\}$. Here, the output of a process in a terminal configuration labels the vertex corresponding to its state in this configuration. CSC3 requires this labelling to be a *Sperner labelling*: The subdivision of each vertex is labelled by the input of the vertex and the vertices of the subdivision of a boundary edge are each labelled by the input of an endpoint of the edge.

Informally, Theorem 2 says that if the collection \mathbb{H} satisfies the computational Sperner conditions, then in at least one (partial) terminal configuration in \mathbb{T} , each process outputs a different value.

► **Theorem 2.** *Let \mathbb{H} be a collection of (partial) initial configurations that satisfies the computational Sperner conditions for some protocol and let $m \geq 2$ be the number of processes represented by each (partial) configuration in \mathbb{H} . For each $(C, Q) \in \mathcal{B}(\mathbb{H})$, let $\mathbb{T}(C, Q)$ be the set of all (partial) terminal configurations T reachable from configurations in \mathbb{H} in the iterated immediate snapshot model such that*

- *m different values are output by the m processes in T and,*
- *each value in $I(C, Q)$ is output by some process in T .*

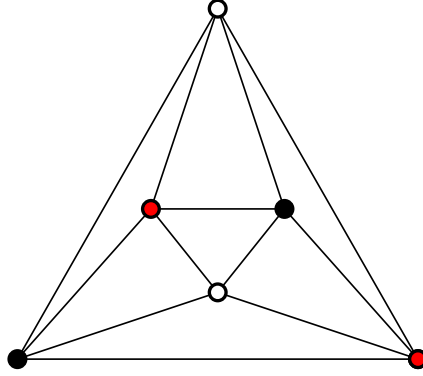
Then $|\mathbb{T}(C, Q)|$ is odd and, hence, $|\mathbb{T}(C, Q)| \geq 1$.

The proof of Theorem 2 is deferred to the appendix. In the next section, we demonstrate how it could be easily applied to obtain impossibility results for wait-free computation in the iterated immediate snapshot model.

4 New Impossibility Results from Sperner's Lemma

We first look at approximate agreement on the octahedron graph. We show that there is no wait-free algorithm when the number of processes n is at least 4. We then extend our result to a larger class of graphs.

To help with the presentation, we recall some standard notions in graph theory. A *vertex coloring* is an assignment of colors (or labels) to each vertex of a graph such that no edge has endpoints with the same color. A *k -coloring* is a vertex coloring that uses at most k different colors. A graph is *k -colorable* if it has a *k -coloring*. The *chromatic number* of a graph is the smallest number k such that the graph is *k -colorable*. The *clique number* of a graph G , denoted $\omega(G)$, is the size of its largest clique. Note that the chromatic number of G is always at least as large as $\omega(G)$.



■ **Figure 2** A 3-coloring of the octahedron graph.

4.1 The impossibility of approximate agreement on the octahedron graph

The octahedron graph (Figure 2) is obtained by taking the vertices and edges of an octahedron. It consists of a set of six vertices, $V = \{a_1, \dots, a_6\}$, and a set of 12 edges. It is 3-colorable and its largest clique has size 3. To show that approximate agreement is unsolvable on the octahedron graph, it is sufficient to show that any protocol has an execution in which four processes output different values.

Consider a system with 4 processes p_0, p_1, p_2, p_3 and a 3-coloring of the octahedron graph, where we use $\{p_1, p_2, p_3\}$ as colors. We use $p(a_i)$ to denote the color of vertex a_i . For each triangle $\{a_i, a_j, a_k\}$ in the octahedron graph, let $C_{\{i,j,k\}}$ be the initial configuration where process $p(a_i)$ has input a_i , $p(a_j)$ has input a_j , $p(a_k)$ has input a_k , and p_0 has input a_1 . Let \mathbb{H} be the collection of all such configurations. (The choice of input vertex a_1 for process p_0 is arbitrary, but it has to be the same for all configurations in \mathbb{H} .)

The following two observations allow us to determine what pairs are in $\mathcal{B}(\mathbb{H})$.

► **Observation 3.** *Every edge in the octahedron graph is shared by exactly two triangles. If $\{a_i, a_j, a_k\}$ and $\{a_i, a_j, a_{k'}\}$ are both triangles in the octahedron graph, then configurations $C_{\{i,j,k\}}$ and $C_{\{i,j,k'\}}$ are indistinguishable to the processes in $\{p(a_i), p(a_j), p_0\}$. Since the edge $\{a_i, a_j\}$ is only shared by the two triangles $\{a_i, a_j, a_k\}$ and $\{a_i, a_j, a_{k'}\}$, there is no other configuration in \mathbb{H} that is indistinguishable from configuration $C_{\{i,j,k\}}$ to the set of processes $\{p(a_i), p(a_j), p_0\}$.*

► **Observation 4.** *Each configuration $C_{\{i,j,k\}} \in \mathbb{H}$ is identified by the set $\{p_1, p_2, p_3\}$. In other words, in any other configuration $C_{\{i',j',k'\}} \in \mathbb{H}$, there is at least one process in $\{p_1, p_2, p_3\}$ that has a different input in $C_{\{i',j',k'\}}$ and in $C_{\{i,j,k\}}$.*

Observation 4 implies that $(C_{\{i,j,k\}}, \{p_1, p_2, p_3\}) \in \mathcal{B}(\mathbb{H})$ for all $C_{\{i,j,k\}} \in \mathbb{H}$. Observation 3 implies that $\{p_1, p_2, p_3\}$ is the only set of 3 processes that identifies $C_{\{i,j,k\}}$ in \mathbb{H} . Thus, $\mathcal{B}(\mathbb{H})$ contains only these pairs.

► **Lemma 5.** *For any wait-free protocol that solves approximate agreement on the octahedron graph, \mathbb{H} satisfies the computational Sperner conditions.*

Proof. We prove that \mathbb{H} satisfies the four conditions:

- **CSC1:** For any $(C_{\{i,j,k\}}, \{p_1, p_2, p_3\}) \in \mathcal{B}(\mathbb{H})$, processes p_1, p_2, p_3 received input values a_i, a_j, a_k in some order in configuration $C_{\{i,j,k\}}$. Hence, the three processes received three different input values.

- CSC2: Consider any two distinct pairs $(C_{\{i,j,k\}}, \{p_1, p_2, p_3\}), (C_{\{i',j',k'\}}, \{p_1, p_2, p_3\}) \in \mathcal{B}(\mathbb{H})$. Then, by definition, processes p_1, p_2, p_3 in the two configurations received vertices from two different triangles $\{a_i, a_j, a_k\}$ and $\{a_{i'}, a_{j'}, a_{k'}\}$ as input values (in some order). Hence, the set of input values $I(C, Q)$ is different for each pair $(C, Q) \in \mathcal{B}(\mathbb{H})$.
- CSC3: For any $(C_{\{i,j,k\}}, \{p_1, p_2, p_3\}) \in \mathcal{B}(\mathbb{H})$ and any subset $S \subseteq \{p_1, p_2, p_3\}$, the input values received by processes in S in configuration $C_{\{i,j,k\}}$ is a subset of the triangle $\{a_i, a_j, a_k\}$. Hence, in any protocol that solves approximate agreement on the octahedron graph, the clique validity condition ensures that the outputs of processes in S in any S -first execution starting from $C_{\{i,j,k\}}$ is a subset of the input values received by processes in S in configuration $C_{\{i,j,k\}}$.
- CSC4: Consider any $C_{\{i,j,k\}} \in \mathbb{H}$ and any 3 processes subset $Q \subsetneq \{p_0, p_1, p_2, p_3\}$. If $Q = \{p_1, p_2, p_3\}$, then Q identifies $C_{i,j,k}$ in \mathbb{H} by Observation 4. In other words, no configuration in \mathbb{H} is indistinguishable from $C_{i,j,k}$ to the set of processes $\{p_1, p_2, p_3\}$. Otherwise, without loss of generality, suppose $Q = \{p(a_i), p(a_j), p_0\}$. By Observation 3, the triangle $\{a_i, a_j, a_k\}$ shares the edge $\{a_i, a_j\}$ with exactly one other triangle $\{a_i, a_j, a_{k'}\}$ in octahedron graph, and $C_{\{i,j,k'\}}$ is the only configuration in \mathbb{H} that is indistinguishable from $C_{\{i,j,k\}}$ to the set of processes $\{p(a_i), p(a_j), p_0\}$. ◀

► **Theorem 6.** *There is no wait-free protocol among 4 processes in the iterated immediate snapshot model that solves the approximate agreement problem on the octahedron graph.*

Proof. Consider a protocol that claims to solve approximate agreement on the octahedron graph. By Lemma 5, \mathbb{H} satisfies the computational Sperner conditions. Since $\mathcal{B}(\mathbb{H})$ is nonempty, Theorem 2 implies that there exists a terminal configuration in which 4 different vertices are output by p_0, p_1, p_2, p_3 . Since the largest clique in the octahedron has size 3, this contradicts the agreement condition. ◀

4.2 The impossibility of approximate agreement on a larger class of graphs

In this section, we define a class of graphs on which it is impossible to solve wait-free approximate agreement for sufficiently large number of processes. Given a point c in \mathbb{R}^d , a *sphere centered at c* is the set of all points equidistant from c in \mathbb{R}^d . It can be viewed as a subspace of dimension $d-1$ and, hence, is called a $(d-1)$ -dimensional sphere. A *triangulation* of a $(d-1)$ -dimensional sphere is a subdivision of the sphere into $(d-1)$ -dimensional simplices, such that the intersection of any two simplices is either a common face of both simplices or empty. Our class of graphs includes the graph of any triangulated sphere that is not a clique. In particular, a cycle is the graph of a triangulated circle and the octahedron graph is the graph of a triangulated 2-dimensional sphere. We also compare our class of graphs to graphs that admit AER impossibility labellings and show that neither class contains the other.

Our class of graphs is defined by a set of *clique containment conditions*. The fact that every edge in the octahedron graph is shared by exactly two triangles was used in the previous section to show that a certain collection of initial configurations, \mathbb{H} , satisfies the computational Sperner conditions. We generalize this property to require that every clique of size $k-1$ in the graph is contained in exactly two cliques of size k , for some $k \geq 2$. In the proof of Theorem 6, we used Theorem 2 to show the existence an execution in which 4 processes output 4 different values. Since the octahedron graph contains no clique of size 4, this execution violates agreement. More generally, for any k -clique in the graph, we can use Theorem 2 to show the existence of an execution where the k -clique is a strict subset of the outputs. If this k -clique is a maximal clique in the graph, then agreement is violated.

The clique containment conditions

We say a graph G satisfy the *clique containment conditions* if there is a subgraph A of G and an integer k , where $2 \leq k$, such that the following hold:

1. every clique of size $k - 1$ in A is contained in exactly two cliques of size k in A and
2. there is a clique of size k in A that is not contained in any clique of size $k + 1$ in G .

The graph G of any triangulated d -dimensional sphere satisfies the first condition with $A = G$ and $k = d + 1$ and, provided it is not a clique of size $k + 2$, G also satisfies the second condition. In particular, the octahedron graph is the graph of a triangulated 2-dimensional sphere. Since it contains no clique of size 4, none of its cliques of size 3 is contained in a clique of size 4.

When $k = 2$, every graph G that satisfies our clique containment conditions also has an AER impossibility labelling. In this case, our first impossibility condition implies that A is a collection of disjoint cycles and our second impossibility condition implies that there exists an edge $\{u, v\}$ in A that is not contained in any triangle in G . Label u with value 1, v with value 2, and all other vertices in G with value 0. This gives an AER impossibility labelling of the graph G .

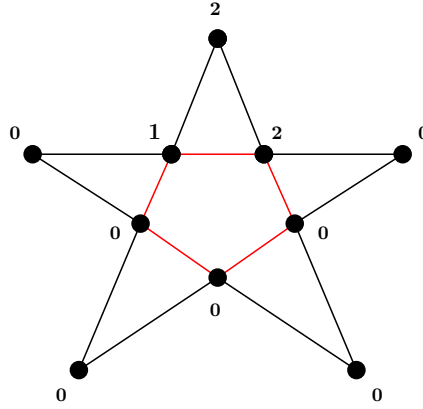
When $k \geq 3$, there are some graphs, in particular the octahedron graph, that satisfy the clique containment conditions, but do not have an AER impossibility labelling. For contradiction, suppose the octahedron graph has an AER impossibility labelling. Then it contains a cycle C of length at least 4 with three consecutive vertices labelled 0, 1, and 2. Since an AER impossibility labelling has no triangle with three different labels, these three vertices do not form a triangle. To finish labelling the rest of the graph so that there is no triangle with three different labels, observe that all other vertices can only receive the label 1. Hence, the cycle C contains at least two different vertices with the label 1, contradicting the definition of AER impossibility labelling.

There are also examples of graphs that have AER impossibility labellings, but do not satisfy the clique containment conditions. For example, consider the graph G' with ten vertices shown in Figure 3, where C is the cycle of length 5 in the middle of G' . Since each edge of G' is contained in some triangle, G' does not satisfy our second impossibility condition when $k = 2$. Since each edge of G' is contained in exactly one triangle, G' does not satisfy our first impossibility condition when $k = 3$. Note that G' has no clique of size greater than 3, hence G' does not satisfy our second impossibility condition when $k > 3$.

Clique containment conditions imply impossibility of approximate agreement

Consider a graph G that satisfies the clique containment conditions. We first construct a collection \mathbb{H} of (partial) initial configurations for any protocol that claims to solve approximate agreement on G . Consider a subgraph $A = (V, E)$ of G and an integer k such that the clique containment conditions are satisfied. Let a_1, \dots, a_ℓ denote the vertices in V , where $\ell = |V|$. Consider a system with n processes p_0, p_1, \dots, p_{n-1} , where n is greater than the chromatic number of A . Consider an $n - 1$ coloring of the graph A , where we use $\{p_1, \dots, p_{n-1}\}$ as colors. We use $p(a_i)$ to denote the color of vertex a_i . For each k -clique $\{a_{i_1}, \dots, a_{i_k}\}$ in the graph A , let $C_{\{i_1, \dots, i_k\}}$ be the (partial) initial configuration consisting of the states of processes $p(a_{i_1}), \dots, p(a_{i_k})$ and p_0 , such that process p_0 has input value a_1 and, for $1 \leq j \leq k$, process $p(a_{i_j})$ has input value a_{i_j} . Let \mathbb{H} be the collection of all such (partial) configurations. As in Section 4.1, the input value assigned to p_0 is not important, as long as it is the same in all (partial) configurations in \mathbb{H} .

The following two observations allow us to determine what pairs are in $\mathcal{B}(\mathbb{H})$.



■ **Figure 3** An AER impossibility labelling of a graph G' with ten vertices. The edges of the cycle C are colored in red.

► **Observation 7.** If $\{a_{i_1}, \dots, a_{i_{k-1}}, a_{i_k}\}$ and $\{a_{i_1}, \dots, a_{i_{k-1}}, a_{i'_k}\}$ are both k -cliques of A , then partial configurations $C_{\{i_1, \dots, i_k\}}$ and $C_{\{i_1, \dots, i'_k\}}$ are indistinguishable to the processes in $\{p(a_{i_1}), \dots, p(a_{i_{k-1}}), p_0\}$. Moreover, $C_{\{i_1, \dots, i'_k\}}$ is the only other partial configuration in \mathbb{H} that is indistinguishable from $C_{\{i_1, \dots, i_k\}}$ to the set of processes $\{p(a_{i_1}), \dots, p(a_{i_{k-1}}), p_0\}$.

The second statement of Observation 7 is true because, the first impossibility condition says that the $(k-1)$ -clique $\{a_{i_1}, \dots, a_{i_{k-1}}\}$ is contained in no other k -cliques of A .

► **Observation 8.** Each partial configuration $C_{\{i_1, \dots, i_k\}}$ is identified by the set of processes $\{p(a_{i_1}), \dots, p(a_{i_k})\}$. In other words, in any other configuration $C_{\{i'_1, \dots, i'_k\}}$, either $\{p(a_{i_1}), \dots, p(a_{i_k})\} \neq \{p(a_{i'_1}), \dots, p(a_{i'_k})\}$, or there is at least one process in $\{p(a_{i_1}), \dots, p(a_{i_k})\}$ that has a different input in $C_{\{i_1, \dots, i_k\}}$ and in $C_{\{i'_1, \dots, i'_k\}}$.

Observation 8 implies that $(C_{\{i_1, \dots, i_k\}}, \{p(a_{i_1}), \dots, p(a_{i_k})\}) \in \mathcal{B}(\mathbb{H})$ for all $C_{\{i_1, \dots, i_k\}} \in \mathbb{H}$. Observation 7 implies that $\{p(a_{i_1}), \dots, p(a_{i_k})\}$ is the only set of size k that identifies $C_{\{i_1, \dots, i_k\}}$ in \mathbb{H} . Thus $\mathcal{B}(\mathbb{H})$ contains only these pairs.

The next lemma is a generalization of Lemma 5, and has a similar proof.

► **Lemma 9.** For any wait-free algorithm that solves approximate agreement on the graph G , \mathbb{H} satisfies the computational Sperner conditions.

Proof. We prove \mathbb{H} satisfies the four conditions:

- **CSC1:** For any $(C_{\{i_1, \dots, i_k\}}, \{p(a_{i_1}), \dots, p(a_{i_k})\}) \in \mathcal{B}(\mathbb{H})$, processes $p(a_{i_1}), \dots, p(a_{i_k})$ received vertices of the k -clique $\{a_{i_1}, \dots, a_{i_k}\}$ as input values in some order. Hence, the k processes received k different input values in $C_{\{i_1, \dots, i_k\}}$.
- **CSC2:** Consider any two distinct pairs $(C_{\{i_1, \dots, i_k\}}, \{p(a_{i_1}), \dots, p(a_{i_k})\}), (C_{\{i'_1, \dots, i'_k\}}, \{p(a_{i'_1}), \dots, p(a_{i'_k})\}) \in \mathcal{B}(\mathbb{H})$. Then, by definition, processes $p(a_{i_1}), \dots, p(a_{i_k})$ in configuration $C_{\{i_1, \dots, i_k\}}$ and processes $p(a_{i'_1}), \dots, p(a_{i'_k})$ in configuration $C_{\{i'_1, \dots, i'_k\}}$ received vertices from two different k -cliques $\{a_{i_1}, \dots, a_{i_k}\}$ and $\{a_{i'_1}, \dots, a_{i'_k}\}$ as input values. Hence, the set of input values $I(C, Q)$ is different for each pair $(C, Q) \in \mathcal{B}(\mathbb{H})$.
- **CSC3:** For any $(C_{\{i_1, \dots, i_k\}}, \{p(a_{i_1}), \dots, p(a_{i_k})\}) \in \mathcal{B}(\mathbb{H})$ and any subset $S \subseteq \{p(a_{i_1}), \dots, p(a_{i_k})\}$, the set of inputs received by processes in S in $C_{\{i_1, \dots, i_k\}}$ is a subset of the k -clique $\{a_{i_1}, \dots, a_{i_k}\}$. Hence, in any protocol that solves approximate agreement on graph G , the clique validity condition ensures that the outputs of processes in S in any S -first execution starting from $C_{\{i_1, \dots, i_k\}}$ is a subset of input values received by processes in S in $C_{\{i_1, \dots, i_k\}}$.

22:12 The Impossibility of Approximate Agreement on a Larger Class of Graphs

- CSC4: Consider any $C_{\{i_1, \dots, i_k\}} \in \mathbb{H}$ and any $(k-1)$ -process subset $Q \subsetneq \pi(C_{\{i_1, \dots, i_k\}}) = \{p(a_{i_1}), \dots, p(a_{i_k}), p_0\}$. If $Q = \{p(a_{i_1}), \dots, p(a_{i_k})\}$, then, by Observation 8, Q identifies $C_{\{i_1, \dots, i_k\}}$ in \mathbb{H} . In other words, for any other partial configuration $C_{\{i'_1, \dots, i'_k\}} \in \mathbb{H}$ such that $Q \subsetneq \pi(C_{\{i'_1, \dots, i'_k\}})$, at least one process in Q has a different state in $C_{\{i_1, \dots, i_k\}}$ and in $C_{\{i'_1, \dots, i'_k\}}$. Otherwise, without loss of generality, suppose $Q = \{p(a_{i_1}), \dots, p(a_{i_{k-1}}), p_0\}$. By Observation 7, the k -clique $\{a_{i_1}, \dots, a_{i_k}\}$ shares the $(k-1)$ -clique $\{a_{i_1}, \dots, a_{i_{k-1}}\}$ with exactly one other k -clique $\{a_{i_1}, \dots, a_{i_{k-1}}, a_{i'_k}\}$. Furthermore, $C_{\{i_1, \dots, i'_k\}}$ is the only configuration in \mathbb{H} such that $Q \subsetneq \pi(C_{\{i_1, \dots, i'_k\}})$ and $C_{\{i_1, \dots, i'_k\}}$ is indistinguishable from $C_{\{i_1, \dots, i_k\}}$ to the set of processes Q . ◀

We are now ready to prove the impossibility of a wait-free solution to the approximate agreement problem on graphs that satisfy the clique containment conditions.

► **Theorem 10.** *Let G be a graph that satisfies the clique containment conditions with subgraph A and integer k . Then there is no wait-free protocol among n processes in the iterated immediate snapshot model that solves approximate agreement on G when n is greater than the chromatic number of A .*

Proof. Consider a protocol that claims to solve approximate agreement on the graph G . Pick a k -clique $\{a_{i_1}, \dots, a_{i_k}\}$ in A that is not contained in any $(k+1)$ -clique in G . By Observation 8, $(C_{\{i_1, \dots, i_k\}}, \{p(a_{i_1}), \dots, p(a_{i_k})\}) \in \mathcal{B}(\mathbb{H})$. Then, by Theorem 2, there exists a partial terminal configuration T in which $k+1$ different values are output, including each value in $\{a_{i_1}, \dots, a_{i_k}\}$. Since the clique $\{a_{i_1}, \dots, a_{i_k}\}$ is maximal, the $k+1$ values output by processes in T is not a clique in the graph G . This contradicts the agreement condition. ◀

5 More Impossibility Results from Reductions

In this section, we describe a simple reduction from approximate agreement on one graph to another graph. These reductions allow us to extend our impossibility result to even more graphs.

Let $G = (V, E)$ and $G' = (V', E')$ be undirected graphs. We say that a vertex map $\psi : V \rightarrow V'$ is a *clique map* if, for every clique κ in G , $\psi(\kappa)$ is a clique in G' .

► **Theorem 11.** *Let $G = (V, E)$ and $G' = (V', E')$ be graphs for which there exists clique maps $\psi : V \rightarrow V'$ and $\psi' : V' \rightarrow V$, such that $\psi'(\psi(u)) = u$ for all $u \in V$. Then, if approximate agreement on G' has a wait-free solution among n processes, so does approximate agreement on G .*

Proof. Let \mathcal{A}' be a wait-free protocol solving approximate agreement on the graph G' . We construct a wait-free protocol \mathcal{A} solving approximate agreement on the graph G as follows: each process with input x runs the approximate agreement algorithm \mathcal{A}' on G' using $\psi(x)$ as its input. If y' is the output it obtained from this execution of \mathcal{A}' , then it outputs $\psi'(y')$.

By the agreement property of \mathcal{A}' , the set of outputs in each execution of \mathcal{A}' is a clique κ' in G' . Since ψ' is a clique map, the set of outputs in each execution of \mathcal{A} , $\psi'(\kappa')$, is a clique in G . Hence, \mathcal{A} satisfies agreement.

To see that \mathcal{A} satisfies validity, suppose the set of inputs in some execution of \mathcal{A} is a clique κ in G . Since ψ is a clique map, it follows that $\psi(\kappa)$ is a clique in G' . By validity of \mathcal{A}' , the set of outputs in this execution of \mathcal{A} is a subset $\kappa' \subseteq \psi(\kappa)$. Thus, $\psi'(\kappa') \subseteq \psi'(\psi(\kappa)) = \kappa$.

Hence, \mathcal{A} solves approximate agreement on G in a wait-free manner among n processes. ◀

We present two applications of Theorem 11. Let $G = (V, E)$ be the 5-cycle, let $G' = (V', E')$ be the graph in Figure 3, and let C be the cycle of length 5 in the middle of G' . Let $\psi : V \rightarrow V'$ be the clique map that maps G onto C . Let $\psi' : V' \rightarrow V$ be the clique map

such that $\psi'(\psi(u)) = u$ for each vertex $u \in V$ and $\psi'(v) = w$ for each $v \in V' \setminus C$, where w is a vertex on the 5-cycle such that $\psi(w)$ is adjacent to v in G' . Since approximate agreement on the 5-cycle has no wait-free solution among $n \geq 3$ processes, Theorem 11 implies that approximate agreement on G' has no wait-free solution among $n \geq 3$ processes. Note that in Section 4.2 we showed that G' has an AER impossibility labelling, which gives another proof of this result.

The *stellated octahedron* is obtained by attaching a tetrahedron to each face of the octahedron. More formally, let $H = (V, E)$ be the octahedron graph. We can obtain the graph of the stellated octahedron graph $H' = (V', E')$ from H as follows: $V \subseteq V'$, $E \subseteq E'$, and, for each triangle $\{v_i, v_j, v_k\}$ in H , there is a new vertex $v_{\{i,j,k\}} \in V'$ and three new edges $\{v_{\{i,j,k\}}, v_i\}, \{v_{\{i,j,k\}}, v_j\}, \{v_{\{i,j,k\}}, v_k\} \in E'$. Then $\psi : V \rightarrow V'$, which maps each vertex $v \in V$ to $v \in V'$, is a clique map. Likewise, let $\psi' : V' \rightarrow V$ map each vertex $v \in V \subsetneq V'$ to v and map each vertex $v' \in V' \setminus V$ to a vertex $u \in V \subsetneq V'$ that is adjacent to v' in the stellated octahedron graph. Then ψ' is a clique map such that $\psi'(\psi(v)) = v$ for all $v \in V$. Combining Theorem 6 and Theorem 11 gives the following result.

► **Corollary 12.** *There is no wait-free protocol among 4 processes in the iterated immediate snapshot model that solves the approximate agreement problem on the stellated octahedron graph.*

6 Extension-Based Proofs

The notion of extension-based proofs was introduced by Alistarh, Aspnes, Ellen, Gelashvili, and Zhu [2]. It describes a class of impossibility proofs that includes valency arguments. Extension-based proofs are defined as an interaction between a *prover* and any protocol that claims to solve a task in a wait-free manner. The prover repeatedly queries the protocol while it attempts to construct a faulty or infinite execution of the protocol. It is known that extension-based proofs cannot be used to prove the impossibility of $(n - 1)$ -set agreement [2] and approximate agreement on 4-cycle [4]. In contrast, combinatorial proofs of these impossibility results exist [14, 20, 9, 3]. In the full version of our paper, we show that extension-based proofs cannot be used to prove the impossibility of approximate agreement on any connected graph.

► **Theorem 13.** *For any connected graph G , there is no extension-based proof of the impossibility of a wait-free solution for approximate agreement on G among $n \geq 3$ processes.*

7 Further work

We conclude by discussing a few open problems about approximate agreement on graphs.

- Is there a wait-free protocol using registers for approximate agreement on the octahedron graph for $n = 3$ processes? When $n \geq 4$, Theorem 6 implies that no wait-free algorithm exists. The algorithm by Alistarh, Ellen, and Rybicki that tolerates one crash failure [3] solves approximate agreement in a wait-free manner for $n = 2$ processes. We know that extension-based proofs are not powerful enough to obtain any impossibility result for wait-free approximate agreement on graphs. Thus, to prove that approximate agreement on the octahedron graph for $n = 3$ processes is impossible, a reduction or a combinatorial approach is required.
- For any graph G that satisfies the clique containment conditions, what is the largest number of processes for which there is a wait-free protocol using registers that solves approximate agreement on G ?

- Find a wait-free protocol using registers for approximate agreement on graphs whose complex of cliques is contractible or show that no such algorithm exists. Does knowing that a graph is contractible help to solve approximate agreement? More generally, find a decidable characterization of the class of graphs on which approximate agreement has a wait-free solution using registers. Note that there exists a topological characterization of tasks that have wait-free solutions using registers [14], but this characterization is not decidable.

References

- 1 Manuel Alcántara, Armando Castañeda, David Flores-Peñaloza, and Sergio Rajsbaum. The topology of look-compute-move robot wait-free algorithms with hard termination. *Distrib. Comput.*, 32(3):235–255, 2019.
- 2 Dan Alistarh, James Aspnes, Faith Ellen, Rati Gelashvili, and Leqi Zhu. Why extension-based proofs fail. In *Proceedings of the 51st Annual ACM Symposium on Theory of Computing*, (STOC), pages 986–996, 2019.
- 3 Dan Alistarh, Faith Ellen, and Joel Rybicki. Wait-free approximate agreement on graphs. In *Proceeding of the 28th International Colloquium on Structural Information and Communication Complexity*, (SIROCCO), pages 87–105, 2021.
- 4 Dan Alistarh, Faith Ellen, and Joel Rybicki. Wait-free approximate agreement on graphs, 2021. [arXiv:2103.08949](https://arxiv.org/abs/2103.08949).
- 5 Hagit Attiya and Armando Castañeda. A non-topological proof for the impossibility of k-set agreement. In *Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, (SSS), pages 108–119, 2011.
- 6 Hagit Attiya and Faith Ellen. *Impossibility results for distributed computing*. Morgan et Claypool Publishers, 2014.
- 7 Hagit Attiya and Faith Ellen. The step complexity of multidimensional approximate agreement. In *Proceedings of the 26th International Conference on Principles of Distributed Systems*, (OPODIS), pages 25:1–25:12, 2022.
- 8 Hagit Attiya, Nancy Lynch, and Nir Shavit. Are wait-free algorithms fast? *J. ACM*, 41(4):725–763, 1994.
- 9 Elizabeth Borowsky and Eli Gafni. Generalized flip impossibility result for t-resilient asynchronous computations. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, (STOC), pages 91–100. Association for Computing Machinery, 1993.
- 10 Armando Castañeda, Sergio Rajsbaum, and Matthieu Roy. Convergence and covering on graphs for wait-free robots. *Journal of the Brazilian Computer Society*, 24(1):1–15, 2018.
- 11 Danny Dolev, Nancy Lynch, Shlomit Pinter, Eugene Stark, and William Weihl. Reaching approximate agreement in the presence of faults. *Journal of the ACM*, 33(3):499–516, 1986.
- 12 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.
- 13 Maurice Herlihy, D. N. Kozlov, and Sergio Rajsbaum. *Distributed computing through combinatorial topology*. Morgan Kaufmann, 2014.
- 14 Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *J. ACM*, 46(6):858–923, 1999.
- 15 Gunnar Hoest and Nir Shavit. Toward a topological characterization of asynchronous complexity. *SIAM Journal on Computing*, 36(2):457–497, 2006.
- 16 Jérémy Ledent. Brief announcement: variants of approximate agreement on graphs and simplicial complexes. In *Proceedings of the 40th Annual ACM Symposium on Principles of Distributed Computing*, (PODC), pages 427–430, 2021.
- 17 Hammurabi Mendes, Maurice Herlihy, Nitin Vaidya, and Vijay K Garg. Multidimensional agreement in byzantine systems. *Distributed Computing*, 28(6):423–441, 2015.

- 18 Thomas Nowak and Joel Rybicki. Byzantine Approximate Agreement on Graphs. In *Proceedings of the 33rd International Symposium on Distributed Computing*, (DISC), pages 29:1–29:17, 2019.
- 19 M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.
- 20 Michael Saks and Fotios Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. *SIAM Journal on Computing*, 29(5):1449–35, 2000.
- 21 E. Schenk. Faster approximate agreement with multi-writer registers. In *Proceedings of the 36th IEEE Annual Symposium on Foundations of Computer Science*, (FOCS), pages 714–723, 1995.

A

 Proof of Theorem 2

In this appendix, we give a complete proof of Theorem 2. We begin with a few lemmas describing technical properties of the iterated immediate snapshot model.

► **Lemma 14.** *Let C be a (partial) configuration and $S \subseteq \pi(C)$ be a nonempty subset of processes. Let $\beta = B_1, \dots, B_\ell, \dots, B_r$ be a 1-round S -first schedule from C , where $B_1 \cup \dots \cup B_\ell$ is the set of active processes in S . Then for any 1-round schedule $\beta' = B'_1, \dots, B'_m$ such that $C\beta$ and $C\beta'$ are indistinguishable to all processes in S , B_1, \dots, B_ℓ is a prefix of β' .*

Proof. Consider the smallest k such that $B_k \neq B'_k$. Note that $B_k \cup \dots \cup B_r = B'_k \cup \dots \cup B'_m$. Suppose B_1, \dots, B_ℓ is not a prefix of β' . Then $k \leq \ell$ and $B_k \subseteq S$.

If there exists some process $q \in B'_k \setminus B_k$, then every process in $B'_k \cup \dots \cup B'_m$ will see the update by process q during β' . However, processes in $B_k \subseteq B'_k \cup \dots \cup B'_m$ will not see the update by process q during β . Hence, all processes in $B_k \subseteq S$ distinguish between $C\beta$ and $C\beta'$.

So suppose that $B'_k \subsetneq B_k$. Let $q \in B_k \setminus B'_k$. Then every process in B_k will see the update by process q during β . However, processes in B'_k will not see the update by process q during β' . Hence, all processes in $B'_k \subseteq S$ distinguish between $C\beta$ and $C\beta'$. ◀

The next result is a restatement of Lemma 8.4 from [6]. Its proof is similar.

► **Lemma 15.** *Let C be a (partial) configuration. For any (partial) configuration $D \in \chi(C, \delta)$ and any subset $Q \subsetneq \pi(C)$ of $|\pi(C)| - 1$ processes, there is at most one other (partial) configuration $D' \in \chi(C, \delta)$ such that D and D' are indistinguishable to processes in Q . Moreover, Q identifies D in $\chi(C, \delta)$ if and only if D is reached from C via a 1-round Q -first schedule.*

Proof. Consider the the 1-round schedule $\beta = B_1, B_2, \dots, B_r$ such that $D = C\beta$. Let p be the only process in $\pi(C) \setminus Q$ and let B_1, \dots, B_ℓ be the longest Q -only prefix of β . Consider an arbitrary 1-round schedule $\beta' = B'_1, \dots, B'_m$, such that $C\beta$ and $C\beta'$ are indistinguishable to processes in Q . By Lemma 14, $B'_1, \dots, B'_\ell = B_1, \dots, B_\ell$.

Case 1: β is Q -first.

If p is not active, then $\ell = r = m$ and, hence, $\beta' = \beta$. Otherwise, $\ell = r - 1$ and $B_r = \{p\}$. Hence $B'_1, \dots, B'_{r-1} = B_1, \dots, B_{r-1}$ and $B'_r = B_r = \{p\}$, so $\beta' = \beta$. In both cases, Q identifies $D = C\beta$.

Case 2: β is not Q -first.

Then process p is active in C . By definition of ℓ , $B_{\ell+1}$ is the block containing process p .

Case 2.1: $B_{\ell+1} = \{p\}$.

Then $\ell+1 < r$ because β is not Q -first. If there exists a process $q \in B'_{\ell+1} \setminus (B_{\ell+2} \cup \{p\})$, then every process in $B'_{\ell+1} \cup \dots \cup B'_m$ will see the update by process q during β' . However, processes in $B_{\ell+2} \subsetneq B'_{\ell+1} \cup \dots \cup B'_m$ will not see the update by process q during β . Hence, all processes in $B_{\ell+2} \subseteq S$ distinguish between $C\beta$ and $C\beta'$, contradicting the definition of β' . Therefore, $B'_{\ell+1} \subseteq B_{\ell+2} \cup \{p\}$.

Likewise, if there exists a process $q \in (B_{\ell+2} \cup \{p\}) \setminus B'_{\ell+1}$, then every process in $B_{\ell+2} \cup \dots \cup B_r$ will see the update by process q during β . However, processes in $B'_{\ell+1}$ will not see the update by process q during β' and, hence, are able to distinguish between $C\beta$ and $C\beta'$. Since no process in Q can distinguish between $C\beta$ and $C\beta'$, this implies that either $B'_{\ell+1} = B_{\ell+2} \cup \{p\}$ or $B'_{\ell+1} = \{p\}$.

Case 2.2: $B_{\ell+1} \neq \{p\}$.

Then $B_{\ell+1} \cap S \neq \emptyset$. If there exists $q \in B'_{\ell+1} \setminus B_{\ell+1}$, then every process in $B'_{\ell+1} \cup \dots \cup B'_m$ will see the update by q during β' . However, processes in $B_{\ell+1} \cap Q \subseteq B'_{\ell+1} \cup \dots \cup B'_m$ will not see the update by q during β . Hence, all processes in $B_{\ell+1} \cap Q$ distinguish between $C\beta$ and $C\beta'$, contradicting the definition of β' . Therefore, $B'_{\ell+1} \subseteq B_{\ell+1}$.

Likewise, if there exists $q \in B_{\ell+1} \setminus B'_{\ell+1}$, then every process in $B_{\ell+1} \cup \dots \cup B_r$ will see the update by q during β . However, processes in $B'_{\ell+1} \cap Q$ will not see the update by q during β' and, hence, are able to distinguish $C\beta$ from $C\beta'$. Since no process in Q can distinguish between $C\beta$ and $C\beta'$, this implies that either $B'_{\ell+1} = B_{\ell+1}$, or $B'_{\ell+1} \cap Q = \emptyset$. Note that $B'_{\ell+1} \cap Q = \emptyset$ implies that $B'_{\ell+1} = \{p\}$.

Thus process p has exactly two possible states in $C\beta'$. For each of these states of p , applying Lemma 14 with $S = \pi(C)$ gives a unique schedule. \blacktriangleleft

We can apply Lemma 15 repeatedly to each round of an execution to show the following result.

► Lemma 16. *Let C be a (partial) initial configuration and let t be the maximum number of non-trivial rounds taken by any $\pi(C)$ -only execution starting from C . For any $0 \leq r \leq t$ and any (partial) configuration $T \in \chi^r(C, \delta)$, a set of $|\pi(C)| - 1$ processes $Q \subsetneq \pi(C)$ identifies T in $\chi^r(C, \delta)$ if and only if T is reachable via a Q -first schedule from C .*

Proof. Let β_1, \dots, β_r be the r -round schedule such that $T = C\beta_1 \dots \beta_r$. Consider any subset $Q \subseteq \pi(C)$ of $|\pi(C)| - 1$ processes.

First suppose that β_1, \dots, β_r is not Q -first. Let $k \leq r$ be the largest index such that β_k is not Q -first. The remaining 1-round schedules $\beta_{k+1}, \dots, \beta_r$ are all Q -first. Let $C' = C\beta_1 \dots \beta_{k-1}$ and let $D = C'\beta_k \in \chi(C', \delta)$. Then, Lemma 15 says that there exists exactly one other $D' \in \chi(C', \delta)$ such that D and D' are indistinguishable to processes in Q . Since $\beta_{k+1}, \dots, \beta_r$ are Q -first, it follows that $D\beta_{k+1} \dots \beta_j$ and $D'\beta_{k+1} \dots \beta_j$ are indistinguishable to processes in Q , for each $k+1 \leq j \leq r$. Hence, Q does not identify $T = C\beta_1 \dots \beta_r$ in $\chi^r(C, \delta)$.

Now suppose that β_1, \dots, β_r is Q -first. We inductively show that Q identifies $C\beta_1 \dots \beta_i$ in $\chi^i(C, \delta)$ for all $0 \leq i \leq r$. For the base case, since C is the only configuration in $\chi^0(C, \delta)$, Q identifies C . For the inductive case, let $i < r$ and assume that Q identifies $D = C\beta_1 \dots \beta_i$ in $\chi^i(C, \delta)$. Let $E = D\beta_{i+1}$ and let $E' \in \chi^{i+1}(C, \delta)$ be such that $E \neq E'$. Then $E' \in \chi(D', \delta)$ for some $D' \in \chi^i(C, \delta)$. If $D \neq D'$, then by the inductive hypothesis, some process $q \in \pi(C)$ distinguishes between D and D' and, hence, distinguishes between E and E' . If $D = D'$, then, since β_{i+1} is Q -first, Lemma 15 implies that Q identifies E in $\chi(D, \delta)$. Hence, Q identifies $C\beta_1 \dots \beta_{i+1}$ in $\chi^{i+1}(C, \delta)$. \blacktriangleleft

► **Lemma 17.** *Let $m \geq 2$ and let \mathbb{K} be a collection of (partial) configurations such that $|\pi(C)| = m$ for all $C \in \mathbb{K}$. Suppose that, for any $C \in \mathbb{K}$ and any subset $Q \subsetneq \pi(C)$ of $m - 1$ processes, there is at most one other $C' \in \mathbb{K}$ such that $Q \subsetneq \pi(C')$ and the (partial) configurations C and C' are indistinguishable to all processes in Q . Then, for any $D \in \chi(\mathbb{K}, \delta)$ and any subset $R \subsetneq \pi(D)$ of $m - 1$ processes, there is at most one other $D' \in \chi(\mathbb{K}, \delta)$ such that $R \subsetneq \pi(D')$ and the (partial) configurations D and D' are indistinguishable to all processes in R .*

Proof. Consider a (partial) configuration $D \in \chi(\mathbb{K}, \delta)$ and a subset $R \subsetneq \pi(D)$ of $m - 1$ processes. By definition, $D = C\beta$ for some (partial) configuration $C \in \mathbb{K}$ and some 1-round $\pi(C)$ -only schedule β .

First suppose that β is not R -first. Let p be the only process in $\pi(D) \setminus R$. Then process p and at least one process in R are active in C . Consider any $D' \in \chi(\mathbb{K}, \delta)$ such that $R \subsetneq \pi(D')$ and D' is indistinguishable from D to all processes in R . Then $D' \in \chi(C', \delta)$ for some $C' \in \mathbb{K}$. Note that $\pi(C) = \pi(D)$ and $\pi(C') = \pi(D')$. Suppose that $C' \neq C$. Since β is not R -first, the scan of some active process $q \in R$ sees the update by every active process in $\pi(D)$ during β . If $\pi(D') \neq \pi(D)$, then $p \notin \pi(D')$ since $R \subsetneq \pi(D') \neq \pi(D) = R \cup \{p\}$. In this case, then q distinguishes between D and D' , because q sees the update by process p during β . Hence, $\pi(D') = \pi(D)$. If there is a process that is active in C and has a different state in $C' \neq C$, then q distinguishes between D and D' . Hence, every process that is active in C has the same state in C' . Since $C' \neq C$, there is a process q' that is terminated in C and has a different state in C' . Since p is active in C and q' is not, $q' \neq p$ and, hence, $q' \in R$. Thus, q' is a process in R that distinguishes between D' and D . This contradicts the definition of D' . Therefore, $C' = C$. Then, by Lemma 15, either $D' = D$ or $D' \neq D$ is the unique (partial) configuration in $\chi(C, \delta)$ that is indistinguishable from D to processes in R .

Now suppose that β is R -first. By assumption, there is at most one other configuration $C' \in \mathbb{K}$, such that $R \subsetneq \pi(C')$ and C' is indistinguishable from C to all processes in R . It follows that $C\beta$ and $C'\beta$ are indistinguishable to all processes in R . Since β is R -first, R identifies $C\beta$ in $\chi(C, \delta)$ and R identifies $C'\beta$ in $\chi(C', \delta)$. Hence for any other 1-round schedule $\beta' \neq \beta$, at least one process in R distinguishes between $C\beta$ and $C'\beta'$, and at least one process in R distinguishes between $C\beta$ and $C''\beta'$. For any configuration $C'' \in \mathbb{K}$ such that $C'' \neq C, C'$, there is at least one process $q \in R$ that distinguishes between C'' and C . Since q also distinguishes between $C\beta$ and $C''\beta''$ for any 1-round schedule β'' , it follows that $C'\beta$ is the only configuration in $\chi(\mathbb{K}, \delta)$ that is indistinguishable from $C\beta$ to processes in R . ◀

The rest of this section is devoted to proving Theorem 2.

► **Theorem 2.** *Let \mathbb{H} be a collection of (partial) initial configurations that satisfies the computational Sperner conditions for some protocol and let $m \geq 2$ be the number of processes represented by each (partial) configuration in \mathbb{H} . For each $(C, Q) \in \mathcal{B}(\mathbb{H})$, let $\mathbb{T}(C, Q)$ be the set of all (partial) terminal configurations T reachable from configurations in \mathbb{H} in the iterated immediate snapshot model such that*

- *m different values are output by the m processes in T and,*
- *each value in $I(C, Q)$ is output by some process in T .*

Then $|\mathbb{T}(C, Q)|$ is odd and, hence, $|\mathbb{T}(C, Q)| \geq 1$.

The proof is by strong induction on m . Let $m \geq 2$ and assume that the claim is true for all m' such that $2 \leq m' < m$. Let t be the maximum number of non-trivial rounds taken by the protocol in executions starting from (partial) initial configurations in \mathbb{H} and let $\mathbb{T} = \chi^t(\mathbb{H}, \delta)$. If $\mathcal{B}(\mathbb{H})$ is empty, then there is nothing to prove. So assume $\mathcal{B}(\mathbb{H})$ is nonempty. Fix an arbitrary $(C, Q) \in \mathcal{B}(\mathbb{H})$. Define a graph $G = (\mathbb{T} \cup \{w\}, E)$ as follows:

22:18 The Impossibility of Approximate Agreement on a Larger Class of Graphs

- There is an edge in E between (partial) terminal configurations T and T' if and only if T and T' are indistinguishable to a subset $Q' \subseteq \pi(T) \cap \pi(T')$ of $m - 1$ processes and $I(C, Q)$ is the set of values output in T (and hence in T') by the processes in Q' .
- There is an edge in E between a (partial) terminal configuration T and vertex w if and only if there is a subset $Q' \subsetneq \pi(T)$ of $m - 1$ processes that identifies T in \mathbb{T} and $I(C, Q)$ is the set of values output in T by the processes in Q' .

► **Lemma 18.** *For each (partial) terminal configuration $T \in \mathbb{T}$ adjacent to w in G , there is some $(C', Q') \in \mathcal{B}(\mathbb{H})$ such that $I(C', Q') = I(C, Q)$, T is reachable from C' via a unique schedule β , and β is Q' -first.*

Proof. Let T be adjacent to w in G . Then there exists a subset $Q' \subsetneq \pi(T)$ of $m - 1$ processes that identifies T in \mathbb{T} and $I(C, Q)$ is the set of values output in T by the processes in Q' . Since $\mathbb{T} = \chi^t(\mathbb{H}, \delta)$, it follows that $T = C' \beta_1 \dots \beta_t$ for some (partial) initial configuration $C' \in \mathbb{H}$ and some t -round $\pi(C')$ -only schedule β_1, \dots, β_t starting from C' . Since we are considering the full-information iterated immediate snapshot model, this schedule β_1, \dots, β_t is unique. Since $\chi^t(C', \delta) \subseteq \mathbb{T}$, Q' also identifies T in $\chi^t(C', \delta)$. Hence, by Lemma 16, β_1, \dots, β_t is Q' -first. If some other (partial) initial configuration $C'' \in \mathbb{H}$ is indistinguishable from C' to all processes in Q' , then $C'' \beta_1 \dots \beta_t$ is indistinguishable from $C' \beta_1 \dots \beta_t$ to all processes in Q' . This contradicts the fact that Q' identifies $C' \beta_1 \dots \beta_t$ in \mathbb{T} . Therefore, Q' identifies C' in \mathbb{H} . Hence, $(C', Q') \in \mathcal{B}(\mathbb{H})$.

Since $\beta_1 \dots \beta_t$ is Q' -first, by CSC3, the outputs of processes in Q' in T is a subset of $I(C', Q')$. Since T is adjacent to w , $I(C, Q)$ is the set of values output by processes in Q' in T . Hence we know $I(C, Q) \subseteq I(C', Q')$. However, by CSC1, $|I(C, Q)| = |I(C', Q')| = m - 1$. Thus, $I(C, Q) = I(C', Q')$. ◀

► **Lemma 19.** *For each $(C', Q') \in \mathcal{B}(\mathbb{H})$ such that $I(C', Q') = I(C, Q)$, there are an odd number of (partial) terminal configurations in \mathbb{T} reachable from C' that are adjacent to w in G .*

Proof. If $m = 2$, then $|Q'| = m - 1 = 1$. Let q be the only process in Q' . Let β_1, \dots, β_t be a t -round $\{q\}$ -first schedule starting from C' . By Lemma 16, $C' \beta_1 \dots \beta_t$ is identified by $\{q\}$ in \mathbb{T} . By CSC3, process q outputs its own input in $C' \beta_1 \dots \beta_t$. Hence, $C' \beta_1 \dots \beta_t$ is adjacent to w in G . Furthermore, by Lemma 18, if a (partial) terminal configuration T adjacent to w is reachable from C' via a schedule $\beta'_1, \dots, \beta'_t$, then $\beta'_1, \dots, \beta'_t$ is $\{q\}$ -first. Since $m = 2$, there is only one t -round $\{q\}$ -first schedule starting from C . Hence, $C' \beta_1 \dots \beta_t$ is the only (partial) terminal configuration in \mathbb{T} reachable from C' that is adjacent to w .

Now suppose $m > 2$. Consider the partial initial configuration D' of C' induced by the set of processes Q' . Let \mathbb{H}' be the collection consisting of the single partial configuration D' , let $\mathbb{T}' = \chi^t(D', \delta)$, and let $m' = m - 1 = |\pi(D')|$. Note that, \mathbb{H}' satisfies CSC4 and every subset of $\pi(D')$ identifies D' in \mathbb{H}' . Hence, $\mathcal{B}(\mathbb{H}')$ consists of the pairs (D', R') for all subsets $R' \subsetneq Q'$ of $m' - 1$ processes.

Because $(C', Q') \in \mathcal{B}(\mathbb{H})$ and \mathbb{H} satisfies CSC1, each process in Q' has a different input value in C' . Therefore, for each pair $(D', R') \in \mathcal{B}(\mathbb{H}')$, each process in $R' \subsetneq Q'$ has a different value in the partial configuration D' and, hence, \mathbb{H}' satisfies CSC1. Moreover, the set of inputs $I(D', R')$ is different for each pair $(D', R) \in \mathbb{H}'$, so \mathbb{H}' satisfies CSC2.

Let $\alpha' = \alpha'_1, \dots, \alpha'_t$ be any t -round Q' -only schedule starting from D' and let p be the only process in $\pi(C')$ that is not in $\pi(D') = Q'$. We inductively define $\phi(\alpha')$ to be the t -round $\pi(C')$ -only Q' -first schedule $\alpha_1, \dots, \alpha_t$ starting from C' , where $\alpha_i = \alpha'_i \{p\}$ if p is active in $C' \alpha_1 \dots \alpha_{i-1}$, and $\alpha_i = \alpha'_i$ otherwise. Each process in the set Q' has the same state in $D' \alpha'$ and $C' \phi(\alpha')$ and, thus, outputs the same value in both (partial) configurations.

Consider any $(D', R') \in \mathcal{B}(\mathbb{H}')$ and any subset $S \subseteq R'$. If α' is a t -round S -first Q' -only schedule starting from D' , then $\phi(\alpha')$ is a t -round S -first (and Q' -first) $\pi(C')$ -only schedule starting from C' . Since \mathbb{H} satisfies CSC3, each process in S outputs a value in $I(C', S) = I(D', S)$ in $C'\phi(\alpha')$. Each process in S outputs the same value in $D'\alpha'$ and $C'\phi(\alpha')$, so \mathbb{H}' also satisfies CSC3. Therefore, \mathbb{H}' satisfies all four computational Sperner conditions.

Fix an arbitrary pair $(D', R') \in \mathcal{B}(\mathbb{H}')$. Let $\hat{\mathbb{T}}'$ be the set of all partial terminal configuration in \mathbb{T}' such that m' different values are output by the processes in Q' . Let $T' \in \hat{\mathbb{T}}'$ and let β be the t -round Q' -only schedule such that $T' = D'\beta$. Since $(C', Q') \in \mathcal{B}(\mathbb{H})$, $\phi(\beta)$ is Q' -first, and \mathbb{H} satisfies CSC3, it follows that all values output by processes in Q' in the (partial) configuration $C'\phi(\beta)$ are elements of $I(C', Q')$. Each process in Q' outputs the same value in T' and $C'\phi(\beta)$, so all values output by processes in Q' in partial configuration T' are elements of $I(C', Q') = I(C, Q)$. By CSC1, $|I(C, Q)| = m - 1 = m'$, so $I(C, Q)$ is the set of values output by the processes in Q' in partial configuration T' . Hence each value in $I(D', R') \subseteq I(C, Q)$ is output by some process in T' . By the inductive hypothesis of Theorem 2 applied to \mathbb{H}' , it follows that $|\hat{\mathbb{T}}'|$ is odd.

Since $\phi(\beta)$ is Q' -first, Lemma 16 says that Q' identifies $C'\phi(\beta)$ in $\chi^t(C', \delta)$. Since $(C', Q') \in \mathcal{B}(\mathbb{H})$, we know that Q' also identifies C' in \mathbb{H} . Hence, for any other (partial) configuration $C'' \in \mathbb{H}$, some process $q \in Q'$ distinguishes between C'' and C' . Since the protocol is full-information, it follows that q also distinguishes between $C''\beta$ and $C'\beta$. This implies that Q' identifies $C'\phi(\beta)$ in \mathbb{T} . Hence, $C'\phi(\beta)$ is adjacent to w .

Consider any (partial) terminal configuration $T \in \mathbb{T}$ reachable from C' that is adjacent to w . By Lemma 18, there is a $\pi(C')$ -only Q' -first schedule α starting from C' such that $T = C'\alpha$. Since α is Q' -first and $Q' = \pi(C') \setminus \{p\}$, it follows that $\alpha = \phi(\alpha')$ for some Q' -only schedule α' starting from D' . Since $D'\alpha'$ is indistinguishable from T to processes in Q' and $I(C, Q)$ is the set of values output by processes in Q' in T , it follows that $I(C, Q)$ is also the set of values output by processes in Q' in $D'\alpha'$ and, thus, $D'\alpha' \in \mathbb{T}'$. Hence $|\mathbb{T}'|$ is at least the number of (partial) terminal configurations in \mathbb{T} reachable from C' that are adjacent to w in G .

Now consider any two different partial terminal configurations $T', T'' \in \mathbb{T}'$. Then there exists two different schedules β', β'' starting from D' such that $T' = D'\beta'$ and $T'' = D'\beta''$. Since the protocol is full-information, some process $q \in \pi(D') = Q'$ distinguishes between $D'\beta'$ and $D'\beta''$. Hence, q distinguishes between $C'\phi(\beta')$ and $C'\phi(\beta'')$. Both $C'\phi(\beta')$ and $C'\phi(\beta'')$ are (partial) terminal configurations adjacent to w . Thus $|\mathbb{T}'|$ is at most the number of (partial) terminal configurations in \mathbb{T} reachable from C' that are adjacent to w in G . Therefore, $|\mathbb{T}'|$ is number of (partial) terminal configurations in \mathbb{T} reachable from C' that are adjacent to w in G . Since $|\mathbb{T}'|$ is odd, the statement of the lemma follows. \blacktriangleleft

The next lemma follows from Lemma 18, Lemma 19, and the fact that \mathbb{H} satisfies CSC2.

► **Lemma 20.** *Vertex w has odd degree in G .*

Proof. By Lemma 18, each (partial) terminal configuration T adjacent to w in G is reachable from C' for some $(C', Q') \in \mathcal{B}(\mathbb{H})$ such that $I(C', Q') = I(C, Q)$. By CSC2, there are an odd number of pairs $(C', Q') \in \mathcal{B}(\mathbb{H})$ such that $I(C', Q') = I(C, Q)$. For each such pair (C', Q') , Lemma 19 tells us that there are an odd number of (partial) terminal configurations in \mathbb{T} reachable from C that are adjacent to w in G . Thus, w has odd degree in G . \blacktriangleleft

The collection \mathbb{H} of initial configurations satisfies CSC4. Thus, applying Lemma 17 t times shows that, for each (partial) terminal configuration $T \in \mathbb{T} = \chi^t(\mathbb{H}, \delta)$ and each subset $Q' \subsetneq \pi(T)$ of $m - 1$ processes, there is at most one other (partial) terminal configuration

22:20 The Impossibility of Approximate Agreement on a Larger Class of Graphs

$T' \in \mathbb{T}$ such that $Q' \subsetneq \pi(T')$ and T and T' are indistinguishable to all processes in Q' . If there is such a (partial) terminal configuration T' and $I(C, Q)$ is the set of values output in T by the processes in Q' , then T is adjacent to T' in G . If Q' identifies T in \mathbb{T} and $I(C, Q)$ is the set of values output in T by the processes in Q' , then T is adjacent to w in G . Hence, the degree of T in graph G is the number of $(m - 1)$ -element subsets $Q' \subsetneq \pi(T)$ such that $I(C, Q)$ is the set of values output in T by the processes in Q' .

Let $\hat{\mathbb{T}}$ be the set of all (partial) terminal configurations in \mathbb{T} with odd degree in G . By the handshaking lemma, every graph must have an even number of odd degree vertices and, by Lemma 20, w has odd degree. Thus, $|\hat{\mathbb{T}}|$ is odd.

Let $T \in \mathbb{T}$. Recall that $|\pi(T)| = m$ and, by CSC1, $|I(C, Q)| = m - 1$. If $I(C, Q)$ is the set of values output by $\pi(T)$ in T , then T has degree 2. If $I(C, Q)$ is a proper subset of the set of values output by $\pi(T)$ in T , then T has degree 1. Otherwise, T has degree 0.

Hence $\hat{\mathbb{T}}$ is the set of all (partial) terminal configurations $T \in \mathbb{T}$ such that m different values are output and each value in $I(C, Q)$ is output by some process. Therefore, $\hat{\mathbb{T}} = \mathbb{T}(C, Q)$. This concludes the proof of Theorem 2.

On the Hierarchy of Distributed Majority Protocols

Petra Berenbrink  

Universität Hamburg, Germany

Amin Coja-Oghlan  

TU Dortmund University, Germany

Oliver Gebhard  

TU Dortmund University, Germany

Max Hahn-Klimroth  

TU Dortmund University, Germany

Dominik Kaaser  

TU Hamburg, Germany

Malin Rau  

Universität Hamburg, Germany

Abstract

We study the *consensus* problem among n agents, defined as follows. Initially, each agent holds one of two possible opinions. The goal is to reach a consensus configuration in which every agent shares the same opinion. To this end, agents randomly sample other agents and update their opinion according to a simple update function depending on the sampled opinions.

We consider two communication models: the *gossip model* and a variant of the *population model*. In the gossip model, agents are activated in parallel, synchronous rounds. In the population model, one agent is activated after the other in a sequence of discrete time steps. For both models we analyze the following natural family of majority processes called j -Majority: when activated, every agent samples j other agents uniformly at random (with replacement) and adopts the majority opinion among the sample (breaking ties uniformly at random). As our main result we show a hierarchy among majority protocols: $(j + 1)$ -Majority (for $j > 1$) converges *stochastically faster* than j -Majority for any initial opinion configuration. In our analysis we use Strassen's Theorem to prove the existence of a coupling. This gives an affirmative answer for the case of two opinions to an open question asked by Berenbrink et al. [PODC 2017].

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Theory of computation → Random walks and Markov chains; Mathematics of computing → Stochastic processes

Keywords and phrases Consensus, Majority, Hierarchy, Stochastic Dominance, Population Protocols, Gossip Model, Strassen's Theorem

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2022.23

Related Version *Full Version:* <https://arxiv.org/abs/2205.08203>

Funding *Petra Berenbrink:* DFG FOR 2975

Amin Coja-Oghlan: DFG FOR 2975

Oliver Gebhard: DFG CO 646/3

Max Hahn-Klimroth: DFG FOR 2975

Malin Rau: DFG FOR 2975

1 Introduction

We consider the problem of *consensus* in a distributed system of n identical, anonymous agents. Initially each agent has one of two opinions and the goal is that all agents agree on the same opinion. Reaching consensus is a fundamental task in distributed computing



© Petra Berenbrink, Amin Coja-Oghlan, Oliver Gebhard, Max Hahn-Klimroth, Dominik Kaaser, and Malin Rau;

licensed under Creative Commons License CC-BY 4.0

26th International Conference on Principles of Distributed Systems (OPODIS 2022).

Editors: Eshcar Hillel, Roberto Palmieri, and Etienne Rivière; Article No. 23; pp. 23:1–23:19



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

with a multitude of applications, including fault tolerance in distributed sensor array, clock synchronization, control of autonomous robots, or blockchains. In computational sciences, consensus protocols model, e.g., dynamic particle systems or biological processes. In social sciences, consensus protocols have been studied in the context of opinion formation processes among social interaction systems. See [8] for a quite recent survey including references and further applications.

We study the simple and well-known class of j -Majority protocols [10, 35, 12] in two communication models, the classical *gossip model* [19, 9, 8] and a *sequential model*, a variant of the prominent *population model* [4]. In the gossip model, all agents are activated in parallel, synchronous rounds. In the sequential model, one agent is activated after the other uniformly at random. Every activated agent u considers the opinions of j agents v_1, \dots, v_j sampled uniformly at random (with replacement). It then adopts the majority opinion among the sampled opinions, breaking ties uniformly at random. We are interested in the time it takes until the protocol *converges* such that all agents share the same opinion. Setting $j = 1$ yields the so-called VOTER process [17]. A variant of 2-Majority with lazy tie-breaking is known as TWO-SAMPLE VOTING [23] or the TWOCHOICES process [35], and the 3-Majority dynamics is analyzed in [10].

The main idea of majority processes with $j > 1$ is to speed up the convergence time. For the VOTER process in the gossip model, the convergence time is linear in n (independently of the number of initial opinions) [17], whereas the convergence time of 3-Majority is $O(k \log n)$ for $k = o(n)$ possible initial opinions [35]. In [12] the authors compare the TWOCHOICES process to 3-Majority. They show a stochastic dominance of the convergence time of 3-Majority over the convergence time of VOTER and TWOCHOICES, assuming k initial opinions. For j -Majority, they conjecture a *hierarchy* of protocols (see Conjecture 6.1 in [12]). In particular, they ask whether one can couple j -Majority and $(j + 1)$ -Majority for $j \in \mathbb{N}$ such that $(j + 1)$ -Majority is stochastically faster than j -Majority.

In this paper, we settle the matter for the case of $k = 2$ opinions and prove the existence of such a hierarchy of majority protocols. Intuitively, this establishes that the processes converge *faster* (or at least *equally fast*) for larger values of j . Let T_j be the random variable for the convergence time of j -Majority. We formally prove that T_{j+1} stochastically minorizes T_j , written $T_{j+1} \preceq T_j$, assuming both processes start in the same configuration. Formally, we show that $\Pr[T_{j+1} \geq t] \leq \Pr[T_j \geq t]$ for any $t \in \mathbb{N}$. Our main technical contribution is the formal proof of this stochastic dominance.

Our proof has its foundations in quite natural observations regarding the transition properties of the j -Majority processes. Similar results for individual steps of the process have been shown, e.g., in [33]. However, formally proving and maintaining the stochastic dominance over all possible configurations requires a lot of care, and to the best of our knowledge, our result is the first proof of stochastic dominance that covers the entire execution of j -Majority for all $j \in \mathbb{N}$ in the setting with two opinions. To motivate the obstacles we have to overcome, observe that the process is influenced by opposing *forces*. Specifically, in order to make progress, an agent from the minority opinion must be activated to interact with at least $j/2$ agents from the majority opinion. Activating an agent with minority opinion becomes less likely with increasing majority, while sampling at least $j/2$ agents with majority opinion becomes more likely with increasing majority. In our analysis we carefully prove that these forces balance out in a favorable manner.

Finally, we consider 3-Majority. We show an asymptotically optimal bound in the sequential model on the convergence time of $O(n \log n)$ activations. This matches a similar result shown by Ghaffari and Lengler [35] for 3-Majority in the gossip model. Our theoretical findings are complemented by empirical results. We simulate j -Majority processes for various values of j and large numbers of agents ranging from $n = 10^2$ to $n = 10^8$.

1.1 Related Work

Consensus in the Gossip Model. A simple and natural consensus process is the so-called VOTER process [37, 43, 22, 17, 38] where every agent adopts the opinion of a single, randomly chosen agent in each round. The expected convergence time of VOTER in the gossip model is at least linear [17]. In order to speed up the process, two related protocols have been proposed, namely the TWOCHOICES process [31, 23, 24, 25] and the 3-MAJORITY dynamics [10, 35, 12]. In both processes, each agent u takes three opinions and updates its opinion to the majority among the sample. In the TWOCHOICES process, u takes its own opinion and samples two opinions u.a.r. Ties are broken towards u 's own opinion. In the 3-MAJORITY dynamics, u samples three opinions u.a.r. breaking ties randomly. In [35] the authors consider arbitrary initial configurations in the gossip model. They show that TWOCHOICES with $k = O(\sqrt{n/\log n})$ and 3-MAJORITY with $k = O(n^{1/3}/\log n)$ reach consensus in $O(k \cdot \log n)$ rounds, improving a result by Becchetti et al. [10]. For arbitrary k they show that 3-MAJORITY reaches consensus in $O(n^{2/3} \log^{3/2} n)$ rounds w.h.p., improving a result by Berenbrink et al. [12].

Schoenebeck and Yu [45] consider a generalization of multi-sample consensus protocols on complete and Erdős-Rényi graphs for two opinions. Their probabilistic model covers various consensus processes, including j -Majority, by using a so-called *update rule*, a function $f: [0, 1] \rightarrow [0, 1]$. In each round, every agent u adopts opinion a with probability $f(\alpha(u))$ for some function f , where $\alpha(u)$ is the fraction of neighbors of agent u that have opinion a . Depending on certain natural properties on f , they analyze the convergence time for complete graphs and Erdős-Rényi graphs.

Another related process is the MEDIANRULE [26], where in each round every agent adopts the median of its own opinion and two sampled opinions, assuming a total order among opinions. It reaches consensus in $O(\log k \log \log n + \log n)$ rounds w.h.p. For two opinions the MEDIANRULE is equivalent to the TWOCHOICES process, and their analysis is tight. For the case of $k > 2$ opinions we remark that assuming a total order among the opinions is a strong assumption that is not required by any of the other protocols.

Finally, considerable amount of work has been spent on analyzing the so-called undecided state dynamics introduced by Angluin et al. [5]. The basic idea is that whenever two agents with different opinions interact, they lose their opinions and become *undecided*, and undecided agents adopt the first opinion they encounter. Clementi et al. [20] study the undecided state dynamics in the gossip model. They consider two opinions and show that the protocol reaches consensus in $O(\log n)$ rounds w.h.p. If there is a so-called *bias* of order $\Omega(\sqrt{n \log n})$, the initial plurality opinion prevails. The (additive) bias is the difference between the numbers of agents holding either opinion. Becchetti et al. [9] analyze the undecided state dynamics for $k = O(n/\log n)^{1/3}$ opinions and show a convergence time of $O(k \cdot \log n)$ rounds w.h.p. Bankhamer et al. [36], Berenbrink et al. [16], and Ghaffari and Parter [6] consider a synchronized variant that runs in phases of length $\Theta(\log k)$. Agents can become undecided only at the start of such a phase and use the rest of the phase to obtain a new opinion. These synchronized protocols achieve consensus in $O(\log^2 n)$ rounds w.h.p. and can be further refined using more sophisticated synchronization mechanisms.

Majority and Consensus in the Population Model. In *exact* majority the goal is to identify the majority among two possible opinions, even if the bias is as small as only one [30, 41, 3, 42, 29, 1, 2, 18, 39, 14, 15, 11, 28]. The best known protocol by Doty et al. [28] solves exact majority with $O(\log n)$ states and $O(\log n)$ parallel time, both in expectation

and w.h.p. This is optimal: it takes at least $\Omega(n \log n)$ interactions until each agent interacts at least once, and any majority protocol which stabilizes in expected $n^{1-\Omega(1)}$ parallel time requires at least $\Omega(\log n)$ states (under some natural conditions, see [2]).

Approximate majority is easier: a simple 3-state protocol [5, 21] reaches consensus w.h.p. in $O(\log n)$ parallel time and correctly identifies the initial majority w.h.p. if an initial bias of order $\Omega(\sqrt{n \log n})$ is present. Condon et al. [21] also consider a variant of the 3-Majority process in (a variant of) the gossip model where three randomly chosen agents interact. They show a parallel convergence time of $O(k \log n)$ w.h.p., provided a sufficiently large initial bias is present. Furthermore, Kosowski and Uznanski [39] mention a protocol which determines the exact majority in $O(\log^2 n)$ parallel time w.h.p. using only constantly many states.

Less is known about population protocols that solve consensus among more than two opinions. One line of research considers only the required number of states to eventually identify the opinion with the largest initial support correctly. For this problem, Natale and Ramezani [44] show a lower bound of $\Omega(k^2)$ states via an indistinguishability argument. The currently best known protocol uses $O(k^6)$ states if there is an order among the opinions and $O(k^{11})$ states otherwise [34]. Sacrificing the strong guarantees of always-correct exact plurality consensus, Bankhamer et al. [6] achieve *approximate consensus* in $O(\log^2 n)$ parallel time w.h.p. using only $O(k \log n)$ states. If there is an initial bias of order $\Omega(\sqrt{n \log n})$, the initial plurality opinion wins w.h.p. In [7] another variant of the population model is considered where agents are activated by random clocks. At each clock tick, every agent may open communication channels to constantly many other agents chosen uniformly at random or from a list of at most constantly many agents contacted in previous steps. In this model, opening communication channels is subject to a random delay. The authors show that consensus is reached by all but a $1/\text{poly } \log n$ fraction of agents in $O(\log \log_\alpha k \log k + \log \log n)$ parallel time w.h.p., provided a sufficiently large bias is present.

1.2 Models and Results

Gossip Model. In the gossip model [19, 9, 8] all agents are activated simultaneously in synchronous rounds. In each round every agent u opens a communication channel to j agents v_1, \dots, v_j chosen independently and uniformly at random with replacement. (For simplicity we also allow that $v_i = u$ and assume that the v_i are sampled with replacement.) The running time (or *convergence time*) of a majority protocol is measured in the numbers of rounds until all agents agree on the same opinion.

Sequential Model. The population model was introduced by Angluin et al. [4] to model systems of resource limited mobile agents that perform a computation via a sequence of pairwise interactions. We consider a variant where in each time step one agent u is chosen uniformly at random to interact with j randomly sampled agents v_1, \dots, v_j . (As before, we do not rule out that $u = v_i$ for some i). When u is activated it updates its opinion according to the random sample. The running time is measured in the number of interactions. To allow for a comparison with the (inherently) parallel gossip model, the so-called *parallel time* is defined as the number of interactions divided by the number of agents n . Note that our processes do not *halt*: agents do not know that consensus has been reached (see also the impossibility result in [27]).

j -Majority Processes. In the following we use P_j to denote the j -Majority process. When executing process P_j , the system transitions through a sequence of *configurations* $(C_t)_{t \in \mathbb{N}_0}$. At time $t \in \mathbb{N}_0$ the *configuration* $C_t \in \{a, b\}^n$ assigns each agent an opinion in $\{a, b\}$. In

our analysis we are interested in the number of agents with majority opinion. We will always assume w.l.o.g. that a is the majority opinion and we denote a *state* X_t as the number of agents with majority opinion in configuration C_t . The configuration C_0 at time 0 is called the *initial configuration* and the corresponding state X_0 is called the *initial state*. The *convergence time* $T_j(C_0)$ is defined as the first time where all agents have the same opinion when starting process P_j in initial configuration C_0 . Note that the convergence time only depends on the number of agents with majority opinion since two agents with the same opinion are not distinguishable. Hence we write $T_j(X_0)$ in the following. Formally, $T_j(X_0)$ is a stopping time defined as $T_j(X_0) = \min \{ t \in \mathbb{N}_0 \mid X_t = n \}$. Each transition of the system is done according to the following update rule.

► **Definition (Process P_j).** *Agents are activated according to either the gossip model or the sequential model. In process P_j each activated agent u samples j agents with replacement and adopts the majority opinion among the sample, breaking ties uniformly at random.*

Note that tie-breaking is not required in process P_{2j+1} (i.e., when every agent samples an odd number of agents). Since we have $k = 2$ opinions we are guaranteed to have a clear majority in this case.

Stochastic Dominance. Before we formally present our result, it remains to define stochastic dominance.

► **Definition (Stochastic Dominance).** *Let \mathcal{E} be a Polish space¹ with a partial ordering $\leq_{\mathcal{E}}$. Let $\mu, \nu \in \mathcal{P}(\mathcal{E})$ be probability measures on \mathcal{E} . If, for every $x \in \mathcal{E}$, we have*

$$\mu(\{y \in \mathcal{E} : y \geq_{\mathcal{E}} x\}) \geq \nu(\{y \in \mathcal{E} : y \geq_{\mathcal{E}} x\}),$$

we say that μ stochastically dominates ν . In this case we also say that μ majorizes ν (written as $\nu \succeq \mu$) or ν minorizes μ ($\nu \preceq \mu$).

We now formally state our main result which applies for both communication models, the gossip model and the sequential model.

► **Theorem 1 (Main Result).** *Let $T_j(X_0)$ be the convergence time of process P_j with initial state X_0 in either the gossip model or the sequential model. Then*

$$T_{j+1}(X_0) \preceq T_j(X_0) \quad \text{for any } j > 1.$$

Furthermore, for all $j > 1$,

$$\mathbb{E}[T_{2j+2}(X_0)] = \mathbb{E}[T_{2j+1}(X_0)] < \mathbb{E}[T_{2j}(X_0)].$$

In our second result we show that 3-Majority P_3 converges in $O(n \log n)$ time w.h.p.² To the best of our knowledge, this is the first analysis of 3-Majority with sequential updates. Our proof is similar to the proof by Condon et al. [21] for the convergence time of approximate majority in tri-molecular chemical reaction networks. We emphasize that Theorem 1 implies that all j -Majority processes with $j > 3$ converge in $O(n \log n)$ time w.h.p.

¹ A Polish space is a complete metric space with a countable dense subset.

² The expression *with high probability (w.h.p.)* refers to a probability of $1 - n^{-\Omega(1)}$.

► **Theorem 2.** Let $T_3(X_0)$ be the convergence time of the 3-Majority process P_3 in the sequential model with initial configuration X_0 .

1. It holds that $T_3(X_0) \leq O(n \log n)$ w.h.p.
2. If $X_0 \geq n/2 + \zeta \sqrt{n \log n}$ for some sufficiently large constant $\zeta > 0$ then the initial majority opinion wins w.h.p.

We remark that the convergence time of $O(n \log n)$ is asymptotically tight. Indeed, for any number of time steps in $o(n \log n)$ there is a constant probability that two agents with opposing opinions are not activated even once.

2 Analysis

In this section we formally prove our theorems. We prove Theorem 1 in Section 2.1 and Section 2.2 for the sequential model and the gossip model, respectively. Theorem 2 is then shown in Section 2.3. All technical details for the rigorous proofs can be found in the full version [13].

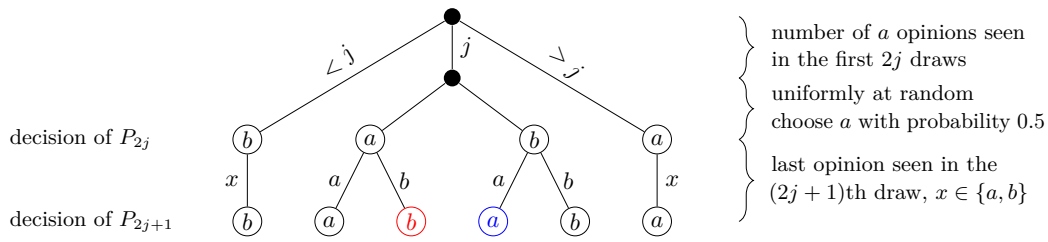
2.1 Sequential Model

We start our analysis with a comparison of one step of the processes P_j and P_{j+1} at time t when starting in an identical state X_t . We are able to express the differences in the probabilities of increasing the majority opinion, decreasing it or remaining in the same state for the both processes. To this end, we visualize a possible coupling by a decision tree that incorporates all the different possibilities. We will observe that, within this one step, we can couple both processes such that the supposedly faster process increases the majority opinion with probability one if the supposedly slower process increases this opinion. This coupling will be guaranteed by an application of *Strassen's Theorem*.

The proof of the main result will be conducted inductively. We start both processes in the same initial state and assume that there is a majority opinion a . Now, the aforementioned coupling ensures that, after the first step, the supposedly faster process will have at least as many agents of opinion a than the supposedly slower process. Now, we show a kind of *monotony* in the studied processes. Assume we have two instances of the same process, one in state $X_t = s$ and one in state $X'_t = s'$ where X_t, X'_t denote the number of agents with opinion a after t steps. If $s > s'$, then the random variable X_{t+1} will stochastically dominate X'_{t+1} , formally $X'_{t+1} \preceq X_{t+1}$. This observation is crucial. It allows us to show that in the second step, we can again construct a coupling such that, if the supposedly slower process moves, the supposedly faster process does as well almost surely. Indeed, either both processes are in the same state, then we find the stochastic dominance by the decision trees, or the fast process has more agents of opinion a . But as stochastic dominance is transitive, we can construct a coupling via the triangle inequality.

Finally, we will describe the overall coupling of the two processes as the *path-coupling* along those couplings per step which will prove the first part of Theorem 1. The second part will follow analogously as we can show via the decision trees that in the comparison of P_{2j-1} and P_{2j} , the chance to obtain the same state in the next step is equal under both processes while in the comparison of P_{2j} and P_{2j+1} those decision trees show that the probability of increasing the majority opinion is larger in P_{2j+1} .

► **Observation 3.** The processes P_1 and P_3 have, almost surely, a finite stopping time.



■ **Figure 1** Decision tree comparing P_{2j} and P_{2j+1} .

For the sequential process, we show this for P_3 in Section 2.3, while for P_1 this follows by the results of Schoenebeck and Yu [45]. For the gossip model, this is proven in [35]. In this setting, Strassen’s Theorem guarantees the existence of a coupling $\gamma \in \mathcal{P}(\mathcal{E}^2)$ of μ and ν with the following property.

► **Theorem 4** (Strassen’s Theorem [46]). *Let μ, ν be probability measures on a Polish space endowed with a partial ordering \preceq such that μ stochastically dominates ν . Let $X \sim \mu$ and $Y \sim \nu$, then there is a coupling γ of μ and ν such that, if $(\hat{X}, \hat{Y}) \sim \gamma$, we have*

$$X \stackrel{d}{=} \hat{X}, \quad Y \stackrel{d}{=} \hat{Y} \quad \text{and} \quad \Pr[\hat{Y} \preceq \hat{X}] = 1.$$

In words, this means that if μ stochastically dominates ν , X is sampled from μ , and Y is sampled from ν , there is a coupling under which $X \leq Y$ almost surely (with probability 1).

► **Lemma 5.** *We find for P_k the following. Let X_t denote the number of agents with majority opinion at time t . If $s > s'$, then for all $d \in \{0, 1, \dots, n\}$*

$$\Pr[X_{t+1} \geq d \mid X_t = s] \geq \Pr[X_{t+1} \geq d \mid X_t = s'].$$

We provide the detailed calculation in the full version [13] and get the following corollary.

► **Corollary 6.** *For any two processes P, P' , we find the following stochastic dominance. Let X_t denote the number of agents with opinion a with respect to process P at time t and let X'_t be the analogous quantity with respect to P' . Assume that for any $d \in [n]$*

$$\Pr[X_{t+1} \geq d \mid X_t = s] \geq \Pr[X'_{t+1} \geq d \mid X'_t = s],$$

then we have also

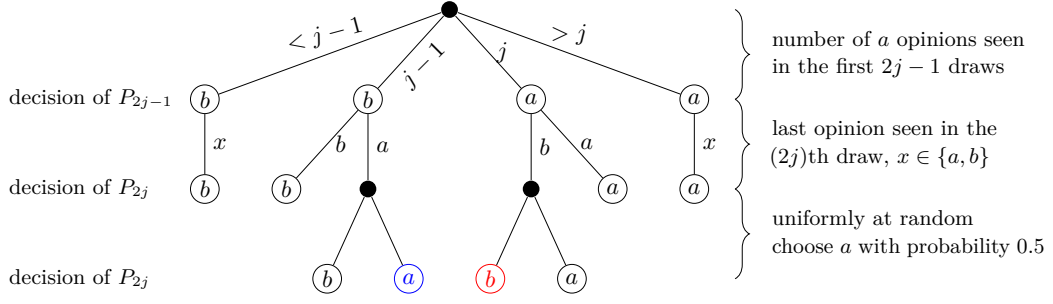
$$\Pr[X_{t+1} \geq d \mid X_t = s + t'] \geq \Pr[X'_{t+1} \geq d \mid X'_t = s],$$

for any $d \in [n]$ and $t' > 0$ such that $s + t' \leq n$.

Let $X_t^{(k)}$ denote the number of agents with majority opinion after step t of process P_k for any $k \in \mathbb{N}$. Furthermore, for a given agent x we denote by $x_t^{(k)}$ its opinion in process P_k at time t . In the following we compare two processes with each other. The comparisons of P_{2j} (even) to P_{2j+1} (odd) and P_{2j-1} (odd) to P_{2j} (even) require slightly different calculations. Therefore, we have to show two similar lemmas for these two cases, Lemma 7 for the former case and Lemma 9 for the latter case.

First we compare two successive processes P_{2j} and P_{2j+1} . The following lemma states that in process P_{2j+1} it is more likely for an agent with opinion b to change to a while in process P_{2j} it is more likely that an agent with opinion a changes to opinion b than in the other process respectively.

23:8 On the Hierarchy of Distributed Majority Protocols



■ **Figure 2** Decision tree comparing P_{2j-1} and P_{2j} .

► **Lemma 7.** Let x be an agent that is updated in the next step, $x_t^{(k)}$ its opinion in process P_k at time t , $s \in [n]$ and $\alpha = \frac{s}{n}$. It holds that

$$\begin{aligned} & \Pr \left[x_{t+1}^{(2j+1)} = a \mid x_t^{(2j+1)} = b, X_t^{(2j+1)} = s \right] \\ &= \Pr \left[x_{t+1}^{(2j)} = a \mid x_t^{(2j)} = b, X_t^{(2j)} = s \right] + \frac{(2\alpha - 1) \binom{2j}{j}}{2} \alpha^j (1 - \alpha)^j \end{aligned}$$

and

$$\begin{aligned} & \Pr \left[x_{t+1}^{(2j+1)} = b \mid x_t^{(2j+1)} = a, X_t^{(2j+1)} = s \right] \\ &= \Pr \left[x_{t+1}^{(2j)} = b \mid x_t^{(2j)} = a, X_t^{(2j)} = s \right] - \frac{(2\alpha - 1) \binom{2j}{j}}{2} \alpha^j (1 - \alpha)^j. \end{aligned}$$

To prove these equations it is sufficient to study the cases where the two processes have a different outcome. The probability for these cases directly reflects the difference in probability for that specific outcome. These cases are highlighted in Figure 1. We provide the detailed calculation in the full version.

This difference in probabilities allows us to prove that, given the same state, P_{2j+1} stochastically dominates the process P_{2j} in the next step:

► **Lemma 8.** For each $j \in \mathbb{N}_0$ and each $s \in \mathbb{N}$ with $s > n/2$ and any $d \in [n]$ it holds that

$$\Pr \left[X_{t+1}^{(2j+1)} \geq d \mid X_t^{(2j+1)} = s \right] \geq \Pr \left[X_{t+1}^{(2j)} \geq d \mid X_t^{(2j)} = s \right].$$

Note that this inequality follows trivially for $d \leq s - 1$ and $d > s + 1$. To prove the property for the cases $d = s$ and $d = s + 1$ we can directly use the properties from Lemma 7.

On the other hand, when comparing the processes P_{2j-1} and P_{2j} with respect to the difference in probability for an agent to change its opinion, we note that there is no difference in the probabilities given that all agents are in the same state.

► **Lemma 9.** Let x be an agent that is updated in the next step, $x_t^{(k)}$ its opinion in process P_k at time t , and $s \in [n]$. It holds that

$$\Pr \left[x_{t+1}^{(2j-1)} = a \mid x_t^{(2j-1)} = b, X_t^{(2j-1)} = s \right] = \Pr \left[x_{t+1}^{(2j)} = a \mid x_t^{(2j)} = b, X_t^{(2j)} = s \right]$$

and

$$\Pr \left[x_{t+1}^{(2j-1)} = b \mid x_t^{(2j-1)} = a, X_t^{(2j-1)} = s \right] = \Pr \left[x_{t+1}^{(2j)} = b \mid x_t^{(2j)} = a, X_t^{(2j)} = s \right].$$

A similar statement has previously been shown by Fraigniaud and Natale [33] for a related model. The proof of Lemma 9 is analogous to the proof of Lemma 7. For completeness, it can be found in the full version

► **Lemma 10.** *For each $j \in \mathbb{N}_0$ and each $s \in \mathbb{N}$ with $s > n/2$ and any $d \in [n]$ it holds that*

$$\Pr \left[X_{t+1}^{(2j)} \geq d \mid X_t^{(2j)} = s \right] = \Pr \left[X_{t+1}^{(2j-1)} \geq d \mid X_t^{(2j-1)} = s \right].$$

The proof can be found in the full version. We are now ready to put everything together and prove our main result for the sequential model.

Proof of Theorem 1. We prove Theorem 1 by induction given the initial state X_0 and start with the case $T_{2j}(X_0) \preceq T_{2j+1}(X_0)$. Given X_0 , Lemma 8 guarantees that for all $s > 0$

$$\Pr \left[X_1^{(2j+1)} \geq s \mid X_0 \right] \geq \Pr \left[X_1^{(2j)} \geq s \mid X_0 \right].$$

Therefore, by Theorem 4, we find a coupling γ_1 such that, under γ_1 , $X_1^{(2j+1)} \geq X_1^{(2j)}$ almost surely. Now, assume that we constructed a coupling $\gamma^{(t)} = \gamma_1 \otimes \dots \otimes \gamma_t$ of $(X_1^{2j}, \dots, X_t^{2j})$ and $(X_1^{2j+1}, \dots, X_t^{2j+1})$, where \otimes denotes the product measure. Under $\gamma^{(t)}$ we have by induction hypothesis that

$$\Pr_{\gamma^{(t)}} \left[X_t^{2j+1} \geq X_t^{2j} \right] = 1.$$

Therefore, by Corollary 6 and Lemma 8, we find given $X_t^{(2j+1)} \geq X_t^{(2j)}$ that

$$\Pr \left[X_{t+1}^{(2j+1)} \geq s \mid X_t^{(2j+1)} \right] \geq \Pr \left[X_{t+1}^{(2j)} \geq s \mid X_t^{(2j)} \right].$$

Thus, Theorem 4 implies, given $X_t^{(2j+1)} \geq X_t^{(2j)}$ the existence of a coupling γ_{t+1} such that

$$\Pr[\gamma_{t+1}] X_{t+1}^{2j+1} \geq X_{t+1}^{2j} = 1.$$

We define $\gamma^{(t+1)} = \gamma^{(t)} \otimes \gamma_{t+1}$ and $T_{2j}(X_0) \preceq T_{2j+1}(X_0)$ follows by induction.

Next, we need to prove that $T_{2j-1}(X_0) \preceq T_{2j}(X_0)$. This follows completely analogously with Lemma 8 replaced by Lemma 10.

Finally, we need to construct the bounds on the expectation. Given the coupling $\gamma^{T(2j+1)(X_0)}$ of P_{2j+1} and P_{2j} , we find that, under this coupling, for every step $t = 1 \dots T_{2j+1}$, we have $X_t^{(2j+1)} \geq X_t^{(2j)}$ almost surely and therefore $\mathbb{E}[T_{2j+1}(X_0)] \leq \mathbb{E}[T_{2j}(X_0)]$. In more detail, due to the coupling γ , we know that

$$\Pr \left[X_{t+1}^{(2j+1)} \geq s \mid X_0 \right] \geq \Pr \left[X_{t+1}^{(2j)} \geq s \mid X_0 \right]$$

and hence

$$\Pr \left[X_{t+1}^{(2j+1)} < s \mid X_0 \right] \leq \Pr \left[X_{t+1}^{(2j)} < s \mid X_0 \right]$$

for each $s \geq n/2$ and $t \in \mathbb{N}$. Furthermore, for the first possible convergence time it holds that

$$\begin{aligned} \Pr \left[X_{n-s}^{(2j)} < n \mid X_0 = s \right] &= 1 - \Pr \left[X_{n-s}^{(2j)} = n \mid X_0 = s \right] \\ &= 1 - \prod_{i=1}^{n-s} \Pr \left[X_i^{(2j)} = s+i \mid X_{i-1}^{(2j)} = s+i-1 \right] \\ &\stackrel{\text{Lemma 7}}{>} 1 - \prod_{i=1}^{n-s} \Pr \left[X_i^{(2j+1)} = s+i \mid X_{i-1}^{(2j+1)} = s+i-1 \right] \\ &= 1 - \Pr \left[X_{n-s}^{(2j+1)} = n \mid X_0 = s \right] = \Pr \left[X_{n-s}^{(2j+1)} < n \mid X_0 = s \right]. \end{aligned}$$

23:10 On the Hierarchy of Distributed Majority Protocols

Therefore,

$$\begin{aligned} \mathbb{E}[T_{2j}(X_0)] &= \sum_{t \geq 0} \Pr[T_{2j}(X_0) > t] = \sum_{t \geq 0} \Pr[X_t^{(2j)} < n \mid X_0] \\ &> \sum_{t \geq 0} \Pr[X_t^{(2j+1)} < n \mid X_0] = \sum_{t \geq 0} \Pr[T_{2j+1}(X_0) > t] = \mathbb{E}[T_{2j+1}(X_0)]. \end{aligned}$$

Next we prove the equality in expectation for the convergence time of the processes P_{2j-1} and P_{2j} . To this end, we get from Lemma 10 that

$$\Pr[X_t^{(2j-1)} < n \mid X_{t-1} = s] = \Pr[X_t^{(2j)} < n \mid X_{t-1} = s].$$

Therefore, inductively,

$$\Pr[X_t^{(2j-1)} < n \mid X_0 = s] = \Pr[X_t^{(2j)} < n \mid X_0 = s].$$

But then

$$\begin{aligned} \mathbb{E}[T_{2j-1}(X_0)] &= \sum_{t \geq 0} \Pr[T_{2j-1}(X_0) > t] = \sum_{t \geq 0} \Pr[X_t^{(2j-1)} < n \mid X_0] \\ &= \sum_{t \geq 0} \Pr[X_t^{(2j)} < n \mid X_0] = \sum_{t \geq 0} \Pr[T_{2j}(X_0) > t] = \mathbb{E}[T_{2j}(X_0)]. \quad \blacktriangleleft \end{aligned}$$

2.2 Gossip Model

We now extend the previous analysis to the gossip model. Recall that in this model all agents are activated in parallel rounds. In such a round, all agents sample j other agents v_1, \dots, v_j u.a.r. Then they compute their new opinion as the majority opinion among the sample, breaking ties u.a.r. Here, the agents use the opinions of the other agents from the beginning of the round. At the end of the round (once all agents have computed the new opinion) all agents synchronously update their opinion to the new value.

Proof of Theorem 1 for the Gossip Model. In our extended analysis we use a coupling of the two parallel processes similarly to the coupling of one step of the sequential model. Observe that in process P_{2j} every agent samples $2j$ agents u.a.r., while in process P_{2j+1} every agent samples $2j + 1$ agents. Therefore, process P_{2j} makes $2j \cdot n$ random choices from $[n]$ in each round, while P_{2j+1} makes $(2j + 1) \cdot n$ random choices. We use the straight-forward coupling and define that the $2j$ choices of every agent u in P_{2j} are identical to the first $2j$ choices of agent u in process P_{2j+1} .

We now analyze the deviation of the two processes that stems from the $2j + 1$ th additional choice in process P_{2j+1} . Here we observe the following. In each round of process P_{2j} there are three disjoint sets of agents, M_a, M_b , and M_u . The sets M_a and M_b are comprised of agents that sample at least $j + 1$ agents of the majority opinion a and the minority opinion b , respectively. All other agents are in M_u . The agents in M_a will adopt opinion a at the end of the round in both processes: the $j + 1$ samples of opinion a is larger than the winning margin in both processes, which is j in P_{2j} and $(2j + 1)/2$ in P_{2j+1} . Analogously, the agents in M_b will adopt opinion b in both processes. Finally, the interesting group are the M_u agents. These agents have sampled a tie in process P_{2j} , meaning they have sampled j agents with opinion a and another j agents with opinion b . This means, in process P_{2j} all agents in M_u adopt either opinion a or opinion b with probability $1/2$ each. In process P_{2j+1} , however,

the $2j + 1$ th sample makes the decision. (Recall that in a process P_{2j+1} with an odd number of samples and $k = 2$ opinions no ties are possible.) Therefore, in process P_{2j+1} all agents in M_u adopt opinion a with probability α and opinion b with probability $(1 - \alpha)$.

Summarizing, we have the following. Due to the coupling of P_{2j} with P_{2j+1} , all agents in M_a or M_b behave exactly the same in both processes. We use $Z_a = |M_a|$ and $Z_b = |M_b|$ to denote their respective numbers. (Observe that Z_a and Z_b are the same in P_{2j} and P_{2j+1} due to the coupling.) In the following, we condition on the event that $|M_u| = m_u$. For the agents in M_u , the outcome can be described by binomial random variables: let Z_u^{2j} in process P_{2j} and Z_u^{2j+1} in process P_{2j+1} be the numbers of agents in M_u that adopt opinion a . Then

$$Z_u^{2j} \sim \text{Bin}(m_u, 1/2) \quad \text{and} \quad Z_u^{2j+1} \sim \text{Bin}(m_u, \alpha)$$

with $\alpha \geq 1/2$. Irrespective of the value of m_u we observe from well-known properties of binomial distributions that Z_u^{2j} is stochastically dominated by Z_u^{2j+1} , and hence

$$X_{t+1}^{(2j)} = Z_a + Z_u^{2j} \prec Z_a + Z_u^{2j+1} = X_{t+1}^{(2j+1)}.$$

The proof for the dominance of P_{2j} over P_{2j-1} uses similar definitions and follows analogously, with exception that M_u represents the agents that are undecided after the first $2j - 2$ draws and that Z_u^{2j-1} and Z_u^{2j} follow the same binomial distribution $\text{Bin}(m_u, \alpha)$.

The only ingredient that is left to prove is the monotonicity within one specific process. Indeed, if an analogous result as Lemma 5 in the sequential model can be proven, the path coupling argument follows the same lines as in the previous section.

► **Lemma 11.** *We find for P_{2j} and P_{2j+1} in the gossip model the following. Let X_t denote the number of agents with majority opinion at time t . If $s > s'$, then for all $d \in \{0, 1, \dots, n\}$*

$$\Pr[X_{t+1} \geq d \mid X_t = s] \geq \Pr[X_{t+1} \geq d \mid X_t = s'].$$

Proof. As before, let M_a and M_b denote the sets of agents that sample at least $j + 1$ agents of the majority opinion a and the minority opinion b , respectively. If $s > s'$, the monotonicity of the binomial distribution yields

$$|M_a|_{X_t=s} \succeq |M_a|_{X_t=s'} \quad \text{and} \quad |M_b|_{X_t=s} \preceq |M_b|_{X_t=s'}.$$

Therefore, the lemma follows from Strassen's theorem. ◀

Now the path coupling follows analogously to the previous section. ◀

2.3 Analysis of 3-Majority

In this section we analyze 3-Majority in the sequential model. We start with an overview of the proof of Theorem 2. The proof consists of three parts. The first part follows along the lines of the proof by Condon et al. [21] for the related approximate majority process in tri-molecular chemical reaction networks. It shows that we preserve the initial majority (assuming a bias of $\sqrt{n \log n}$) and reach a bias of ϵn within $O(n \log n)$ time w.h.p. (Recall that the *bias* is defined as the difference of the numbers of agents supporting opinion a and opinion b .) The proof is based on the following result for gambler's ruin from [32].

► **Lemma 12** (Asymmetric one-dimensional random walk, [32, XIV.2], version from [21]). *If we run an arbitrarily long sequence of independent trials, each with success probability at least p , then the probability that the number of failures ever exceeds the number of successes by b is at most $\left(\frac{1-p}{p}\right)^b$.*

23:12 On the Hierarchy of Distributed Majority Protocols

In the second part we use a drift analysis based on [40] to show that we reach consensus on the initial majority opinion quickly once we have a bias of order $\Omega(n)$. The proof is based on a carefully conducted drift-analysis, where we use the following fairly recent result.

► **Theorem 13** (Special case of Theorem 18 of [40]). *Let $\{Y_t\}_{t \geq 0}$ be a sequence of non-negative random variables with a finite state space $S \subset \mathbb{R}_{\geq 0}$ such that $0 \in S$. Define*

$$s_{\min} = \min(S \setminus \{0\}) \quad \text{and} \quad T = \inf \{t \geq 0 \mid Y_0 = 0\}.$$

If $Y_0 = s_0$ and there is $\delta > 0$ (independent from t) such that for all $s \in S \setminus \{0\}$ and all $t \geq 0$ we have

$$\mathbb{E}[Y_t - Y_{t+1} \mid Y_t = s] \geq \delta s,$$

then, for all $r \geq 0$,

$$\Pr \left[T > \left\lceil \frac{r + \log(s_0/s_{\min})}{\delta} \right\rceil \right] \leq e^{-r}.$$

In the third part we again show that the analysis from [21] is applicable in our setting if we do not have an initial bias. All three parts together prove the first statement of our theorem. The second statement follows from part one together with part two.

Part 1. We start with the first part. We follow along the lines of [21] and use Lemma 12 to show the following statement.

► **Lemma 14.** *Let Δ_t be the additive bias at time t . With probability $1 - e^{-\Omega(\Delta_t^2/n)}$, the bias Δ_t does not drop below $\Delta_t/2$ and increases to $\min\{2\Delta_t, n\}$ within $2n$ time steps.*

Proof. Let X_t denote the number of agents with the majority opinion at time t and let $Y_t = n - X_t$ denote the number of agents with the minority opinion at time t . We analyze our process as a variant of gamblers' ruin and apply Lemma 12. We only consider *productive* steps in which the number of agents of a specific opinion changes. For $X_t \in (\frac{n}{2}, \frac{n}{2} + \varepsilon n)$ it holds that $\Pr[X_{t+1} \neq X_t] = \Omega(1)$ and hence conditioning on productive steps only increases the constants hidden in the asymptotic notation.

In each productive step, the success probability reads $p = \Pr[X_{t+1} > X_t \mid X_{t+1} \neq X_t]$ and the failure probability reads $1 - p = \Pr[X_{t+1} < X_t \mid X_{t+1} \neq X_t]$. Let $\Delta_t = X_t - \frac{n}{2}$ denote the bias at time t . We have for any Δ that

$$\frac{1-p}{p} = \frac{2(\frac{1}{2} - \frac{\Delta}{n}) - 3(\frac{1}{2} - \frac{\Delta}{n})^3 + (\frac{1}{2} - \frac{\Delta}{n})}{2(\frac{1}{2} - \frac{\Delta}{n})^4 - 5(\frac{1}{2} - \frac{\Delta}{n})^3 + 3(\frac{1}{2} - \frac{\Delta}{n})^2} < 1 - 16\frac{\Delta}{n}. \quad (1)$$

Unfortunately, the success probabilities vary over time as they depend on the bias. We proceed to bound the probabilities from below.

Let Δ_0 be the bias at time $t = 0$ and let \mathcal{R} denote the following event: during $2n$ productive steps we always have at least half of the initial bias, i.e., $\mathcal{R} = \{\forall 1 \leq i \leq 2n: \Delta_i \geq \Delta_0/2\}$. From Lemma 12 we get with $b = \Delta_0/2$ that

$$\Pr[\mathcal{R}] \geq 1 - e^{-\Omega(\Delta_0^2/n)}. \quad (2)$$

Similarly to [21], we couple the productive steps of the 3-Majority process with a biased random walk with (fixed) success probability $p > \frac{1}{2} + \frac{\Delta_0}{4n}$. As (1) is monotonously decreasing in Δ , the number of steps required by the biased random walk to increase the bias stochastically dominates the number of steps that 3-Majority requires. It follows from Chernoff bounds that the random walk reaches $2\Delta_0$ within $2n$ time steps with probability $1 - e^{-\Omega(\Delta_0^2/n)}$. Together with (2) the statement follows. ◀

We now use Lemma 14 and show that if there is a small bias of size $\sqrt{n \log n}$ then within $O(n \log n)$ rounds there will be a bias of size $\Omega(n)$ w.h.p.

► **Corollary 15.** *Assume $X_0 = \frac{n}{2} + \sqrt{n \log n}$. Then there is a time $t = O(n \log n)$ such that $X_t > \frac{1+\varepsilon}{2}n$ for some constant $\varepsilon > 0$ w.h.p. Moreover, the initial majority opinion is preserved.*

Proof. The proof follows by applying Lemma 14 $O(\log n)$ times. We remark that the initial majority opinion is preserved since the random walk modeling the bias never returns to zero. ◀

Part 2. We now show the second part, where we prove that the process converges within $O(n \log n)$ further steps once we have a bias of εn . Let Y_t denote the number of agents of the *minority* opinion at time t and assume that $Y_0 \leq \frac{n}{2} - \varepsilon n$. In a first step, we claim that the process will not improve the minority opinion severely if only $Cn \log n$ steps are conducted for some large constant C .

► **Lemma 16.** *Assume $Y_0 \leq \frac{n}{2} - \varepsilon n$. Then there is a time $t = O(n \log n)$ such that $Y_t = 0$ w.h.p. Moreover, $Y_{t'} \leq \frac{1-\varepsilon}{2}n$ for all $t' \leq t$.*

Proof. We start the proof by showing the following claim:

▷ **Claim.** $Y_{t'} \leq \frac{1-\varepsilon}{2}n$ for all $t' = O(n \log n)$ w.h.p.

This is an immediate consequence of the following coupling. Let R_t be the (unbiased) random walk on \mathbb{Z} . It is a well known fact that after T steps the random walk R_t has distance at most $O(\log^2 n \cdot \sqrt{n})$ from the origin w.h.p. By construction, $R_t \preceq Y_t$ and the claim follows.

We now calculate $\mathbb{E}[Y_t - Y_{t+1} \mid Y_t = s]$ for P_3 in the sequential model. Given $Y_t = s$, let $p_s(a, b)$ be the probability to increase the minority opinion by one and let $p_s(b, a)$ be the probability to decrease the minority opinion by one. Then,

$$\mathbb{E}[Y_t - Y_{t+1} \mid Y_t = s] = p_s(b, a) - p_s(a, b).$$

We observe

$$p_s(a, b) = \frac{n-s}{n} \Pr\left[\text{Bin}\left(3, \frac{s}{n}\right) \geq 2\right], \quad p_s(b, a) = \frac{s}{n} \Pr\left[\text{Bin}\left(3, \frac{s}{n}\right) \leq 1\right],$$

and therefore

$$\frac{p_s(b, a) - p_s(a, b)}{b} = \frac{2s^2 - 3sn + n^2}{n^3}.$$

We define $\delta_s = \frac{p_s(b, a) - p_s(a, b)}{s}$ and observe

$$\delta_s - \delta_{s-1} = \frac{4s - 3n - 2}{n^3} < 0 \quad \text{if } s \leq 0.75n.$$

Therefore, since $s \leq \frac{1-\varepsilon}{2}n$ by the previous claim, δ_s is monotonously decreasing in s . Furthermore,

$$\delta_{\frac{1-\varepsilon}{2}n} = \frac{(1 + \varepsilon/2)\varepsilon}{4n} \quad \text{and} \quad \delta_1 = n^{-1} + O(n^{-2}).$$

Thus, we apply Theorem 13 with

$$\delta = \frac{(1 + \varepsilon/2)\varepsilon}{4n}, \quad s_0 = \left(\frac{1}{2} - \varepsilon\right)n, \quad r = \log n, \quad \text{and} \quad s_{\min} = 1$$

and the statement follows. ◀

Part 3. It remains to show the third part of the proof. We observe the following. We use the same *checkpoint states* g_j as in [21] where $g_0 = 0$ and $g_j = 2^{j+3} \cdot \sqrt{n}$. A checkpoint state can be intuitively described as follows. We let P_3 run in packages of $2n$ productive update steps and monitor the majority opinion. Suppose we are in checkpoint state $g_1 = 8\sqrt{n}$. After $2n$ productive updates, Lemma 14 guarantees that with probability at least $1 - 1/(2^j + O(1))$ the majority opinion exceeds g_2 . Now we interpret this process as a (biased) random walk on the checkpoint states $\{g_j\}_j$ in which every conducted step consists of $2n$ productive update steps of 3-Majority. Analogously to the analysis of [21], it holds that

1. the transition between checkpoint states g_0 and g_1 has probability $\Omega(1)$, and
2. for $j \geq 1$ the transition between checkpoint states g_j and $g_j + 1$ has probability at least $1 - 1/(2^j + O(1))$.

As in [21], the first statement follows from a coupling with an unbiased random walk, and the second statement follows from Lemma 14. It follows from the analysis in [21, Section 3.2] that 3-Majority reaches a bias of $\sqrt{n \log n}$ within $O(n \log n)$ time. This proof is based on a careful trade-off between the geometrically increasing success probability $1 - 1/(2^j + O(1))$ to get into the next checkpoint state and the number of trials that are necessary to indeed reach the next state instead of falling back.

With all three parts, we are now ready to put everything together and prove Theorem 2.

Proof of Theorem 2. Assume there is no bias. From the analysis in [21] we obtain (see above) that we reach a bias of size $\sqrt{n \log n}$ within $O(n \log n)$ time w.h.p. From Corollary 15 we obtain that within further $O(n \log n)$ time the bias is amplified to ϵn for some constant $\epsilon > 0$ w.h.p. Finally, from the drift analysis in Lemma 16 we get that we converge in further $O(n \log n)$ time once we have a constant-factor bias w.h.p. Together, this shows the first part of the theorem.

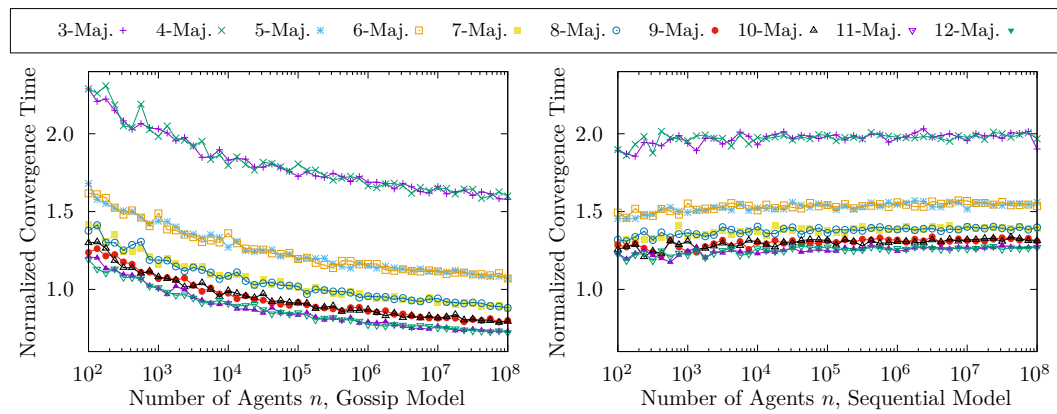
The second part of the theorem follows from Lemma 14 and Lemma 16, where we observe that the initial majority opinion is preserved w.h.p. This concludes the proof. ◀

3 Empirical Analysis

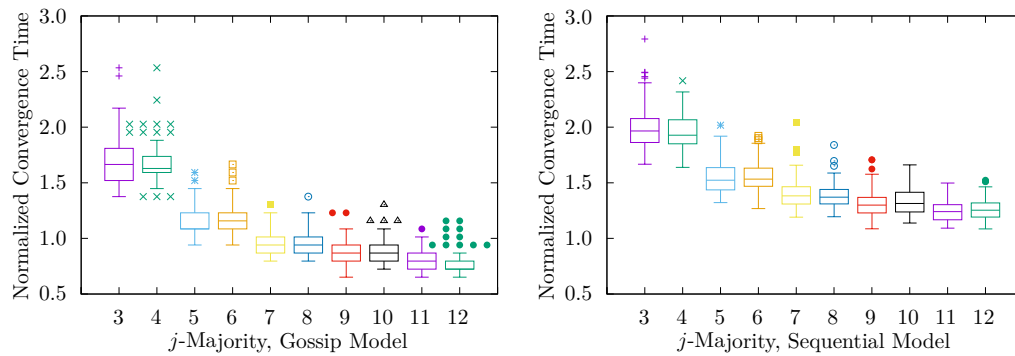
In this section we present simulation results to support our theoretical findings. Our simulation software is implemented in the C++ programming language. As a source of randomness it uses the Mersenne Twister `mt19937_64` provided by the C++11 `<random>` library. Our simulations have been carried out on machines with two Intel(R) Xeon(R) E5-2630 v4 CPUs and 128 GiB of memory each running the Linux 5.13 kernel. The simulation software and all required tools to reproduce our plots are publicly available in our Github repository.

In Figure 3 we plot the required number of rounds until j -Majority converges when each opinion is initially supported $n/2$ agents. The data show the average convergence time over 100 independent simulation runs for $j = 3, \dots, 12$. The number of agents n is shown on the x -axis, and the normalized convergence time is shown on the y -axis. The left plot shows the data for the gossip model, where the normalization means that the required number of rounds is divided by $\log n$. The right plot shows the data for the sequential model, where the normalization means that the required number of interactions is divided by $n \log n$.

Our empirical data confirm our theoretical findings. In particular, we observe that the processes exhibit a running time of $\Theta(\log n)$ rounds (gossip model) or $\Theta(n \log n)$ interactions (sequential model) for the values of j we consider. Furthermore, we clearly see that $\mathbb{E}[T_{2j+2}(X_0)] = \mathbb{E}[T_{2j+1}(X_0)]$ (i.e., 3-Majority converges as quickly as 4-Majority, 5-Majority converges as quickly as 6-Majority, and so on) and $\mathbb{E}[T_{2j+1}(X_0)] \leq \mathbb{E}[T_{2j}(X_0)]$ (i.e.,



■ **Figure 3** Average convergence time of j -Majority without initial bias and $j = 3, \dots, 12$ normalized over $\log n$ (gossip model) or $n \log n$ (sequential model). Each data point shows the average of 100 independent runs. The left plot shows the gossip model and the right plot shows the sequential model.



■ **Figure 4** Boxplots for the normalized convergence time of j -Majority without initial bias. The plots show details of the distribution of the same data as in Figure 3 for $n = 10^6$.

5-Majority is faster than 4-Majority, 7-Majority is faster than 6-Majority, and so on). This empirically confirms our results from Theorem 1 for both models, and it shows that the known results from the gossip model for 3-Majority [35] carry over to the sequential model as predicted in Theorem 2.

In the left plot in Figure 3 for the gossip model we additionally observe that the required number of rounds to reach consensus is slightly larger for smaller values of n . This appears to be a consequence of the discrete rounds in the synchronous model: the observed deviation scales as $O(1/\log n)$, which is of the same size as the rounding error that arises when reporting the running time in discrete rounds of n interactions each.

Finally, in Figure 4 we show additional detail for the distribution of the convergence times of the j -Majority processes with $n = 10^6$ and $j = 3, \dots, 12$. Our boxplots show that the running times are strongly concentrated around the mean, and the constants hidden in the asymptotic analysis are small: the running time is less than $3 \log n$ rounds in the gossip model and less than $3n \log n$ interactions in the sequential model. The small constants hint at the practical applicability of the simple 3-Majority process.

4 Conclusions and Open Problems

We analyze the family of j -Majority processes in two communication models with parallel and sequential activations. In both models our results affirmatively answer an open question from [12] for the case of two opinions and prove the existence of a *hierarchy*: our results show the stochastic dominance of the convergence time of the $(j + 1)$ -Majority process over the j -Majority process. For 3-Majority in the sequential model we show an asymptotically optimal bound of $O(n \log n)$ sequential activations. This matches the well-known bounds for the corresponding process in the gossip model.

An open question is whether a similar hierarchy exists for *lazy* processes where agents keep their previous opinion if there is a tie among the sampled opinions. A coupling between 3-Majority and the (lazy) TwoCHOICES process was analyzed in [12]. However, their general framework cannot be adapted to lazy processes for larger value of j : their analysis requires so-called *AC-Processes* in which the next state of an agent depends only on the global opinion distribution but not on the agent's current state. This is obviously not the case for lazy processes. Note that our analysis also cannot be applied to lazy processes directly: Lemmas 8 and 10 do not hold for lazy processes.

Another interesting open question considers the *communication complexity* of a protocol instead which counts the number of interactions. Note that in j -Majority each activated agent interacts with j agents. It would be interesting to rigorously analyze the trade-off between the convergence time and the communication complexity.

Finally, the most interesting open question is whether similar results can be shown for more than two opinions. Unfortunately, our majoritization-based approach does not generalize to $k > 2$. The main reason is that natural monotonicity properties do not hold: the probability to increase the majority opinion does not only depend on the size of the majority opinion itself but instead on the entire opinion distribution. This aligns well with a conjecture from [12] that states that counterexamples exist for any majoritization attempt that uses a total order on opinion state vectors. We believe that in order to show a hierarchy of majority protocols for more than two opinions different techniques will be needed.

References

- 1 Dan Alistarh, James Aspnes, David Eisenstat, Rati Gelashvili, and Ronald L. Rivest. Time-Space Trade-offs in Population Protocols. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017*, pages 2560–2579. SIAM, 2017. doi:10.1137/1.9781611974782.169.
- 2 Dan Alistarh, James Aspnes, and Rati Gelashvili. Space-Optimal Majority in Population Protocols. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018*, pages 2221–2239. SIAM, 2018. doi:10.1137/1.9781611975031.144.
- 3 Dan Alistarh, Rati Gelashvili, and Milan Vojnovic. Fast and Exact Majority in Population Protocols. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015*, pages 47–56. ACM, 2015. doi:10.1145/2767386.2767429.
- 4 Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Comput.*, 18(4):235–253, 2006. doi:10.1007/s00446-005-0138-3.
- 5 Dana Angluin, James Aspnes, and David Eisenstat. A simple population protocol for fast robust approximate majority. *Distributed Comput.*, 21(2):87–102, 2008. doi:10.1007/s00446-008-0059-z.
- 6 Gregor Bankhamer, Petra Berenbrink, Felix Biermeier, Robert Elsässer, Hamed Hosseinpour, Dominik Kaaser, and Peter Kling. Fast Consensus via the Unconstrained Undecided State Dynamics. In *Proceedings of the 2022 ACM-SIAM Symposium on Discrete Algorithms, SODA 2022*, pages 3417–3429. SIAM, 2022. doi:10.1137/1.9781611977073.135.

- 7 Gregor Bankhamer, Robert Elsässer, Dominik Kaaser, and Matjaz Krnc. Positive Aging Admits Fast Asynchronous Plurality Consensus. In *PODC '20: ACM Symposium on Principles of Distributed Computing*, pages 385–394. ACM, 2020. doi:10.1145/3382734.3406506.
- 8 Luca Becchetti, Andrea E. F. Clementi, and Emanuele Natale. Consensus Dynamics: An Overview. *SIGACT News*, 51(1):58–104, 2020. doi:10.1145/3388392.3388403.
- 9 Luca Becchetti, Andrea E. F. Clementi, Emanuele Natale, Francesco Pasquale, and Riccardo Silvestri. Plurality Consensus in the Gossip Model. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015*, pages 371–390. SIAM, 2015. doi:10.1137/1.9781611973730.27.
- 10 Luca Becchetti, Andrea E. F. Clementi, Emanuele Natale, Francesco Pasquale, Riccardo Silvestri, and Luca Trevisan. Simple dynamics for plurality consensus. *Distributed Comput.*, 30(4):293–306, 2017. doi:10.1007/s00446-016-0289-4.
- 11 Stav Ben-Nun, Tsvi Kopelowitz, Matan Kraus, and Ely Porat. An $O(\log^{3/2} n)$ Parallel Time Population Protocol for Majority with $O(\log n)$ States. In *PODC '20: ACM Symposium on Principles of Distributed Computing*, pages 191–199. ACM, 2020. doi:10.1145/3382734.3405747.
- 12 Petra Berenbrink, Andrea E. F. Clementi, Robert Elsässer, Peter Kling, Frederik Mallmann-Trenn, and Emanuele Natale. Ignore or Comply?: On Breaking Symmetry in Consensus. In *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017*, pages 335–344. ACM, 2017. doi:10.1145/3087801.3087817.
- 13 Petra Berenbrink, Amin Coja-Oghlan, Oliver Gebhard, Max Hahn-Klimroth, Dominik Kaaser, and Malin Rau. On the Hierarchy of Distributed Majority Protocols. *CoRR*, abs/2205.08203, 2022. doi:10.48550/arXiv.2205.08203.
- 14 Petra Berenbrink, Robert Elsässer, Tom Friedetzky, Dominik Kaaser, Peter Kling, and Tomasz Radzik. A Population Protocol for Exact Majority with $O(\log^{5/3} n)$ Stabilization Time and $\Theta(\log n)$ States. In *32nd International Symposium on Distributed Computing, DISC 2018*, pages 10:1–10:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.DISC.2018.10.
- 15 Petra Berenbrink, Robert Elsässer, Tom Friedetzky, Dominik Kaaser, Peter Kling, and Tomasz Radzik. Time-space trade-offs in population protocols for the majority problem. *Distributed Comput.*, 34(2):91–111, 2021. doi:10.1007/s00446-020-00385-0.
- 16 Petra Berenbrink, Tom Friedetzky, George Giakkoupis, and Peter Kling. Efficient Plurality Consensus, Or: the Benefits of Cleaning up from Time to Time. In *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016*, pages 136:1–136:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.ICALP.2016.136.
- 17 Petra Berenbrink, George Giakkoupis, Anne-Marie Kermarrec, and Frederik Mallmann-Trenn. Bounds on the Voter Model in Dynamic Networks. In *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016*, pages 146:1–146:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.ICALP.2016.146.
- 18 Andreas Bilke, Colin Cooper, Robert Elsässer, and Tomasz Radzik. Brief Announcement: Population Protocols for Leader Election and Exact Majority with $O(\log^2 n)$ States and $O(\log^2 n)$ Convergence Time. In *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017*, pages 451–453. ACM, 2017. doi:10.1145/3087801.3087858.
- 19 Keren Censor-Hillel, Bernhard Haeupler, Jonathan A. Kelner, and Petar Maymounkov. Global computation in a poorly connected world: fast rumor spreading with no dependence on conductance. In *Proceedings of the 44th Symposium on Theory of Computing Conference, STOC 2012*, pages 961–970. ACM, 2012. doi:10.1145/2213977.2214064.
- 20 Andrea E. F. Clementi, Mohsen Ghaffari, Luciano Gualà, Emanuele Natale, Francesco Pasquale, and Giacomo Scornavacca. A Tight Analysis of the Parallel Undecided-State Dynamics with Two Colors. In *43rd International Symposium on Mathematical Foundations of Computer Science, MFCS 2018*, pages 28:1–28:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.MFCS.2018.28.

- 21 Anne Condon, Monir Hajiaghayi, David G. Kirkpatrick, and Ján Manuch. Approximate majority analyses using tri-molecular chemical reaction networks. *Nat. Comput.*, 19(1):249–270, 2020. doi:10.1007/s11047-019-09756-4.
- 22 Colin Cooper, Robert Elsässer, Hirotaka Ono, and Tomasz Radzik. Coalescing random walks and voting on graphs. In *ACM Symposium on Principles of Distributed Computing, PODC '12*, pages 47–56. ACM, 2012. doi:10.1145/2332432.2332440.
- 23 Colin Cooper, Robert Elsässer, and Tomasz Radzik. The Power of Two Choices in Distributed Voting. In *Automata, Languages, and Programming – 41st International Colloquium, ICALP 2014*, pages 435–446. Springer, 2014. doi:10.1007/978-3-662-43951-7_37.
- 24 Colin Cooper, Robert Elsässer, Tomasz Radzik, Nicolas Rivera, and Takeharu Shiraga. Fast Consensus for Voting on General Expander Graphs. In *Distributed Computing – 29th International Symposium, DISC 2015*, pages 248–262. Springer, 2015. doi:10.1007/978-3-662-48653-5_17.
- 25 Colin Cooper, Tomasz Radzik, Nicolas Rivera, and Takeharu Shiraga. Fast Plurality Consensus in Regular Expanders. In *31st International Symposium on Distributed Computing, DISC 2017*, pages 13:1–13:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.DISC.2017.13.
- 26 Benjamin Doerr, Leslie Ann Goldberg, Lorenz Minder, Thomas Sauerwald, and Christian Scheideler. Stabilizing consensus with the power of two choices. In *SPAA 2011: Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 149–158. ACM, 2011. doi:10.1145/1989493.1989516.
- 27 David Doty and Mahsa Eftekhari. Efficient Size Estimation and Impossibility of Termination in Uniform Dense Population Protocols. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019*, pages 34–42. ACM, 2019. doi:10.1145/3293611.3331627.
- 28 David Doty, Mahsa Eftekhari, Leszek Gasieniec, Eric E. Severson, Przemyslaw Uznanski, and Grzegorz Stachowiak. A time and space optimal stable population protocol solving exact majority. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021*, pages 1044–1055. IEEE, 2021. doi:10.1109/FOCS52979.2021.00104.
- 29 David Doty and David Soloveichik. Stable leader election in population protocols requires linear time. *Distributed Comput.*, 31(4):257–271, 2018. doi:10.1007/s00446-016-0281-z.
- 30 Moez Draief and Milan Vojnovic. Convergence Speed of Binary Interval Consensus. *SIAM J. Control. Optim.*, 50(3):1087–1109, 2012. doi:10.1137/110823018.
- 31 Robert Elsässer, Tom Friedetzky, Dominik Kaaser, Frederik Mallmann-Trenn, and Horst Trinker. Brief Announcement: Rapid Asynchronous Plurality Consensus. In *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017*, pages 363–365. ACM, 2017. doi:10.1145/3087801.3087860.
- 32 William Feller. *An Introduction to Probability Theory and Its Applications*, volume 1. Wiley, 3rd edition, 2008.
- 33 Pierre Fraigniaud and Emanuele Natale. Noisy rumor spreading and plurality consensus. *Distributed Comput.*, 32(4):257–276, 2019. doi:10.1007/s00446-018-0335-5.
- 34 Leszek Gasieniec, David D. Hamilton, Russell Martin, Paul G. Spirakis, and Grzegorz Stachowiak. Deterministic Population Protocols for Exact Majority and Plurality. In *20th International Conference on Principles of Distributed Systems, OPODIS 2016*, pages 14:1–14:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.OPODIS.2016.14.
- 35 Mohsen Ghaffari and Johannes Lengler. Nearly-Tight Analysis for 2-Choice and 3-Majority Consensus Dynamics. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018*, pages 305–313. ACM, 2018. doi:10.1145/3212734.3212738.
- 36 Mohsen Ghaffari and Merav Parter. A Polylogarithmic Gossip Algorithm for Plurality Consensus. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016*, pages 117–126. ACM, 2016. doi:10.1145/2933057.2933097.

- 37 Yehuda Hassin and David Peleg. Distributed Probabilistic Polling and Applications to Proportionate Agreement. *Inf. Comput.*, 171(2):248–268, 2001. doi:10.1006/inco.2001.3088.
- 38 Varun Kanade, Frederik Mallmann-Trenn, and Thomas Sauerwald. On coalescence time in graphs: When is coalescing as fast as meeting?: Extended Abstract. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019*, pages 956–965. SIAM, 2019. doi:10.1137/1.9781611975482.59.
- 39 Adrian Kosowski and Przemyslaw Uznanski. Brief Announcement: Population Protocols Are Fast. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018*, pages 475–477. ACM, 2018. doi:10.1145/3212734.3212788.
- 40 Johannes Lengler. Drift Analysis. In Benjamin Doerr and Frank Neumann, editors, *Theory of Evolutionary Computation: Recent Developments in Discrete Optimization*, pages 89–131. Springer, 2020. doi:10.1007/978-3-030-29414-4_2.
- 41 George B. Mertzios, Sotiris E. Nikolettseas, Christoforos L. Raptopoulos, and Paul G. Spirakis. Determining Majority in Networks with Local Interactions and Very Small Local Memory. In *Automata, Languages, and Programming – 41st International Colloquium, ICALP 2014*, pages 871–882. Springer, 2014. doi:10.1007/978-3-662-43948-7_72.
- 42 Yves Mocquard, Emmanuelle Anceaume, James Aspnes, Yann Busnel, and Bruno Sericola. Counting with Population Protocols. In *14th IEEE International Symposium on Network Computing and Applications, NCA 2015*, pages 35–42. IEEE Computer Society, 2015. doi:10.1109/NCA.2015.35.
- 43 Toshio Nakata, Hiroshi Imahayashi, and Masafumi Yamashita. A probabilistic local majority polling game on weighted directed graphs with an application to the distributed agreement problem. *Networks*, 35(4):266–273, 2000. doi:10.1002/1097-0037(200007)35:4<266::AID-NET5>3.0.CO;2-4.
- 44 Emanuele Natale and Iliad Ramezani. On the Necessary Memory to Compute the Plurality in Multi-agent Systems. In *Algorithms and Complexity – 11th International Conference, CIAC 2019*, pages 323–338. Springer, 2019. doi:10.1007/978-3-030-17402-6_27.
- 45 Grant Schoenebeck and Fang-Yi Yu. Consensus of Interacting Particle Systems on Erdős-Rényi Graphs. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018*, pages 1945–1964. SIAM, 2018. doi:10.1137/1.9781611975031.127.
- 46 Volker Strassen. The existence of probability measures with given marginals. *The Annals of Mathematical Statistics*, 36(2):423–439, 1965.

Communication-Efficient BFT Using Small Trusted Hardware to Tolerate Minority Corruption

Sravya Yandamuri ✉

Duke University, Durham, NC, USA

Ittai Abraham ✉

VMware Research, Herzliya, Israel

Kartik Nayak ✉

Duke University, Durham, NC, USA

Michael K. Reiter ✉

Duke University, Durham, NC, USA

Abstract

Agreement protocols for partially synchronous networks tolerate fewer than one-third Byzantine faults. If parties are equipped with trusted hardware that prevents equivocation, then fault tolerance can be improved to fewer than one-half Byzantine faults, but typically at the cost of increased communication complexity. In this work, we present results that use small trusted hardware without worsening communication complexity assuming the adversary controls a fraction of the network that is less than one-half. In particular, we show a version of HotStuff that retains linear communication complexity in each view, leveraging trusted hardware to tolerate a minority of corruptions. Our result uses expander graph techniques to achieve efficient communication in a manner that may be of independent interest.

2012 ACM Subject Classification Theory of computation → Communication complexity

Keywords and phrases communication complexity, consensus, trusted hardware

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2022.24

Related Version *Full Version:* <https://eprint.iacr.org/2021/184.pdf> [57]

Funding This research was supported in part by NIFA Award 2021-67021-34252 and by VMware and Novi research grants.

1 Introduction

Byzantine fault tolerant (BFT) consensus is an important problem in distributed computing. It has received revived interest as the foundation of decentralized ledgers or blockchains. The goal of BFT consensus is for a set of parties to agree on a value (or a sequence of values) even if a fraction of the parties are Byzantine (malicious). To rule out trivial solutions, these protocols additionally need to satisfy a validity constraint which, depending on the setting, is a function of the input of a designated party or all parties or external clients.

The number of faults tolerated by a BFT protocol depends on the network assumptions between parties, the use of cryptography, and other assumptions. In particular, it is known that to maintain safety when the system is asynchronous, without additional assumptions, one cannot tolerate one-third or more Byzantine faults [21]. However, tolerating fewer than one-third Byzantine faults may not be enough for some applications. There are two known approaches to increase this fault threshold. The first approach is to give up safety in asynchrony. One can tolerate fewer than one-half Byzantine faults by assuming synchrony (any message sent by an honest party reaches its destination within a bounded network delay) and some method to limit the ability of the adversary to simulate honest parties (for example assuming a PKI or proof-of-work) [22, 28, 4, 13, 20, 38, 31]. Protocols using synchrony



© Sravya Yandamuri, Ittai Abraham, Kartik Nayak, and Michael K. Reiter;
licensed under Creative Commons License CC-BY 4.0

26th International Conference on Principles of Distributed Systems (OPODIS 2022).

Editors: Eshcar Hillel, Roberto Palmieri, and Etienne Rivière; Article No. 24; pp. 24:1–24:23



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

increase the fault threshold by detecting equivocation (assuming signatures) and making deductions based on the absence of messages from other parties (e.g., [42, 4]). The second approach lets the adversary delay messages but limits its ability to corrupt by assuming the existence of a *trusted hardware*. The adversary cannot tamper with this hardware even if it fully controls the node. At a high-level, the hardware provides non-equivocation guarantees, essentially transforming Byzantine failures to omission failures and hence improving the fault tolerance threshold to one-half (e.g., [21, 17, 27, 53]) in partial synchrony and asynchrony.

In this work, we focus on the use of *small trusted hardware* primitives to tolerate a minority Byzantine corruption and stay safe in asynchrony. Specifically, each node is equipped with hardware that implements the abstraction of an “append-only log,” the contents to which it can attest using a *conventional* digital signature with a key that it holds. This capability is supported by numerous *existing*, trusted add-ons (TPMs [1], YubiKeys [49], smartcards, etc.) and is far simpler to implement than secure enclaves for arbitrary computation, as Intel SGX [19] attempts to do – but arguably fails [40, 52, 56, 55, 15].

The use of such small trusted hardware to boost fault tolerance was explored in A2M [17] and TrInc [32], which specifically improved PBFT [11]. However, this came at the expense of an $O(n^3)$ communication complexity per view for consensus among n parties, measured as the (expected) number of words that all honest parties send. On the other hand, in the standard setting, we have recently seen considerable progress in improving communication complexity of consensus protocols. In particular, HotStuff [58] achieves linear communication complexity per view under partial synchrony and VABA [5] achieves the optimal $O(n^2)$ communication complexity under asynchrony. A natural question is whether fault tolerance can be boosted (but communication costs retained) in these protocols using small trusted hardware. In this work, we answer these questions affirmatively for a corruption threshold $t \leq (\frac{1}{2} - \epsilon)n$ for an arbitrarily small ϵ , which we term a *minority corruption* adversary. In the following, we describe our results and the key techniques used to achieve these results in the partially synchronous and asynchronous settings. We include $(\frac{n}{2} + 1)$ -Provable-Broadcast, a broadcast primitive with linear communication complexity that can be used to improve the fault tolerance of such protocols, as well as a version of HotStuff that uses this primitive to withstand minority corruption. In the full version of our work [57], we present an expected linear communication complexity and constant round view synchronization protocol that is a strict generalization of that in [39] and withstands minority Byzantine corruption.

1.1 HotStuff-M: HotStuff with Minority Corruption

Our result improves HotStuff to tolerate a $t \leq (\frac{1}{2} - \epsilon)n$ corruption while still retaining its linear communication complexity per view. In particular, we show the following result:

► **Theorem 1** (HotStuff-M). *For any $\epsilon > 0$, there exists a primary-backup based BFT consensus protocol with $O(n)$ communication complexity per view consisting of n parties, each having a small trusted hardware, such that $t \leq (\frac{1}{2} - \epsilon)n$ of the parties are Byzantine.*

HotStuff is a primary-backup protocol that progresses in a sequence of views, each having a designated leader (primary) and consisting of a sequence of phases. HotStuff routes all messages (votes) through the leader independent of whether the communication is within the view or across views, while keeping the message size $O(1)$ for a total of $O(n)$ communication per view. To achieve this, HotStuff crucially relies on threshold signatures to aggregate votes of individual parties into an $O(1)$ -sized message; these signatures act as a proof for parties in subsequent phases/views to determine whether they should vote in that phase. Within a view, HotStuff maintains safety due to the fact that if a leader has a threshold signature

for a given proposal, a majority of the honest parties voted for that proposal. By quorum intersection, and the fact that an honest party votes only once in a given phase, a conflicting proposal cannot have a valid threshold signature.

We improve the resilience of HotStuff from one-third to $\frac{1}{2} - \epsilon$ while keeping a total of $O(n)$ communication per view using small trusted hardware in a partially synchronous network. In the minority corruption model, the presence of a threshold signature on a proposal no longer implies that a majority of the honest parties voted for that proposal, and is therefore insufficient for safety against a Byzantine adversary. The key property that we need from the hardware is its ability to maintain an append-only log that can be used to provide a non-equivocation property, i.e., if the hardware produces a *signed attestation* at a given position in the log, then the party cannot produce a valid signed attestation for a different value at the same position. Thus, intuitively, if $\frac{n}{2} + 1$ parties attest to a value at a position, then no other value can have $\frac{n}{2} + 1$ attestations. However, while a party's attestation from its trusted hardware is sufficient for safety, receiving such proofs from $O(n)$ parties produces an $O(n)$ -sized proof sent to the leader of the view. Since the leader uses this proof in a subsequent round, and these proofs cannot be compressed as they are not threshold signatures, this will grow the communication complexity to $O(n^2)$ per view.

Instead of sending the attestations directly to the leader, our solution relies on diffusing the attestations to a constant number of parties, called its *neighbors*. A party *votes* if it receives attestations from a threshold of its neighbors. This vote can be a threshold signature share, which can eventually be combined by the leader to an $O(1)$ -sized voting proof. Why does this work? We connect parties to each other using a constant-degree expander graph. Informally, to send a (non-attested) vote, a party just needs to verify that a constant fraction of its neighbors have attested. The specific construction of our expander guarantees that if a small ϵn -sized fraction of honest parties have voted for a proposal, then a majority of the parties have attested to that proposal. Thus, if a leader receives votes from $t + \epsilon n = \frac{n}{2}$ parties, at least ϵn are honest, and they can vet attestations from a majority of parties; this guarantees safety within a view. To ensure liveness, the expander graph is also parameterized such that if all honest parties attest, more than $\frac{n}{2}$ parties vote. We note that expander graphs have been used in consensus protocols before [30, 37, 16], although only in the context of synchronous protocols and exploiting a different set of expander properties.

The above arguments do not suffice for safety across views. The key mechanism that ensures safety across views in HotStuff is that if an honest party commits a value v in a given view, they received a threshold signature on v from the previous phase of this view, indicating that a majority of the honest parties “locked” on v in this view. These locked parties will not vote for a different value in later views, and a $v' \neq v$ will never gain a threshold signature in a subsequent view (and will therefore not be committed by an honest party). In the minority corruption model, while our trusted hardware disallows appending different values at the same position (equivocation), we cannot enforce the conditions under which a Byzantine party appends a value to their log. We also cannot enforce that a Byzantine party presents the latest state of its log as necessary. This can potentially result in a safety or liveness violation; e.g., even if a party “locked” on v in a given view by attesting to v , it can present a state that does not involve this attestation. In the original HotStuff protocol, an honest leader waits to hear the value from the highest view in which parties have stored a value during the second phase of that view for liveness. Since, in the minority corruption model, a leader cannot wait to hear from a majority of the honest parties, it must rely on Byzantine parties to present the correct state of their logs. Of course, this could be fixed by requiring a party to always present the entire contents of the log in its trusted hardware, but the

communication complexity would grow (unbounded) with the number of views. Instead, we use a combination of techniques including: multiple logs, one for each phase of the protocol ($O(1)$ total); tying log positions to view numbers; and using one attestation to present the end state of all logs. We elaborate on these techniques in Section 4 when we describe the protocol.

Future work and open questions. Our protocol tolerates $t \leq (\frac{1}{2} - \epsilon)n$ faults for an arbitrarily small ϵ , but addressing the communication complexity with trusted hardware and *optimal resilience* is still an open question. Also, our solution centers on a novel use of expander graphs. While the solution obtains optimal asymptotic communication complexity, the constants incurred due to the use of expanders may not be realistic for practical values of n . Solving this problem for smaller values of n is an interesting open question.

2 Model and Preliminaries

We consider n parties (a.k.a. processors) p_1, \dots, p_n connected by a reliable, authenticated, fully connected network, where up to $t \leq (1/2 - \epsilon)n$ parties may be corrupted by an adversary for $\epsilon > 0$. The corrupted parties are Byzantine and may behave arbitrarily. All the correct (honest) parties follow the protocol specification. We consider a partially synchronous network, where after an unknown period of time called Global Stabilization Time (GST), every message will arrive within a known bounded delay. We solve the validated Byzantine Agreement problem:

► **Definition 2** (Validated Byzantine Agreement). *A validated Byzantine agreement protocol among n parties tolerating a maximum of t faults satisfies the following properties:*

(Agreement/Safety) *If any two honest parties output values v and v' , then $v = v'$.*

(Validity) *If an honest party outputs v , then v is an externally valid value, i.e., $\text{ext-valid}(v) = \text{true}$.*

(Termination/Liveness under partial synchrony) *If all honest parties start with an externally valid value, then after GST, all honest parties will output a value within a bounded time.*

Following Cachin et al. [10], the definition has an external validity property. Such a property can be useful in the context of state machine replication (SMR) where $\text{ext-valid}(v)$ captures validity of a command sent by a client. We assume that each party has access to a small trusted hardware (described in Section 2.1). In addition, for communication efficiency, some of the messages are sent by the parties through an expander graph. We describe the properties needed from the expander graph in Section 2.2. We measure communication complexity as the number of *words* that all honest parties send and receive; each word is $O(\kappa)$ bits long where κ is a security parameter. We also assume all the messages sent by parties are signed using a threshold signature scheme (interface described in Section 2.3). We assume the use of PKI to validate signatures.

2.1 Small Trusted Hardware

In this section, we introduce the abstraction of a *small trusted hardware* enforcing non-equivocation with $O(1)$ storage. Such hardware units have been considered in prior works such as A2M [17], TrInc [32], etc. While the exact interface for the trusted hardware in these works differ, their capabilities are similar and supported by existing hardware modules

such as Trusted Platform Modules (TPMs), YubiKey [49, 1], and hardware security modules (HSMs) [46]. Without loss of generality, we assume the existence of a functionality similar to that of A2M [17]; our solutions apply to all of these small trusted hardware units.

Hardware state and interfaces. The trusted hardware provides a party with a set of append-only logs (denoted log) that can only be modified by the party's trusted hardware component. The functionality is shown in Table 1. Each log within a single party's trusted component has its own identifier (denoted id) and includes a counter (denoted c_{id}) that starts from 0 and is incremented for each entry that is appended to the log. The trusted hardware guarantees that the party cannot modify the information stored in any position of the log. We use the notation $\langle \cdot \rangle_{K_{priv}}$ to denote that an attestation is signed using the private key of the trusted hardware component. To differentiate between a signature from a hardware device and a signature from the party holding it, we always refer to the former signature as an attestation. We refer to the trusted hardware of party p_i as HW_i .

■ **Table 1 Interfaces of the trusted hardware component.** (K_{pub}, K_{priv}) is public-private key pair associated with the hardware device, C is a monotonic counter representing the number of logs maintained by the hardware, and c_{id} is a monotonic counter representing the length of log indexed by id .

CREATELOG()	Increment C Initialize empty log with $id := C, c_{id} := 0$ return id
APPEND (id, c_{new}, x)	if $id \leq C$: if $c_{new} = \perp$: Increment $c_{id}, log[id][c_{id}] := x$ if $c_{new} > c_{id}$: $c_{id} := c_{new}, log[id][c_{id}] := x$ return LOOKUP(id, c_{id})
LOOKUP(id, s)	if $id \leq C$ and $s \leq c_{id}$: return $\langle \text{LOOKUP}, id, s, log[id][s] \rangle_{K_{priv}}$
END(id, z)	if $id \leq C$: return $\langle \text{END}, id, c_{id}, log[id][c_{id}], z \rangle_{K_{priv}}$
COUNTERS(z)	return $\langle \text{HEAD}, \bigcup_{id < C} \{(id, c_{id})\}, z \rangle_{K_{priv}}$

The hardware provides four interfaces. APPEND(id, c_{new}, x) appends the value x to the log identified by id . If $c_{new} = \perp$, the function increments the counter of the log and inserts x into the current position of the log. Otherwise, it appends to position c_{new} if c_{new} is strictly higher than the current log position. LOOKUP(id, s) and END(id, z) return an attestation of the log with identifier id for the value stored at position s and the last position, respectively. Finally, COUNTERS(z) returns the attested current counter values of all logs. The nonce z is used to ensure freshness of an attestation; we omit mentioning the nonce when it is not used. To simplify the description, we imagine that the hardware stores entire logs. In reality, the hardware need only store the end state of a log; a party can always store the attestations for different positions separately.

If party p_i calls APPEND(q, \perp, x), then it receives $\langle \text{LOOKUP}, q, s, x \rangle_{K_{priv}}$ in response, for the log position s at which x was appended. p_i can forward this response – or another copy, obtained by invoking LOOKUP(q, s) – to party p_j to prove that p_i added value x to its log q

at position s . Since the hardware allows only appending to the log, p_j can be assured that p_i can attest to no other value at position s of log q . The use of the $\text{END}(q, z)$ function is similar, with the addition that p_j passes a random nonce z to p_i that will be included in the attestation to prove that it is fresh.

2.2 Expander Graphs

Expander graphs are sparse graphs with a high degree of connectivity between groups of nodes. We refer to a node connected to a node p_i as its neighbor and denote the set of neighbors of p_i by $\rho(i)$. We describe the expander graph properties we need in this section and prove them in Appendix A.

► **Definition 3.** An (n, α, β) -expander graph, denoted $G_{n, \alpha, \beta}$, where $0 < \beta < 1$ and $\alpha < \beta$, is a graph with n vertices such that every set of αn vertices has at least βn unique neighbors.

► **Lemma 4.** There exists a d -regular graph $G_{n, \epsilon, (1-\epsilon)}$ for sufficiently large n and positive constants $0 < \epsilon < \frac{1}{2}$ and $c > 2$ such that:

1. For any set S of $(\frac{1}{2} + \epsilon)n$ nodes, there exists a set Q of more than $\frac{n}{2}$ nodes, $Q \subseteq S$, and every node in Q has at least $(\frac{1}{2} + \frac{\epsilon}{2})d$ neighbors in S .
2. For any partition of its nodes into blocks T and Q where $|T| = (\frac{1}{2} - 2\epsilon)n$ and $|Q| = (\frac{1}{2} + 2\epsilon)n$, there exists a set $T' \subseteq T$, $|T'| > (\frac{1}{2} - 3\epsilon)n$, such that each node in T' has at least $(\frac{1}{2} + \frac{\epsilon}{2})d$ neighbors in Q .
3. For any set S of $(\frac{1}{2} + 2\epsilon)n$ nodes, there exists a set Q of more than $(\frac{1}{2} + \epsilon)n$ nodes, $Q \subseteq S$, and every node in Q has at least $(\frac{1}{2} + \frac{\epsilon}{2})d$ neighbors in S .
4. For any set S of ϵn nodes, and any sets $\{S_i\}_{i \in S}$ where $S_i \subset \rho(i)$ and $|S_i| = (\frac{1}{2} + \frac{\epsilon}{2})d$, the set $U = \bigcup_{i \in S} S_i$ satisfies $|U| > \frac{n}{2}$.

2.3 Cryptographic Abstractions

Threshold signatures. In addition to (and separate from) the signatures generated by hardware modules, we make use of a k out of l threshold signature scheme [47] for $k = \frac{n}{2} + 1$ and $l = n$, i.e., $\frac{n}{2} + 1$ parties must participate in order to create a valid threshold signature. We use the following interface:

- $\text{threshold-sign}_i(m)$: produces signature share produced by p_i on message m .
- $\text{share-validate}(m, s_j, pk_j)$: validate signature share s_j produced by p_j on m .
- $\text{threshold-combine}(m, S)$: combine a set S of signature shares from distinct parties for message m to an $O(1)$ -sized signature where $|S| \geq k$ and $\text{share-validate}(m, s_j, pk_j) = \text{true}, \forall s_j \in S$.
- $\text{threshold-verify}(m, \sigma)$: returns true if σ was a result of computing $\text{threshold-combine}(m, S)$ where $|S| \geq k$ and $\text{share-validate}(m, s_j, pk_j) = \text{true}, \forall s_j \in S$.

3 $(\frac{n}{2} + 1)$ -Provable-Broadcast

In this section, we present a core broadcast primitive that will enable protocols to tolerate up to $(\frac{1}{2} - \epsilon)n$ Byzantine faults for any $0 < \epsilon < \frac{1}{2}$ when every party is equipped with a trusted hardware component as described in Section 2.1. In subsequent sections, we will show how $(\frac{n}{2} + 1)$ -Provable-Broadcast along with trusted hardware can be used to increase the fault tolerance of protocols to minority faults without worsening their communication complexity.

$(\frac{n}{2} + 1)$ -Provable-Broadcast. This primitive is a generalization of $(t+1)$ -provable broadcast introduced by Abraham et al. [5]. Informally, in this broadcast, a designated sender sends a message $m = (v, \sigma_{\text{in}})$ consisting of a value v and a proof σ_{in} to all parties. If the message satisfies a certain predicate denoted by the validation function $\text{validate}()$, parties deliver the message. Finally, the sender delivers a proof σ_{out} indicating that $\frac{n}{2} + 1$ parties have delivered the broadcasted message.

The primitive provides the following guarantees:

- **Integrity.** An honest party (acting as a participant) delivers at most one message m for a given broadcast instance id .
- **Validity.** If an honest party delivers a message m for instance id , then $\text{validate}(id, (m, \sigma_m)) = \text{true}$, where σ_m is the proof of validity for m .
- **Provability.** If a sender can produce two valid proofs σ_{out} and σ'_{out} s.t. they are valid proofs for the delivery of m and m' respectively in instance id , then $m = m'$, and there exist $n/2 + 1$ parties who cannot deliver a value m' such that $m' \neq m$ in instance id .
- **Termination.** If sender is honest, no honest party invokes $\text{abandon}(id)$ (meaning they immediately terminate their execution of this instance of $(\frac{n}{2} + 1)$ -Provable-Broadcast), messages among honest parties arrive, $\text{validate}(id, (m, \sigma_{\text{in}})) = \text{true}$ for all honest parties, then (i) eventually all honest parties deliver m , and (ii) the sender delivers m with a valid proof σ_{out} .

■ **Algorithm 1** $(\frac{n}{2} + 1)$ -PB-Initiate instance id (sender s).

```

1: procedure  $(\frac{n}{2} + 1)$ -PB-Initiate ( $id, (v, \sigma_{\text{in}})$ )
2:    $S := \{\}$ 
3:   send “ $id, send, (v, \sigma_{\text{in}})$ ” to all parties
4:   while  $|S| \leq \frac{n}{2}$ 
5:     upon receiving “ $id, vote, \xi_j$ ” from  $p_j$  for the first time do
6:       if  $\text{share-validate}(v, \xi_j, pk_j) = \text{true}$ 
7:          $S := S \cup \{\xi_j\}$ 
8:    $qc := \text{threshold-combine}(S)$ 
9:    $\sigma_{\text{out}} \cdot id := id, \sigma_{\text{out}} \cdot val := v, \sigma_{\text{out}} \cdot qc_{\sigma_{\text{in}}} := \sigma_{\text{in}} \cdot qc, \sigma_{\text{out}} \cdot qc := qc$ 
10:  deliver  $\sigma_{\text{out}}$ 

```

The requirements described above have minor differences from those in Abraham et al. [5]. In particular, we modify the threshold from $t+1$ to $\frac{n}{2} + 1$ and require the provability property to have $\frac{n}{2} + 1$ parties to not be able to deliver a different message.

Our goal is to tolerate $(\frac{1}{2} - \epsilon)n$ Byzantine parties with linear communication complexity and ensure the size of σ_{out} to be $O(1)$. The $O(1)$ -sized proof allows us to use the primitive in a cascading manner while still maintaining linear communication complexity. To achieve these guarantees, we make use of two components: trusted hardware modules and expander graphs. Each party has access to a trusted hardware module as described in Section 2.1. Parties are connected in a d -regular expander graph $G_{n, \epsilon, (1-\epsilon)n}$ for a constant d , $0 < \epsilon < \frac{1}{2}$, and $c > 2$ that satisfies the properties in Lemma 4; the expander graph is used to communicate messages with constant communication complexity per party with its neighbors. We denote the neighbors of party p_i in the expander graph by $\rho(i)$.

Intuition. In the presence of a trusted hardware, any party receiving a valid message from the sender can attest to this message using the $\text{APPEND}()$ call to their trusted hardware in a specified log and sequence number. Sending this attestation back to the sender guarantees

both provability as well as termination against a corruption threshold of $< 1/2$. For provability, if delivery for every honest party requires attesting at a specific position in a log as a proof, then receiving $\frac{n}{2} + 1$ attestations from a set of parties P is a sufficient proof to state that parties in P cannot deliver a different message. For termination, if the sender is honest and eventually the sender's messages arrives at honest parties, then all honest parties will attest to this message in the correct log and sequence number and deliver m ; the attestations sent back to the sender will allow it to deliver m with a proof σ_{out} consisting of all the attestations it received. However, the proof σ_{out} is not $O(1)$ words. Observe that the attestations from the small trusted hardware from a linear number of parties provide us with $O(n)$ signatures. Thus, the challenge is to ensure that the proof σ_{out} remains $O(1)$ without relying on the hardware to generate threshold signatures.

■ **Algorithm 2** $(\frac{n}{2} + 1)$ -PB-Respond instance id (party p_i).

```

1: procedure  $(\frac{n}{2} + 1)$ -PB-Respond ( $id$ , validate, validateNeighbor)
2:    $stop := \text{false}$ 
3:   upon receiving " $id, send, (v, \sigma_{\text{in}})$ " from  $s$  do
4:      $(\sigma_i, \text{valid}) := \text{validate}(id, (v, \sigma_{\text{in}}))$ 
5:     if valid:
6:        $(\text{att}_{\log Id}, \text{att}_{\text{counters}}) := \text{createAttestations}(id, (v, \sigma_{\text{in}}))$ 
7:       send " $id, send, ((\text{att}_{\log Id}, \text{att}_{\text{counters}}), \sigma_i)$ " to parties in  $\rho(i)$ 
8:       wait for  $id, send, ((\text{att}_{\log Id, j}, \text{att}_{\text{counters}, j}), \sigma_j)$ 
         from  $(\frac{1}{2} + \frac{\epsilon}{2})d$  parties  $p_j$  in  $\rho(i)$ 
         s.t.  $\text{validateNeighbor}(id, p_j, (v, \sigma_{\text{in}}),$ 
            $(\text{att}_{\log Id, j}, \text{att}_{\text{counters}, j}), \sigma_j) = \text{true}$ 
9:        $\xi_i := \text{threshold-sign}_i(id, (v, \sigma_{\text{in}}.qc))$ 
10:      send " $id, vote, \xi_i$ " to  $s$ 
11:       $stop := \text{true}$ 
12:   upon abandon( $id$ ) do
13:      $stop := \text{true}$ 
14:   procedure createAttestations( $id, (v, \sigma_{\text{in}})$ )
15:      $\log Id := \text{log}(id)$ ,  $\text{seqNo} := \text{seq}(id)$  ▷ parse  $id$ 
16:      $\text{att}_{\log Id} := \text{APPEND}(\log Id, \text{seqNo}, (v, \sigma_{\text{in}}))$ 
17:     deliver  $((v, \sigma_{\text{in}}), (\text{att}_{\log Id}))$ 
18:     return  $(\text{att}_{\log Id}, \text{COUNTERS}())$ 

```

Our key idea is to verify the existence of $\frac{n}{2} + 1$ attestations by spreading this work evenly among the parties. We aim for two seemingly opposing goals: On the one hand, each party needs to check just a constant number of attestations to be locally satisfied. On the other hand, if a majority of parties say they are locally satisfied then, even if t of them are lying then it is still the case that there were $\frac{n}{2} + 1$ attestations. We obtain this through the magic of expander graphs. Every party communicates their attestation with their neighbors in the network. The expansion properties of the graph ensure that receiving correct information from a small fraction of honest parties, specifically ϵn , suffices to learn the state about a majority of the parties in the network. In particular, on receiving some *vote* messages (containing a threshold signature share) from $\frac{n}{2} + 1$ parties (see lines 4–7 in Algorithm 1), out of which at least ϵn parties are honest, the sender can learn that a majority of parties (not necessarily honest) have attested to a message m in the log and sequence number corresponding to the instance, and thus cannot deliver a different message with a valid attestation. To ensure that the proof σ_{out} is $O(1)$ words, the sender can simply combine the threshold signature shares sent in the *vote* messages of each of the $\frac{n}{2} + 1$ parties that vote.

Algorithms 1 and 2 present the pseudocode for $(\frac{n}{2} + 1)$ -Provable-Broadcast. We assume a setup phase during which each party creates the necessary logs for the protocol using the CREATELOG interface. Further, we assume, for an instance of $(\frac{n}{2} + 1)$ -Provable-Broadcast, that every party appends to, and expects attestations from, the log in the trusted hardware module of each node with the same *logId* and in the same position, *seqNo*, within the log.

Protocol. Each instance of this protocol is identified by an *id* and a designated sender *s*. The sender receives two inputs (v, σ_{in}) ; *v* is the value to be sent and σ_{in} is a proof to be validated by other parties using the `validate()` function. The sender sends the message “*id, send, (v, σ_{in})*” to all parties (Algorithm 1 line 3).

On receiving the message from the sender, p_i invokes `validate(id, (v, σ_{in}))` (Algorithm 2 line 4). The `validate()` function is used to check that the sender’s proposal is valid and that it satisfies any predicates on p_i ’s state as necessary for the higher level protocol. Thus, the interface allows each party to optionally provide some additional data/state that can be used for validation. If `validate()` is successful (`valid = true`), it returns a proof, σ_i , as proof that p_i can provide to other parties to prove that its call to `validate(id, (v, σ_{in}))` returned true. Upon successful validation, p_i then delivers the sender’s proposal by appending it to the log in its trusted hardware component using the `createAttestations()` method (Algorithm 2 lines 6, 14-18). In this method, p_i determines the log *logId* and the sequence *seqNo* in the log to be used using the `log(id)` and `seq(id)` functions. p_i then appends (v, σ_{in}) to log *logId* at sequence number *seqNo* in its trusted hardware component. It sends the attestation (along with a COUNTERS attestation, for reasons explained in Section 4) to all its neighbors $\rho(i)$ in the expander graph, as proof that it has delivered (v, σ_{in}) . On receiving messages from a majority of its neighbors (specifically $(\frac{1}{2} + \frac{\epsilon}{2})d$ neighbors) that satisfy `validateNeighbor()` (Algorithm 2 line 8), a party sends a *vote* message with `threshold-signi((v, σ_{in}))` to the sender (line 10). The `validateNeighbor()` function allows an invoking party to perform validation on the messages sent by their neighbors as proof of delivery; in the above instance, we can assume that it only validates that the attestation `attlogId,j` is correct, i.e. that it is from the log *logId*, signed by the sender’s trusted hardware component, and that the value was appended in the correct sequence number. In Section 4, we show how the proof output from `validate()` as well as the `attcounters` attestation are used.

On collecting `threshold-signi((v, σ_{in}))` from a majority of replicas, the sender combines the signature shares to generate σ_{out} and delivers σ_{out} (Algorithm 1 lines 8-10).

Here are the key ideas that ensure provability and termination. We present detailed proofs in Appendix B.

1. **Any set of ϵn nodes, each of which receives an attestation from at least $(\frac{1}{2} + \frac{\epsilon}{2})d$ of its neighbors, collectively receives attestations from at least $\frac{n}{2} + 1$ unique parties.** This property has been shown in Lemma 4 and it guarantees provability since if a sender receives *vote* messages from a majority of parties, at least ϵn of them are honest and they will ensure that at least $\frac{n}{2} + 1$ parties have attested to (v, σ_{in}) in the correct log and sequence number. Thus, they cannot deliver a message other than *v* with a valid proof of attestation. Also, by the same argument, another value $v' \neq v$ cannot receive a sufficient number of attestations, causing another set of ϵn honest parties to send a *vote* message for v' ; this is because at least $\frac{n}{2} + 1$ parties need to attest to $(v', *)$, and two majority sets will intersect in at least one node.
2. **For any set of $(1/2 + \epsilon)n$ nodes *S*, at least $(\frac{n}{2} + 1)$ nodes in *S* each have at least $(\frac{1}{2} + \frac{\epsilon}{2})d$ neighbors in *S*.** This property has been shown in Lemma 4 and it guarantees termination since if an honest sender sends a valid (v, σ_{in}) to all $(1/2 + \epsilon)n$ honest parties, then at least $\frac{n}{2} + 1$ honest parties will send *vote* messages, sufficient to generate σ_{out} .

4 HotStuff-M: HotStuff with Minority Corruption

In this section, we present HotStuff-M, a version of the HotStuff [58] protocol that tolerates minority Byzantine corruption under partial synchrony assuming a minimal trusted hardware at each party. Similar to HotStuff, the protocol has linear communication complexity per view. For simplicity, we show the construction of a single-shot version of HotStuff, though the ideas directly extend to the state machine replication setting.

4.1 Overview of Basic HotStuff

We start with an overview of the Basic HotStuff protocol [58] tolerating $n = 3t + 1$. The protocol proceeds in a sequence of consecutive views where each view has a unique leader. A view consists of four phases: PROMOTE, KEY, LOCK and COMMIT, as first formalized in [5]. We use σ_{phase} to denote the threshold signature collected by the leader of a given view during the *phase* phase of that view. Each view of HotStuff progresses as follows:

- **Promote.** The leader proposes a PROMOTE message containing a proposal v along with the σ_{key} (explained in the next bullet point) from the highest view known to it (referred to as $\sigma_{highKey}$) and sends it to all parties. On receiving a PROMOTE message containing a value v in a view e and a $\sigma_{highKey}$ from the leader of view e , a party sends a vote for v if it is *safe* to vote based on a locking mechanism (explained later). It sends this vote, in the form of a threshold signature share, to the leader.
- **Key.** The leader collects $2t + 1$ votes to form a threshold signature σ_{key} in view e . The leader sends the σ_{key} for view e to all parties. On receiving a σ_{key} in view e containing message v , a party updates its highest σ_{key} to (v, e) and sends LOCK to the leader.
- **Lock.** The leader collects $2t + 1$ such votes to form a threshold signature σ_{lock} , and sends it to all parties. On receiving σ_{lock} in view e containing message v from the leader, a party locks on (v, e) and sends COMMIT message to the leader.
- **Commit.** The leader collects $2t + 1$ such votes to form a threshold signature σ_{commit} and sends it to all parties. On receiving σ_{commit} from the leader, parties commit v .

Once a party locks on a given value v , it only votes for the value v in subsequent views. The only scenario in which it votes for a value $v' \neq v$ is when it observes a $\sigma_{highKey}$ from a higher view in a PROMOTE message. At the end of a view, every party sends its highest σ_{key} to the leader of the next view. The next view's leader collects $2t + 1$ such values, picking the highest σ_{key} as $\sigma_{highKey}$. The safety and liveness of HotStuff follow from the following:

Uniqueness within a view. Since parties only vote once in each phase, a σ_{commit} can be formed for only one value.

Safety and liveness across views. Safety across views is ensured using locks and the voting rule for a PROMOTE message. Whenever a party commits a value, at least $2t + 1$ other replicas are locked on the value in the view. A party only votes for the value it is locked on. The only scenario in which it votes for a conflicting value v' is if the leader includes a σ_{key} for v' from a higher view in a PROMOTE message. This indicates that at least $2t + 1$ replicas are not locked on v in a higher view, and hence it should be safe to vote for it. The latter constraint of voting for v' is not necessary for safety, but only for liveness of the protocol.

■ **Table 2 Validation functions passed to provable broadcast in different phases of view e .** We assume end attestations att_{lock} (containing σ_{lock}) is invoked during $\text{validate}()$ call as needed. Also note that $(\text{att}_{logId,i}, \text{att}_{counters,i})$ are sent by every party to their neighbor as a part of provable broadcast and generated in the invocation of $\text{createAttestations}(id, (v, \sigma_{in}))$.

Phase	$\text{validate}(id, (v, \sigma_{in}))$	$\text{validateNeighbor}(id, (v, \sigma_{in}), (\text{att}_{logId,j}, \text{att}_{counters,j}), \sigma_j)$
PROMOTE	cond: $\text{ext-valid}(v) = \text{true}$, $\sigma_{in}.val = \sigma_{lock}.val$ or $\text{view}(\sigma_{in}) > \text{view}(\sigma_{lock})$ proof: $\sigma_i := \text{att}_{lock}$	$(\sigma_{in}.val = \sigma_{lock}.val$ or $\text{view}(\sigma_{in}) > \text{view}(\sigma_{lock}))$, and LOCK log in $\text{att}_{counters,j}$ is at $\text{view}(\sigma_{lock})$, and PROGRESS log in $\text{att}_{counters,j}$ is at $e - 1$, and $\text{att}_{logId,j}$ is a valid attestation from HW_j for value v in the PROMOTE log and sequence number e
KEY	cond: $\text{view}(\sigma_{in}) = e$ and $\text{phase}(\sigma_{in}) = \text{PROMOTE}$ proof: $\sigma_i := \perp$	PROGRESS log in $\text{att}_{counters,j}$ is at $e - 1$, and KEY log in $\text{att}_{counters,j}$ is at e , and $\text{att}_{logId,j}$ is a valid attestation from HW_j for value v in the KEY log and sequence number e
LOCK	cond: $\text{view}(\sigma_{in}) = e$ and $\text{phase}(\sigma_{in}) = \text{KEY}$ proof: $\sigma_i := \perp$	PROGRESS log in $\text{att}_{counters,j}$ is at $e - 1$, and LOCK log in $\text{att}_{counters,j}$ is at e , and $\text{att}_{logId,j}$ is a valid attestation from HW_j for value v in the LOCK log and sequence number e

4.2 HotStuff-M: Towards Minority Corruption

The arguments for safety and liveness of HotStuff crucially rely on having fewer than one-third Byzantine faults. Otherwise, Byzantine parties could create multiple σ_{key} , σ_{lock} , and σ_{commit} by partitioning the honest parties. Similarly, across views, Byzantine parties could send an incorrect (stale) σ_{key} to the leader, as well as vote for a message in the PROMOTE phase without respecting the locking condition, leading to both safety and liveness concerns.

Our goal is to increase the corruption threshold from one-third to a minority while still retaining the linear communication complexity. The trusted hardware provides a non-equivocation guarantee, i.e., it ensures that once a value v has been appended to a position in a specified log, no other value can be appended at that position in that log. Moreover, the hardware provides an attestation, i.e., verifiable proof of the existence of value v at that position of the specified log. However, a party can still send a stale attestation to another party. For instance, during a view-change, a party can send an attestation to a key from an old view, possibly leading to a liveness violation. It can also potentially participate in a previous view even after quitting the current view. Similarly, a party can potentially append conflicting information at two different positions of the log and provide attestations to these different positions to different parties.

A potential way to fix the above concerns is to always send an attestation of all positions in the log whenever sending a message. The receiving party can validate that the log has been correctly constructed, e.g., absence of conflicting information and absence of a designated message indicating that the party quit a given view on the log. However, this solution makes the communication complexity proportional to the number of views for each message.

Our approach and protocol. Our approach uses multiple logs in the trusted hardware, one for each phase that consists of an instance of $(\frac{n}{2} + 1)$ -Provable-Broadcast, as well as a log to keep track of the view a party is in. For each log, the data appended to position j corresponds to the message sent by the party in view j . Thus, if a party votes for a value v in view e in the KEY phase of the protocol, it calls $\text{APPEND}(\text{KEY}, e, (v, *))$ to the KEY log at position e ($*$ denotes some additional information). However, a disadvantage of using multiple logs is the absence of relative ordering between them. This allows a Byzantine adversary to participate in a previous view by showing a stale state of a log or send a stale

■ **Algorithm 3** HotStuff-M: HotStuff with Minority Corruption (for party p_i).

```

1: for  $e := 1, 2, 3, \dots$  do
2:   as a leader ▷ NEW-VIEW phase
3:     wait for a set  $M$  of  $\geq \frac{n}{2} + 1$  NEW-VIEW messages s.t. the attestations on PROGRESS
      and KEY logs are valid, sequence numbers in  $att_{progress}$  and  $att_{highQC}$  match
      those in the COUNTERS attestation for the respective logs, and counter value of
      PROGRESS log in the COUNTERS attestation is  $e - 1$ 
4:     For each  $m \in M$ , let  $\sigma_{highKey}^m$  denote the highest key QC from party  $p_m$ 
5:      $\sigma_{highKey} := (\arg \max_{m \in M} \{\text{view}(\sigma_{highKey}^m)\})$  ▷  $\text{view}(\sigma_{highKey}^m)$  is the view in which
       $\sigma_{highKey}^m$  was formed
6:     if  $\sigma_{highKey} = \perp$  then proposal := client's command else proposal :=  $\sigma_{highKey}.val$ 
7:   as a party ▷ NEW-VIEW phase
8:     go to this line if no progress happens during the “wait” step in any phase
9:      $att_{progress} := \langle \text{LOOKUP}, \text{PROGRESS}, e, e \rangle := \text{APPEND}(\text{PROGRESS}, e, e)$ 
10:     $att_{highQC} := \langle \text{END}, \text{KEY}, seqNoHighQC, highKeyQC \rangle := \text{END}(\text{KEY})$ 
11:    send “ $((e, \text{NEW-VIEW}), send, (att_{progress}, att_{highQC}, \text{COUNTERS}()))$ ” to view  $e + 1$ 
    leader
12:   as a leader
13:      $\sigma_{key} := (\frac{n}{2} + 1)\text{-PB-Initiate}((e, \text{PROMOTE}), (\text{proposal}, \sigma_{highKey}))$  ▷ PROMOTE
14:      $\sigma_{lock} := (\frac{n}{2} + 1)\text{-PB-Initiate}((e, \text{KEY}), (\text{proposal}, \sigma_{key}))$  ▷ KEY phase
15:      $\sigma_{commit} := (\frac{n}{2} + 1)\text{-PB-Initiate}((e, \text{LOCK}), (\text{proposal}, \sigma_{lock}))$  ▷ LOCK phase
16:     send “ $((e, \text{COMMIT}), send, \sigma_{commit})$ ” ▷ COMMIT phase
17:   as a party
18:      $(\frac{n}{2} + 1)\text{-PB-Respond}((e, \text{PROMOTE}), \text{validate}(), \text{validateNeighbor}())$  ▷ PROMOTE
19:      $(\frac{n}{2} + 1)\text{-PB-Respond}((e, \text{KEY}), \text{validate}(), \text{validateNeighbor}())$  ▷ KEY phase
20:      $(\frac{n}{2} + 1)\text{-PB-Respond}((e, \text{LOCK}), \text{validate}(), \text{validateNeighbor}())$  ▷ LOCK phase
21:     wait for “ $((e, \text{COMMIT}), send, \sigma_{commit})$ ” from view  $e$  leader ▷ COMMIT phase
22:     if  $\sigma_{commit}$  is a signature from view  $e$  from COMMIT phase then commit  $\sigma_{commit}.val$ 

```

σ_{lock} while voting in the PROMOTE phase. We leverage the COUNTERS() call on the hardware to address this concern; it provides the end state of all of the logs at once, thus allowing the receiving party to validate the freshness of the state. Although the functionality provided by the COUNTERS attestation can be achieved using a nonce with the same communication complexity, we use COUNTERS for the simplicity of the description. At the end of this section, we discuss the intuition for how the COUNTERS attestation can be replaced with the use of nonces.

We present our protocol in Algorithm 3 and Table 2. The parties proceed in a sequence of views. We assume that the parties know the leader in a given view. Let e denote the current view of the protocol. At the end of the previous view, each party invokes an APPEND(PROGRESS, $e - 1$, $e - 1$) (line 9) to obtain attestation $att_{progress}$. In addition, it obtains an end attestation att_{highQC} for its $keyQC$ (line 10). The party sends this information together with the COUNTERS() attestation to the leader in a NEW-VIEW message (line 11).

The leader of view e waits for a valid NEW-VIEW message from a majority of parties. Here, the message is considered valid if (i) the attestations are valid (i.e., signed by the trusted hardware component of the sending party), (ii) the sending party has quit view $e - 1$, i.e., the counter value of the PROGRESS log is equal to $e - 1$, and (iii) the sequence number on att_{highQC} and $att_{progress}$ matches the ones in the COUNTERS attestation (line 3). Thus,

even if the sending party is Byzantine, the $\text{att}_{\text{high}QC}$ is fresh and the party can no longer act in the previous view (due to the current counter value of its PROGRESS log and the conditions in $\text{validateNeighbor}()$). The leader picks the $\text{key}QC$ from the highest view as σ_{highKey} and proposes the value in the certificate. Otherwise, it proposes any client command.

Our modular construction allows us to present the next three phases PROMOTE, KEY, and LOCK as invocations of $(\frac{n}{2} + 1)$ -Provable-Broadcast (lines 13-15 and 18-20). As described in the previous section, if the leader successfully receives a σ_{key} (respectively σ_{lock} and σ_{commit}), it guarantees that $\geq \frac{n}{2} + 1$ parties have attested to the proposed value in their PROMOTE log (respectively KEY log and LOCK log) in the position corresponding to view e . However, in each provable broadcast phase, a party should vote for the leader’s proposal only if it is safe to do so depending on the party’s state. We use the $\text{validate}()$ and $\text{validateNeighbor}()$ interface to specify these constraints (described in Table 2). Recall that the former is used to validate the leader’s proposal and provide a proof that the leader’s proposal satisfies $\text{validate}()$ for this party, while the latter is used by a neighbor in the expander graph to verify correct behavior.

In the PROMOTE phase, a party votes for a leader’s message only if it is locked on the same value as the proposal or if σ_{highKey} in the leader’s proposal is from a higher view than the party’s lock, σ_{lock} . The party sends an attestation to its lock as proof for the neighbor to verify. The expander graph neighbor verifies the correctness of the computation in addition to ensuring that the attestations received are valid and fresh (using the counter values in $\text{att}_{\text{counters},j}$ and comparing them to the sequence numbers in the other attestations). In the KEY, LOCK, and COMMIT phases, the parties check if σ_{key} , σ_{lock} , and σ_{commit} , respectively, are from the same view and the proofs were formed in the correct phases. The expander graph neighbors verify validity of attestations and freshness (to ensure they have not quit the view). Finally, the leader sends a COMMIT message along with σ_{commit} as proof of commit. Each of the parties can then commit $\sigma_{\text{commit}.val}$.

Due to space constraints, we present formal proofs in Appendix C and a view synchronization protocol in the full version of our paper [57]; the protocol is a generalization of the expected linear communication complexity protocol of [39] withstanding minority corruption.

Communication complexity. From Theorem 14, an instance of provable broadcast in each of the three phases (PROMOTE, KEY, and LOCK) incurs linear communication complexity. To change views, each party sends a constant number of attestations to the leader in a single message. In both the NEW-VIEW phase and the COMMIT phase, the leader sends a single, constant-sized message to all the parties. Therefore, the HotStuff with Minority Corruption protocol incurs $O(n)$ communication complexity per view.

Replacing the Counters attestation. We now present intuition for how nonces can be used to replace the COUNTERS attestation. We use the COUNTERS attestation to:

1. Prevent parties from presenting the stale state of a log to other parties
2. Force parties to quit a view before sending messages in a subsequent view

We address concern 1 using the COUNTERS attestation by requiring a party to append a value to position e in their PROGRESS log before presenting the state of their log in view $e + 1$. The COUNTERS attestation shows that a party did in fact append the value to their PROGRESS log in position e prior to presenting the state of their logs in view $e + 1$. The following handshake between a proving party and a verifying party can replace the use of the COUNTERS attestation for this scenario:

1. To send a message proving the state of their logs in view $e + 1$, a party sends an END attestation from their PROGRESS log to the verifying party, proving that they have quit view e
2. Upon receiving a valid attestation showing that the proving party quit view e , the verifying party sends the proving party a random nonce
3. The proving party uses the nonce for the END attestations it gathers to send the current state of its logs

We address concern 2 using the COUNTERS attestation by requiring a party to send a COUNTERS attestation showing that they have appended a value to their PROGRESS log in position e with any view $e + 1$ messages that they send. A similar handshake to that described above can be used to replace the COUNTERS attestation for this scenario.

5 Related Work

Trusted hardware and consensus. Trusted hardware can be classified into two categories depending on the computations it can provide. The more powerful class is capable of running arbitrary specified code in a trusted execution environment (TEE). The protected execution state is encrypted by the trusted module and written to a specified memory range that only the trusted module can access while the code is running (e.g. Intel SGX [19], Flicker [36], Aegis [48], XOM [33], and Bastion [12]). They have been used to provide confidentiality [19, 48, 33, 12, 45] as well as to improve resilience and performance in the context of consensus protocols [7, 25, 45, 9, 2, 34]. However, the trusted computing bases of such platforms tend to grow as they increase in their generality, up to an including extensive libraries and OS components (e.g., [50]). Consequently, these platforms can present a large attack surface, giving way to attacks from outside the TEE (e.g., [14]).

Our work focuses on using small trusted hardware with a fixed, limited functionality (e.g., YubiKeys [49]). There have been several works using such a hardware to improve the performance of BFT protocols [27, 53, 54, 17, 32, 41, 3]. Notable works include A2M [17] and TrInc [32]. Chun et al. [17] show how, by introducing append-only (A2M) logs in the trusted hardware component of all processors in a network, the fault tolerance of BFT protocols can be increased to minority faults. They show an implementation of PBFT that withstands minority faults using A2M logs. However, simply applying their approach to a BFT protocol can increase the communication complexity by at least a factor of n due to the communication pattern of the protocol. In TrInc [32], Levin et al. show how A2M logs can be implemented with a small trusted monotonic counter, a key, and a small amount of trusted storage.

Expander graphs and consensus. Expander graphs have been used in the context of consensus protocols in works in the past [30, 29, 37]. Chlebus et al. [16] present an algorithm that solves consensus in the crash fault setting such that the per-process communication complexity is polylogarithmic in the number of processors. The protocol for leader election by King et al. withstands a one-third Byzantine adversary in synchrony using expanders to achieve polylogarithmic per-process communication complexity [30]. They extend this work to obtain $o(n^2)$ total bits of communication against an adaptive adversary. Recently, Momose and Ren [37] used expanders to solve Byzantine agreement against a minority corruption. Their work uses expanders to detect equivocation under synchrony. One could also consider the use of random sampling to be a randomized method to obtain the properties of expanders [23, 43, 24]. The way we use expander graphs in our work differs from their use

in previous works, as we separate the messages that must go through the hardware and those that do not. The separation enables better communication complexity for messages sent from the hardware through the use of expander graphs and better communication complexity for other messages through the use of cryptography.

Non-equivocation. The past two decades have seen various works on non-equivocation [17, 18, 44, 6, 35]. Clement et al. [18] define equivocation and show that non-equivocation alone is not sufficient to increase the threshold of Byzantine parties in a network when trying to reach agreement; doing so requires transferable authentication. Ruffing et al. [44] present non-equivocation contracts, which reveal the Bitcoin credentials of an equivocating party in order to penalize equivocation. Backes et al. [6] show how to use non-equivocation to improve the resilience of asynchronous MPC to match that of synchronous MPC, which tolerates minority corruption. [8] and [18] both explore the gap between omission failures and the non-equivocation property achieved by the use of trusted hardware. Our work expands on these works by showing how non-equivocation implemented by the use of trusted hardware can be combined with expander graph techniques to increase the fault tolerance of BFT protocols without increasing the communication complexity.

References

- 1 Trusted computing group. URL: <https://trustedcomputinggroup.org/>.
- 2 Hyperledger sawtooth, 2019. URL: <https://sawtooth.hyperledger.org/>.
- 3 Ittai Abraham, Marcos K Aguilera, and Dahlia Malkhi. Fast asynchronous consensus with optimal resilience. In *International Symposium on Distributed Computing*, pages 4–19. Springer, 2010.
- 4 Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync hotstuff: Simple and practical synchronous state machine replication. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 106–118. IEEE, 2020.
- 5 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.
- 6 Michael Backes, Fabian Bendun, Ashish Choudhury, and Aniket Kate. Asynchronous mpc with a strict honest majority using non-equivocation. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, pages 10–19, 2014.
- 7 Johannes Behl, Tobias Distler, and Rüdiger Kapitza. Hybrids on steroids: Sgx-based high performance bft. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 222–237, 2017.
- 8 Naama Ben-David, Benjamin Y Chan, and Elaine Shi. Revisiting the power of non-equivocation in distributed protocols. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, pages 450–459, 2022.
- 9 Marcus Brandenburger, Christian Cachin, Rüdiger Kapitza, and Alessandro Sorniotti. Blockchain and trusted computing: Problems, pitfalls, and a solution for hyperledger fabric. *arXiv preprint*, 2018. [arXiv:1805.08541](https://arxiv.org/abs/1805.08541).
- 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 *OSDI*, volume 99, pages 173–186, 1999.
- 12 David Champagne and Ruby B Lee. Scalable architectural support for trusted software. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12. IEEE, 2010.

- 13 Benjamin Y Chan and Elaine Shi. Streamlet: Textbook streamlined blockchains. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, pages 1–11, 2020.
- 14 Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call API is a bad untrusted RPC interface. In *18th International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2013.
- 15 Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. SGXPECTRE: Stealing Intel secrets from SGX enclaves via speculative execution. In *IEEE European Symposium on Security and Privacy*, June 2019.
- 16 Bogdan S Chlebus, Dariusz R Kowalski, and Michal Strojnowski. Fast scalable deterministic consensus for crash failures. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 111–120, 2009.
- 17 Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. *ACM SIGOPS Operating Systems Review*, 41(6):189–204, 2007.
- 18 Allen Clement, Flavio Junqueira, Aniket Kate, and Rodrigo Rodrigues. On the (limited) power of non-equivocation. In *Proceedings of the 2012 ACM symposium on Principles of distributed computing*, pages 301–308, 2012.
- 19 Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptol. ePrint Arch.*, 2016(86):1–118, 2016.
- 20 Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.
- 21 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988. doi:10.1145/42282.42283.
- 22 Michael J Fischer, Nancy A Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1(1):26–39, 1986.
- 23 Seth Gilbert and Dariusz R Kowalski. Distributed agreement with optimal communication complexity. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 965–977. SIAM, 2010.
- 24 Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Dragos-Adrian Seredinschi, and Yann Vonlanthen. Scalable byzantine reliable broadcast (extended version). *arXiv preprint*, 2019. arXiv:1908.01738.
- 25 Mike Hearn. Corda: A distributed ledger. *Corda Technical White Paper*, 2016, 2016.
- 26 Shlomo Hoory, Nathan Linial, and Avi Wigderson. Expander graphs and their applications. *Bulletin of the American Mathematical Society*, 43(4):439–561, 2006.
- 27 Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. Cheapbft: Resource-efficient byzantine fault tolerance. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 295–308, 2012.
- 28 Jonathan Katz and Chiu-Yuen Koo. On expected constant-round protocols for byzantine agreement. *Journal of Computer and System Sciences*, 75(2):91–112, 2009.
- 29 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):1–24, 2011.
- 30 Valerie King, Jared Saia, Vishal Sanwalani, and Erik Vee. Scalable leader election. In *17th ACM-SIAM Symposium on Discrete Algorithms*, pages 990–999, 2006.
- 31 Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. In *Concurrency: the Works of Leslie Lamport*, pages 203–226. ACM, 2019.
- 32 Dave Levin, John R Douceur, Jacob R Lorch, and Thomas Moscibroda. Trinc: Small trusted hardware for large distributed systems. In *NSDI*, volume 9, pages 1–14, 2009.
- 33 David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. *Acm Sigplan Notices*, 35(11):168–177, 2000.

- 34 Jian Liu, Wenting Li, Ghassan O Karame, and N Asokan. Scalable byzantine consensus via hardware-assisted secret sharing. *IEEE Transactions on Computers*, 68(1):139–151, 2018.
- 35 Mads Frederik Madsen and Søren Debois. On the subject of non-equivocation: Defining non-equivocation in synchronous agreement systems. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, pages 159–168, 2020.
- 36 Jonathan M McCune, Bryan J Parno, Adrian Perrig, Michael K Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for tcb minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 315–328, 2008.
- 37 Atsuki Momose and Ling Ren. Optimal communication complexity of byzantine consensus under honest majority. *arXiv preprint*, 2020. [arXiv:2007.13175](https://arxiv.org/abs/2007.13175).
- 38 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, Manubot, 2019.
- 39 Oded Naor and Idit Keidar. Expected linear round synchronization: The missing link for linear byzantine smr. *arXiv preprint*, 2020. [arXiv:2002.07539](https://arxiv.org/abs/2002.07539).
- 40 Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. A survey of published attacks on intel sgx. *arXiv preprint*, 2020. [arXiv:2006.13598](https://arxiv.org/abs/2006.13598).
- 41 Vincent Rahli, Francisco Rocha, Marcus Völp, and Paulo Esteves-Verissimo. Deconstructing minbft for security and verifiability.
- 42 Ling Ren, Kartik Nayak, Ittai Abraham, and Srinivas Devadas. Practical synchronous byzantine consensus. *arXiv preprint*, 2017. [arXiv:1704.02397](https://arxiv.org/abs/1704.02397).
- 43 Team Rocket. Snowflake to avalanche: A novel metastable consensus protocol family for cryptocurrencies. Available [online]. [Accessed: 4-12-2018], 2018.
- 44 Tim Ruffing, Aniket Kate, and Dominique Schröder. Liar, liar, coins on fire! penalizing equivocation by loss of bitcoins. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 219–230, 2015.
- 45 Mark Russinovich, Edward Ashton, Christine Avanesians, Miguel Castro, Amaury Chamayou, Sylvan Clebsch, Manuel Costa, Cédric Fournet, Matthew Kerner, Sid Krishna, et al. Ccf: A framework for building confidential verifiable replicated services. *Technical Report MSR-TR-201916*, 2019.
- 46 Jinho Seol, Seongwook Jin, Daewoo Lee, Jaehyuk Huh, and Seungryoul Maeng. A trusted iaas environment with hardware security module. *IEEE Transactions on Services Computing*, 9(3):343–356, 2015.
- 47 Victor Shoup. Practical threshold signatures. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 207–220. Springer, 2000.
- 48 G Edward Suh, Dwaine Clarke, Blaise Gassend, Marten Van Dijk, and Srinivas Devadas. Aegis: Architecture for tamper-evident and tamper-resistant processing. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 357–368, 2003.
- 49 Suresh Thiru, Shamalee Deshpande, and Stina Ehrensvar. Yubikey strong two factor authentication, January 2021. URL: <https://www.yubico.com/>.
- 50 Chia-Che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *USENIX Annual Technical Conference*, July 2017.
- 51 Salil P. Vadhan. *Pseudorandomness*, volume 7 of *Foundations and Trends in Theoretical Computer Science*. Now Publishers, Inc., 2012.
- 52 Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, August 2018. See also technical report Foreshadow-NG [56].
- 53 Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. Ebawa: Efficient byzantine agreement for wide-area networks. In *2010 IEEE 12th International Symposium on High Assurance Systems Engineering*, pages 10–19. IEEE, 2010.

- 54 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.
- 55 Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindshaedler, Haixu Tang, and Carl A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *ACM Conference on Computer and Communications Security*, October 2017.
- 56 Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. *Technical report*, 2018. See also USENIX Security paper Foreshadow [52].
- 57 Sravya Yandamuri, Ittai Abraham, Kartik Nayak, and Michael K Reiter. Communication-efficient bft protocols using small trusted hardware to tolerate minority corruption. *Cryptology ePrint Archive*, 2021.
- 58 Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT consensus in the lens of blockchain. *arXiv preprint*, 2018. [arXiv:1803.05069](https://arxiv.org/abs/1803.05069).

A Proofs for Expander Graph Lemmas

► **Lemma 5.** *For every constant $0 < \alpha < \beta < 1$ and sufficiently large n , there exists a d -regular graph that is an (n, α, β) -expander.*

Proof. For this proof, we will show a randomized way to construct a d -regular graph $G_{n,\alpha,\beta}$. Then, we will show that with high probability, it satisfies the lemma.

Let $\Gamma(V, G)$ refer to the set of neighbors of the vertices V in a graph G . Consider a random degree- d graph G constructed by taking the union of d random perfect matchings (assume that n is even; if n is odd, we can add a dummy node or assign a vertex to two neighbors). In order to ensure that each vertex has degree exactly d , one can construct each of the perfect matchings in the following way. Create the first perfect matching by taking one vertex at a time and matching it to a random unmatched vertex in the graph, repeating until all vertices have been matched. Repeat this process for d perfect matchings without replacement, so that for the k th perfect matching, each vertex is matched to a random vertex from the set of vertices that it hasn't been matched to in a previous perfect matching.

Consider a perfect matching P , a set of αn nodes, S , and a set of βn nodes, T . Now, consider the matching of the first set of αn nodes, S , in the first perfect matching (the perfect matching that results in a graph with degree 1). The probability that the first vertex that is matched is matched to a vertex in T is $\frac{\beta n}{n}$. Since $\alpha < \beta$, and we match without replacement, the probability of choosing a match in T for the i th node in S after all of the previous matchings of nodes in S have been to nodes in T can only be less than this quantity. Since it is possible that $S \subset T$, and nodes in S are matched to each other, we only multiply this quantity $\frac{\alpha n}{2}$ times. We are therefore able to obtain the following upper bound for the probability that in each perfect matching P , for any set of αn nodes S , and any set of βn nodes T , $\Gamma(S, P) \subseteq T$:

$$\Pr[\Gamma(S, P) \subseteq T] \leq \left(\frac{\beta n}{n}\right)^{\frac{\alpha n}{2}} = \beta^{\frac{\alpha n}{2}} \quad (1)$$

Using this, the probability that any set of αn vertices does not expand to more than βn other vertices, i.e. $|\Gamma(S, G)| \leq \beta n$ for any set S , is bounded above by:

$$\binom{n}{\alpha n} \binom{n}{\beta n} \beta^{\frac{\alpha n d}{2}} \quad (2)$$

$$\leq \left(\frac{e}{\alpha}\right)^{\alpha n} \left(\frac{e}{\beta}\right)^{\beta n} \beta^{\frac{\alpha n d}{2}} \quad (3)$$

$$\leq [e^{\alpha+\beta} \left(\left(\frac{1}{\alpha}\right)^{\alpha} \left(\frac{1}{\beta}\right)^{\beta}\right) \beta^{\frac{\alpha d}{2}}]^n \quad (4)$$

For a sufficiently large constant d , the above probability is exponentially small, which means that with high probability, a graph randomly chosen with the above procedure is an (n, α, β) -expander. Thus, $G_{n, \alpha, \beta}$ exists. \blacktriangleleft

► **Lemma 6.** *There exists an expander graph $G_{n, \epsilon, \beta}$ with degree d , $0 < \epsilon < \frac{1}{2}$, $\epsilon < \beta < 1$, such that for any set S of $(\frac{1}{2} + \epsilon)n$ nodes, there exists a set Q of more than $\frac{n}{2}$ nodes, $Q \subseteq S$, and every node in Q has at least $(\frac{1}{2} + \frac{\epsilon}{2})d$ neighbors in S .*

Proof. From Lemma 5, we know that a $G_{n, \epsilon, \beta}$ -expander exists. For the rest of the proof, we show that with high probability an expander graph with sufficiently high degree constructed using the randomized procedure outlined in the proof for Lemma 5 satisfies the lemma.

Consider a set T of $(\frac{1}{2} - \epsilon)n$ nodes in our expander graph G . Let S be the set of nodes in G that are not in T . If we show that with high probability, it is not the case that ϵn nodes in S have more than $(\frac{1}{2} - \frac{\epsilon}{2})d$ neighbors in T , then there must be a set of nodes Q of size greater than $\frac{n}{2}$ in S that satisfies the lemma. Therefore, using the same technique as that in the proof for Lemma 5, we first bound the probability that for a given set of nodes R of size ϵn in a perfect matching P , all nodes in R have neighbors in a set T of size $(\frac{1}{2} - \epsilon)n$, where R and T are pairwise disjoint.

$$\Pr[\Gamma(R, P) \subseteq T] \leq \left(\frac{(\frac{1}{2} - \epsilon)n}{n}\right)^{\epsilon n} = \left(\frac{1}{2} - \epsilon\right)^{\epsilon n} \quad (5)$$

Then the probability that there does not exist a set Q of more than $\frac{n}{2}$ nodes that satisfies the statement in the lemma is bounded by:

$$\binom{n}{\epsilon n} \binom{(1 - \epsilon)n}{(\frac{1}{2} - \epsilon)n} \left(\frac{1}{2} - \epsilon\right)^{\epsilon n d (\frac{1}{2} - \frac{\epsilon}{2})} \quad (6)$$

$$\leq \left[\left(\frac{e}{\epsilon}\right)^{\epsilon} \left(\frac{e(1 - \epsilon)}{(\frac{1}{2} - \epsilon)}\right)^{(\frac{1}{2} - \epsilon)} \left(\frac{1}{2} - \epsilon\right)^{\epsilon d (\frac{1}{2} - \frac{\epsilon}{2})}\right]^n \quad (7)$$

$$\leq \left[e \left(\frac{1}{\epsilon}\right)^{\epsilon} \left(\frac{1 - \epsilon}{\frac{1}{2} - \epsilon}\right)^{(\frac{1}{2} - \epsilon)} \left(\frac{1}{2} - \epsilon\right)^{\epsilon d (\frac{1}{2} - \frac{\epsilon}{2})}\right]^n \quad (8)$$

Again, for sufficiently large d , the above probability is exponentially small. Thus, with high probability the lemma holds. \blacktriangleleft

► **Lemma 7.** *There exists an expander graph $G_{n, \epsilon, \beta}$ with degree d , $0 < \epsilon < \frac{1}{2}$, $\epsilon < \beta < 1$ such that for any partition of its nodes into blocks T and Q where $|T| = (\frac{1}{2} - 2\epsilon)n$ and $|Q| = (\frac{1}{2} + 2\epsilon)n$, there exists a set $T' \subseteq T$, $|T'| > (\frac{1}{2} - 3\epsilon)n$, such that each node in T' has at least $(\frac{1}{2} + \frac{\epsilon}{2})d$ neighbors in Q .*

Proof. From Lemma 5, we know that a $G_{n,\epsilon,\beta}$ -expander exists. For the rest of the proof, we show that with high probability an expander graph with sufficiently high degree constructed using the randomized procedure outlined in the proof for Lemma 5 satisfies the lemma.

Consider a set T of $(\frac{1}{2} - 2\epsilon)n$ nodes in our expander graph G . Let R be a set of nodes in T of size ϵn . Using the same technique as that in the proof for Lemma 5, we first bound the probability that in a given perfect matching P , all nodes in R have neighbors in T .

$$\Pr[\Gamma(R, P) \subseteq T] \leq \left(\frac{(\frac{1}{2} - 2\epsilon)n}{n}\right)^{\frac{\epsilon n}{2}} = \left(\frac{1}{2} - 2\epsilon\right)^{\frac{\epsilon n}{2}} \quad (9)$$

Then the probability that ϵn nodes in T have more than $(\frac{1}{2} - \frac{\epsilon}{2})d$ neighbors in T is bounded by:

$$\binom{n}{(\frac{1}{2} - 2\epsilon)n} \binom{(\frac{1}{2} - 2\epsilon)n}{\epsilon n} \left(\frac{1}{2} - 2\epsilon\right)^{\frac{\epsilon n d}{2}(\frac{1}{2} - \frac{\epsilon}{2})} \quad (10)$$

$$\leq \left[\left(\frac{e}{\frac{1}{2} - 2\epsilon}\right)^{\frac{1}{2} - 2\epsilon} \left(\frac{e(\frac{1}{2} - 2\epsilon)}{\epsilon}\right)^\epsilon \left(\frac{1}{2} - 2\epsilon\right)^{\frac{\epsilon d}{2}(\frac{1}{2} - \frac{\epsilon}{2})}\right]^n \quad (11)$$

$$\leq \left[e\left(\frac{1}{\frac{1}{2} - 2\epsilon}\right)^{\frac{1}{2} - 2\epsilon} \left(\frac{\frac{1}{2} - 2\epsilon}{\epsilon}\right)^\epsilon \left(\frac{1}{2} - 2\epsilon\right)^{\frac{\epsilon d}{2}(\frac{1}{2} - \frac{\epsilon}{2})}\right]^n \quad (12)$$

Again, for sufficiently large d , the above probability is exponentially small. Thus, with high probability the lemma holds. \blacktriangleleft

► Lemma 8. *There exists an expander graph $G_{n,\epsilon,\beta}$ with degree d , $0 < \epsilon < \frac{1}{2}$, $\epsilon < \beta < 1$, such that for any set S of $(\frac{1}{2} + 2\epsilon)n$ nodes, there exists a set Q of more than $(\frac{1}{2} + \epsilon)n$ nodes, $Q \subseteq S$, and every node in Q has at least $(\frac{1}{2} + \frac{\epsilon}{2})d$ neighbors in S .*

Proof. This lemma follows directly from Lemma 6, as any graph that violates this property also violates the property in Lemma 6. \blacktriangleleft

The following lemma and theorem can be found in any resources on expander graphs, such as Hoory et al. [26].

► Lemma 9 (Expander Mixing Lemma). *Let $G = (V, E)$ be a d -regular graph and let $S, T \subseteq V$. Then,*

$$\left| |E(S, T)| - \frac{d|S||T|}{n} \right| \leq \lambda(G) \cdot d\sqrt{|S|(1 - |S|/n)|T|(1 - |T|/n)} \quad (13)$$

where $|E(S, T)|$ is the number of edges between the two sets (counting edges contained in the intersection of S and T twice) and $\lambda(G)$ is the second largest eigenvalue of the adjacency matrix of G .

► Theorem 10 (Theorem 4.12 of Vadhan [51], restated). *For any constant $d \in \mathbb{N}$, a random d -regular n -vertex graph satisfies $\lambda(G) \leq 2\sqrt{d-1}/d + O(1)$ with probability $1 - O(1)$ where $\lambda(G)$ is the second largest eigenvalue of the adjacency matrix of G and both $O(1)$ terms vanish as n approaches ∞ (and d is held constant).*

► Lemma 11. *For all sufficiently large integers n and positive constants ϵ and β such that $0 < \epsilon < \beta < 1$ there exists an d -regular expander $G_{n,\epsilon,\beta}$ such that for any set S of ϵn nodes and any set T of $(\frac{1}{2} + \frac{\epsilon}{c})n$ nodes, where $c > 2$, the number of edges with one vertex in S and one vertex in T is less than $(\frac{1}{2} + \frac{\epsilon}{2})\epsilon dn$.*

Proof. By Lemma 5, we know that a random d -regular expander $G_{n,\epsilon,\beta}$ exists for a sufficiently large constant d . By the Expander Mixing Lemma [26], we know that for any two sets of vertices S and T where $|S| = \epsilon n$ and $|T| = (\frac{1}{2} + \frac{\epsilon}{c})n$, the number of edges between the vertices in S and those in T , $E(S, T)$, in a d -regular expander graph G is upper bounded by:

$$E(S, T) \leq \lambda d \sqrt{\epsilon n (1 - \epsilon) \left(\frac{1}{2} + \frac{\epsilon}{c}\right) n \left(\frac{1}{2} - \frac{\epsilon}{c}\right)} + \frac{\epsilon d n}{2} + \frac{\epsilon^2 d n}{c} \quad (14)$$

In order to satisfy the lemma, we need:

$$E(S, T) \leq \lambda \sqrt{(\epsilon n - \epsilon^2 n) \left(\frac{1}{2} + \frac{\epsilon}{c}\right) \left(\frac{n}{2} - \frac{\epsilon n}{c}\right)} < \frac{\epsilon^2 n}{2} - \frac{\epsilon^2 n}{c} \quad (15)$$

Since G is a random d -regular expander graph, we can upper bound $\lambda(G)$ using [51, Theorem 4.12]:

$$\lambda \leq \frac{2\sqrt{d-1}}{d} + O(1) < \frac{\frac{\epsilon^2}{2} - \frac{\epsilon^2}{c}}{\sqrt{\frac{\epsilon}{4} - \frac{\epsilon^2}{4} - \frac{\epsilon^3}{c^2} + \frac{\epsilon^4}{c^2}}} \quad (16)$$

With some simplification, and as the $O(1)$ term goes to 0 as n goes to infinity, we get:

$$d > \frac{\epsilon}{\left(\frac{\epsilon^2}{2} - \frac{\epsilon^2}{c}\right)^2} \quad (17)$$

Which is satisfied by sufficiently large constant d . ◀

Proof of Lemma 4

Proof. By Lemma 11, we know that there exists an expander $G_{n,\epsilon,(1-\frac{\epsilon}{c})}$ with sufficiently large constant degree d , such that every set T of $(\frac{1}{2} + \frac{\epsilon}{c})n$ nodes has fewer than $(\frac{1}{2} + \frac{\epsilon}{c})\epsilon d n$ edges to S , where S is any set of ϵn nodes in the graph. Further, by Lemmas 6, 7, 8, we know that there is a randomized construction for a d -regular expander for sufficiently high degree d that satisfies properties 1-3 with high probability. We deterministically choose a graph that satisfies these properties. To show that property 4 holds, we will assume that it doesn't hold and then arrive at a contradiction. Consider an arbitrary set S of ϵn nodes in the graph. Assuming that property 4 does not hold, we create a set U' consisting of $(\frac{1}{2} + \frac{\epsilon}{c})d$ neighbors of each node in the set S such that $|U'| \leq \frac{n}{2}$. If we consider the multiset of $(\frac{1}{2} + \frac{\epsilon}{c})d$ neighbors of each node in S , the size of the multiset is $(\frac{1}{2} + \frac{\epsilon}{c})\epsilon d n$. By the construction of our expander, every set of ϵn nodes expands to more than $(1 - \frac{\epsilon}{c})n$ nodes. Refer to the set of $(1 - \frac{\epsilon}{c})n$ nodes that S expands to as Y . In order for the set U' to exist as defined, within Y there must be a set T of $(\frac{1}{2} + \frac{\epsilon}{c})n$ nodes containing U' such that there are more than $(\frac{1}{2} + \frac{\epsilon}{c})\epsilon d n$ edges with one vertex in S and one in T , where an edge exists between two nodes if they are neighbors in G . We have arrived at a contradiction, as this violates that the graph satisfies the property in Lemma 11. Therefore the lemma holds. ◀

B Proofs for Provable Broadcast

► **Lemma 12 (Provability).** *In $(\frac{n}{2} + 1)$ -PB-Initiate, if the sender delivers two valid proofs σ_{out} and σ'_{out} corresponding to values (v, σ'_{in}) and (v', σ'_{in}) respectively, then (i) $v = v'$, and (ii) at least $\frac{n}{2} + 1$ parties satisfy the criteria in the validate() function, and the parties have created attestations in createAttestations() such that they satisfy validateNeighbor().*

Proof. Since σ_{out} contains a threshold signature for (v, σ_{in}) signed by at least $\frac{n}{2} + 1$ parties, at least $\frac{n}{2} + 1 - t > \epsilon n$ honest parties p_i must have sent messages “ $id, vote, \xi_i$ ” to the sender. Thus, each such p_i must have received at least $(\frac{1}{2} + \frac{\epsilon}{2})d$ messages “ $id, send, ((att_{\log Id}, att_{head,j}), \sigma_j)$ ” from parties p_j such that the attestations $(att_{\log Id}, att_{head,j})$ along with σ_j satisfy the criteria in the `validateNeighbor()` function. Thus, $(att_{\log Id}, att_{head,j})$ were created by running `createAttestations(id, (v, \sigma_{in}))` and σ_j proves that (v, σ_{in}) is valid for p_j 's state based on the conditions in `validate(id, (v, \sigma_{in}))`. By Lemma 4, any ϵn set of parties each receiving attestations from $(\frac{1}{2} + \frac{\epsilon}{2})d$ neighbors, should collectively receive attestations from at least $\frac{n}{2} + 1$ parties such that they satisfy `validateNeighbor()`. This completes part (ii) of the proof. For part (i), observe that any two quorums of size $\frac{n}{2} + 1$ will always intersect at one party, and due to the use of trusted hardware, this party cannot attest to two different values v and v' such that $v \neq v'$ for the same log and sequence number. ◀

► **Lemma 13 (Termination).** *If the sender is honest, no honest party invokes `abandon(id)`, all messages among honest parties arrive, and `validate(id, (v, \sigma_{in})) = true` for all honest parties, then (i) eventually all honest parties deliver v , and (ii) the sender delivers v with a valid proof σ_{out} .*

Proof. Observe that the sender's message will eventually arrive at all $(1/2 + \epsilon)n$ honest parties if no party invokes `abandon()`. Since the message sent by the sender is valid for all honest parties, all honest parties will invoke `createAttestations()` and deliver the sender's message along with a valid attestation as the proof. They then send their attestations to all their neighbors. By Lemma 6, at least $\frac{n}{2} + 1$ of the honest parties H will each receive attestations from at least $(\frac{1}{2} + \frac{\epsilon}{2})d$ of their neighbors that satisfy `validateNeighbor()` without any participation from any Byzantine parties. Each party in H will send a *vote* message with a threshold signature share to the sender, who can combine them into (v, σ_{out}) . ◀

► **Theorem 14.** *The $(\frac{n}{2} + 1)$ -Provable Broadcast algorithm in Algorithms 1 and 2 satisfies Integrity, Validity, Provability, and Termination. Moreover, the protocol has linear communication complexity with an $O(1)$ -sized proof.*

Proof. Integrity is satisfied deterministically by the algorithm. All the messages from the sender to the parties and vice-versa involve messages with $O(1)$ words. All the communication between parties through the expander graph consists of $O(1)$ sized messages to a constant d number of neighbors. Thus the communication complexity is linear. Also, the proof delivered by the sender is a threshold signature of $O(1)$ size.

An honest party only sends a signature share to s for a value v if v is externally valid as per the `validate()` function. Therefore, as long as the threshold for the threshold signature is greater than the number of Byzantine parties in the network, only an externally valid message can obtain a valid threshold signature, satisfying validity.

Provability and Termination property have been shown in Lemmas 12 and 13. ◀

C Proofs for HotStuff-M

► **Lemma 15.** *At the end of a view e , (i) if a party receives a σ_{commit} on value v , then $\geq \frac{n}{2} + 1$ parties appended value v at position e in their LOCK logs prior to appending a value at position e in their PROGRESS logs, and (ii) if a party receives a σ_{lock} on value v , then $\geq \frac{n}{2} + 1$ parties appended value v at position e in their KEY logs prior to appending a value at position e in their PROGRESS logs.*

Proof. This lemma follows from Lemma 12 and the criteria for `validateNeighbor()` in Table 2. ◀

► **Lemma 16.** *Suppose the earliest view in which a value v is committed by an honest party is e . For all views $> e$, a valid σ_{key} for a value $v' \neq v$ does not exist.*

Proof. Suppose for contradiction that v has been committed by an honest party in view e and a σ_{key} for $v' \neq v$ exists in view $e' > e$. Let e^* be the earliest view in which a σ_{key} for a value v^* is formed such that $v^* \neq v$ and $e^* > e$. It follows that $e^* \leq e'$. Since there is a σ_{key} for v^* in e^* , by Lemma 12, a set Q of at least $\frac{n}{2} + 1$ parties have sent messages with attestations $(att_j, att_{counters,j}), \sigma_j$ that satisfy `validateNeighbor()`. Since v was committed in view e , there exists a set P of at least $\frac{n}{2} + 1$ parties who have inserted v into their LOCK log. The two sets P and Q should intersect at least one party p .

We now show a contradiction w.r.t. p 's log and its attestation satisfying `validateNeighbor()`. Since view e^* is the first view where a higher σ_{key} was formed for a different value, the end state of LOCK in view e^* must be for value v . Thus, the predicate $view(\sigma_{in}) > view(\sigma_{lock})$ in the PROMOTE phase is not satisfied. Moreover, in view e^* , the proposed value $v^* \neq v$ for our setup. Thus, the condition $\sigma_{in}.val = \sigma_{lock}.val$ in PROMOTE phase is not satisfied either. Additionally, since a party presents the state of their PROGRESS log through the $att_{counters,j}$ attestation, they always present the state of their logs after the end of view $e^* - 1 \geq e$. Thus, for party p , `validateNeighbor()` cannot be satisfied. Consequently, σ_{key} in view e^* cannot be formed for value v' , a contradiction. ◀

► **Theorem 17 (Safety).** *Two honest parties p_i and p_j cannot commit to values v and v' such that $v' \neq v$.*

Proof. Let e be the view in which v is committed and e' be that of v' s.t. $v \neq v'$. By Lemma 12, $e \neq e'$. Suppose, without loss of generality, $e > e'$. By Lemma 16, no σ_{key} can be formed in view $> e$ for value $\neq v'$. Honest parties only vote for σ_{lock} in the LOCK phase in view e' if it was generated in view e' ; thus, a σ_{commit} cannot be formed in view e' . ◀



► **Theorem 18 (Liveness).** *After GST, there exists a bounded time such that when an honest leader is elected in view e and all honest parties remain in view e for that time, then a value will be committed by all honest parties.*

Proof. View e is a view after GST where the leader is honest. Suppose σ_{lock}^* is the σ_{lock} stored in a party's LOCK log from the highest view e^* for a value v . By Lemma 15, at least $\frac{n}{2} + 1$ parties must have the σ_{key}^* for value v stored in their LOCK logs at position e^* prior to creating the COUNTERS() attestation to report their highest σ_{lock} in e . Since the leader waits for $\frac{n}{2} + 1$ valid σ_{key} messages during a view change, it will obtain σ_{key} from a view $\geq e^*$. Let σ'_{key} be the highest valid σ_{key} received by the leader. The leader will propose $(\sigma'_{key}.val, \sigma'_{key})$ in the PROMOTE phase. Since e^* is the highest view for which a party has a lock, it must be the case that for each honest party, either $view(\sigma'_{key}) >$ than the locked view of the party or that v is the value that the party is locked on (Lemma 12). Since e is after GST, all messages will arrive within the bounded delay Δ , and thus for each of the three phases, the termination property for provable broadcast from Lemma 13 should be satisfied. Thus, all honest parties will commit. ◀

Chopin: Combining Distributed and Centralized Schedulers for Self-Adjusting Datacenter Networks

Neta Rozen-Schiff  

School of computer science and engineering, The Hebrew University of Jerusalem, Israel

Klaus-Tycho Foerster  

Computer Science Department, TU Dortmund, Germany

Stefan Schmid  

TU Berlin, Germany

Faculty of Computer Science, Universität Wien, Austria

David Hay  

School of computer science and engineering, The Hebrew University of Jerusalem, Israel

Abstract

The performance of distributed and data-centric applications often critically depends on the interconnecting network. Emerging reconfigurable datacenter networks (RDCNs) are a particularly innovative approach to improve datacenter throughput. Relying on a dynamic optical topology which can be adjusted towards the workload in a demand-aware manner, RDCNs allow to exploit temporal and spatial locality in the communication pattern, and to provide topological shortcuts for frequently communicating racks. The key challenge, however, concerns how to realize demand-awareness in RDCNs in a scalable fashion.

This paper presents and evaluates *Chopin*, a hybrid scheduler for self-adjusting networks that provides demand-awareness at low overhead, by combining centralized and distributed approaches. *Chopin* allocates optical circuits to elephant flows, through its slower centralized scheduler, utilizing global information. *Chopin*'s distributed scheduler is orders of magnitude faster and can swiftly react to changes in the traffic and adjust the optical circuits accordingly, by using only local information and running at each rack separately.

2012 ACM Subject Classification Networks → Programmable networks; Networks → Data center networks

Keywords and phrases reconfigurable optical networks, centralized scheduler, distributed scheduler

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2022.25

Supplementary Material *Software (Code)*: https://bitbucket.org/NetaRS/sched_analytics
archived at `swh:1:dir:ba2af62f8b1e8f483cb493908b711f9de4dbf488`

Funding Research supported by the European Research Council (ERC), grant agreement No. 864228 (AdjustNet), Horizon 2020, 2020-2025, a grant from Fraunhofer SIT, and the Israeli Innovation Authority through the Peta-Cloud consortium.

1 Introduction

Data-centric and distributed applications, including batch processing, streaming, scale-out databases, or distributed machine learning, generate a significant amount of network traffic and their performance critically depends on the throughput of the underlying network [31, 85].

To improve datacenter throughput, researchers and industry, e.g., Google [64], have recently started exploring innovative new datacenter designs that rely on dynamic and *demand-aware* topologies: topologies that self-adjust toward the workload they currently serve. The motivation behind self-adjusting datacenter topologies is twofold.



© Neta Rozen-Schiff, Klaus-Tycho Foerster, Stefan Schmid, and David Hay;
licensed under Creative Commons License CC-BY 4.0

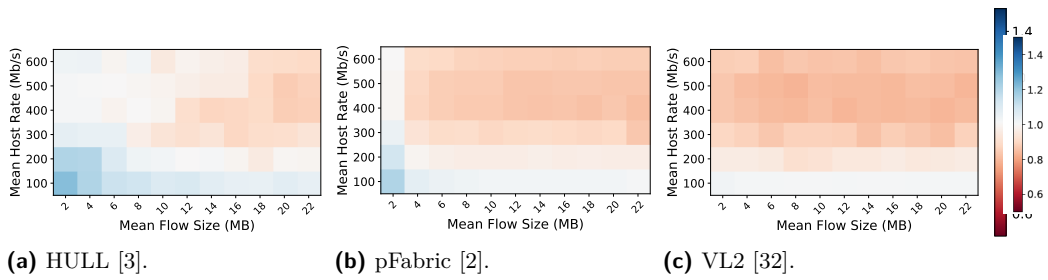
26th International Conference on Principles of Distributed Systems (OPODIS 2022).

Editors: Eshcar Hillel, Roberto Palmieri, and Etienne Rivière; Article No. 25; pp. 25:1–25:23



Leibniz International Proceedings in Informatics

LIPIC Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



■ **Figure 1** Comparison between centralized and distributed schedulers under different traffic patterns (each generated by scaling well-known realistic flow size distributions, assuming Poisson flow arrival times under different rates), when the number of ToR switches is 80, and the optical connectivity of each ToR switch is 4. The color represents the ratio between the optical throughput of the distributed scheduler and the centralized scheduler. Blue cells mark settings where the distributed scheduler outperforms the centralized one; red cells mark the opposite. We refer to §6.1 for topology details.

First, empirical studies reveal that datacenter traffic patterns feature much structure [6, 10, 31, 66], i.e., are sparse, skewed, and bursty, which introduces optimization opportunities. For example, a small number of flows typically carry the majority of traffic (these are called *elephant flows*), while the remainder consists of a large number of flows that carry very little traffic (*mice flows*). Therefore, in a demand-aware network, the elephant flows should be routed through the optical circuits for offloading the electrical bottleneck, which in turn, reduces the latency of the mice flows, and improves the overall throughput.

Second, emerging optical technologies and optical circuit switches enable the required very fast reconfigurations [8, 21, 22, 33]. Over the last years, several interesting hybrid optical datacenter networks were suggested and evaluated [83], augmenting an oversubscribed network with inter-rack optical links [7, 14, 22, 24, 25, 31, 53, 69, 70, 77, 79], see [27] for a survey. The number of optical routes from/to each Top-of-Rack (ToR) switch, which we call the *ToR switch optical degree*, is a single-digit number, typically at most 4 [21, 71, 82].

Challenge: Scalability. While the vision of self-adjusting networks is intriguing and early solutions show promising results, the main challenge faced by such demand-aware networks concerns the scalability of the control plane. Unlike demand-oblivious networks (i.e., static networks like Clos [15], Slim Fly [12], and Xpander [76] or dynamic networks like RotorNet [59], Opera [58] and Sirius [8]), demand-aware networks require the collection and evaluation of traffic patterns. In particular, performing all topology scheduling decisions *centrally* (i.e., a centralized scheduler) may introduce a bottleneck and can result in slow reaction times. A fully distributed decision making (i.e., a distributed scheduler) on the other hand, may be suboptimal as it is based on incomplete information.

In order to show this tradeoff, we analyzed the *optical throughput ratio*. The optical throughput ratio is defined as the ratio between the throughput routed through the optical circuit and the total datacenter throughput. It is a cornerstone measure as it reflects the utilization of the optical circuit, and therefore reduces the bottleneck over the electrical network. Fig. 1 compares the optical throughput ratio of the distributed-only scheduler and the centralized-only scheduler under different traffic patterns (each traffic pattern follows a distribution measured in a real datacenter, where we have parametrized the mean flow size and each host rate). It demonstrates the tension between the two approaches: there

is no “clear winner” and which one is better depends on the traffic pattern. The traffic pattern however is often not known when the datacenter is built and changes over time. For example, consider a datacenter serving a pFabric traffic pattern, with typical mean flow size of approximately 1.7 MB [51], and where each host sending rate is approximately 100 Mbps. In this case, the optical circuit throughput ratio in the distributed scheduler is by 13% higher compared to the centralized scheduler, as can be seen in the blue cells in Figure 1b. However, for the same datacenter, with the same traffic pattern (pFabric), and the same mean flow size distribution, once the host’s sending rate grows beyond 300 Mbps, the centralized scheduler achieves a higher throughput ratio compared to the distributed one (see the relevant red cell).

Motivated by this insight, and by the desire to provide an efficient control plane for self-adjusting networks, we propose to combine both approaches to achieve the best of both worlds: fast reaction times of distributed decision-making and network utilization benefits of centralized optimization.

Introducing Chopin. We present *Chopin*¹, a novel scheduler for reconfigurable datacenter networks that fully exploits the benefits of self-adjusting networks by relying on an efficient control plane. Specifically, *Chopin* provides demand-awareness at low overhead, by combining centralized and distributed approaches. At the heart of Chopin’s approach lies the idea that a relatively complex algorithm (e.g., Maximum Weight Matching, MWM) should be computed centrally, based on complete information.

However, since such an algorithm cannot be computed fast [11, 16, 48] (e.g., MWM may take around 20 ms for 80 ToR switches), we additionally allow distributed *updates* to the centralized optical circuit allocation, based on a *threshold*. The threshold specifies the flow weight changes from which a distributed scheduler can update the centralized scheduler allocation. For example, if there is a large drop in demand in an allocated optical circuit (e.g., when an elephant flow ends), the distributed scheduler may tear it down and try to establish another circuit. Hence, due to the volatility of many flows, we want a distributed constant-round algorithm (ideally just two rounds) and hence forgo more complex distributed algorithms [9] or dynamic centralized algorithms [13]; the indirection via a centralized controller comes with overheads and delays which render this approach problematic to handle continuous update streams.

Our Contributions. In summary, we make the following contributions:

1. We identify and analyze the difference in throughput performance of centralized and distributed schedulers for reconfigurable datacenter networks, for various scenarios and different flow size distributions.
2. We design a hybrid scheduler, *Chopin*, which combines centralized and distributed decision-making based on thresholds. To this end, we present and analyze both a centralized and a local online scheduler, exploring the trade-off between accuracy and running time. *Chopin* relies on commodity devices available today, and required Chopin nodes which can simply be added to existing ToR switches by directing one of the switch ports to them. Moreover, information collection and dissemination of the centralized algorithm can be realized in the control plane using Software-Defined Networks (SDNs).
3. We report on Chopin’s effectiveness through extensive simulations for different settings, showing that Chopin improves upon centralized and distributed approaches. We achieve throughput improvements of up to 20% against centralized and up to 23% against distributed schedulers, *always outperforming both*.

¹ Stands for: Controller for Hybrid OPTical electrical Networks.

2 Optical Background and Related Work

Chopin is motivated by trade-offs between centralized and distributed scheduling, which arise in matching algorithms. We first motivate why matching algorithms are central to Chopin’s setting and then discuss centralized and distributed schedulers in this context.

Optical Model: Why Matchings? From a theoretical viewpoint, we consider the problem of how to augment a static network with (optical) edges in order to improve the total network performance. The reason why this augmentation comes in the form of matchings lies in the underlying hardware, namely optical circuit switches, we refer to Hall et al. [36, §3] for a technological overview. In the simplest case, a set of nodes is connected to the optical circuit switch’s ports by an optical cable each, and the switch “matches” these ports by e.g. adjusting mirrors to steer the light signals s.t. that pairs of ports (and hereby, pairs of nodes) are hence connected by optical circuits. Nodes could also be connected multiple times to the optical switch, or multiple optical switches could be used, giving rise to, e.g., b -matchings [26, 39].² Conceptionally, other hardware could be used to the same effect (e.g., beamformed wireless connections [37] or free-space optics [7]), but on a graph-theoretic level, they form circuits between pairs of nodes, and as thus, matchings. We refer here to the survey by Foerster and Schmid [27] for a further introduction to the enablers, algorithms, and complexity of reconfigurable datacenter networks. We moreover refer to the article by Zerwas et al. [84] on how system delays can be accounted for for scheduling algorithms.

Centralized schedulers. Centralized schedulers operate under the assumption of near-perfect utilization visibility and traffic demands, collected at a centralized location [18], often leveraging SDN. We refer to a recent survey and the references therein [74]. Herein the restriction to large and long-lived flows enables centralized schedulers [24, 79] to also cope with control loop delays. However, these schedulers still suffer from traffic stability assumptions [23].³ Traffic matrix schedulers [22, 54, 77, 78] on the other hand, adjust packet transmissions to coincide with scheduled circuit reconfiguration, with full knowledge of when bandwidth will be available to particular destinations. However, for the duration of the matching schedule, new flows are not accounted for and might need to wait for the next iteration. In contrast, Chopin’s design ensures rapid reactions to local traffic changes and new flow insertions, due to its additional distributed scheduler part.

Distributed schedulers. In practice, the large number of scheduling decisions and status reports can overwhelm centralized schedulers, and in turn lead to long latencies before scheduling decisions are made [18]. ProjecToR [31] initiated a broader interest in distributed scheduling, by proposing a stable-matching algorithm that optimizes for low latency, utilizing high fan-out single hop free-space optics [30]. Via aging of requests, they obtain a constant-factor latency approximation for their online scheduling algorithm [19]. RotorNet [59], Opera [58], and Sirius [8] employ a different approach and use lower fan-out circuits, where the topologies are created in a demand-oblivious manner. RotorNet rotates through matchings independent of the current traffic, that provide eventual connectivity, where traffic is either

² There is also some work that considers multicast by splitting the outgoing light signals [17, 57, 72].

³ Orthogonal to matching algorithms, Xia et al. [80] investigate how to migrate between Clos and random graph topologies. However, they require specialized 4/6-port converter switches and also rely on a centralized control loop, estimating an update delay “*on the order of seconds*” [80].

scheduled to be routed along single hops, or along two hops, via buffering and a proposal and accept mechanism. Sirius follows similar ideas, either transmitting directly or via schemes reminiscent of Valiant’s method. Opera extends RotorNet by also always maintaining an expander graph, motivated by static topologies [46, 76]. Although Opera’s reconfiguration scheduling is deterministic, the precomputation of the topology layouts is in its current form still randomized. Notwithstanding, ProjecToR, RotorNet, Sirius, and Opera can all rapidly deploy traffic along reconfigurable connections, by omitting a centralized control plane. However, it is not clear how to realize the above three distributed systems with off-the-shelf hardware, such as a common optical circuit switch, and hence their application scenario is not as general as with Chopin. Notwithstanding, Decentralized scheduling is also used in several other systems, including SplayNet [68], Cerberus [35], or CacheNet [34].

Lastly, while there is profound research on matching algorithms in the distributed computing community [73], distributed algorithms for maximal matchings in graphs with large degree Δ (as for optical circuit switches) are relatively slow [9]. While approximation [55] and dynamic [56] algorithms are considerably faster, here the constraints of the optical datacenter networking and the distributed computing community are quite different and hence the communities (yet) don’t overlap much in their research applications: ideally, for optical circuit switching, small-constant round algorithms of low computational complexity are desired, whereas in the distributed computing community, the local algorithms can be more complex, with a focus on asymptotic runtime optimization. As thus, Chopin utilizes a low complexity threshold based distributed algorithm, using just two rounds of communication, which falls in line with the requirements of hybrid datacenters.

3 Chopin’s Design

In a nutshell, *Chopin*’s topology scheduler aims to provide demand-awareness efficiently by combining centrally optimized decision making with fast distributed reactions. The idea is hence analogous to the nervous system of animals, which is typically divided into a slower central nervous system and a faster peripheral nervous system [75].

Specifically, *Chopin*’s scheduler uses two different control mechanisms, each carried out in a different location in the datacenter, providing different latency and response times. The **centralized scheduler** is reminiscent of an SDN controller and allows *Chopin* to adapt to global changes (such as traffic rates). This optimization uses traffic measurements across the network and has a (relatively) long response time. Moreover, it may receive additional information (e.g., from applications that have specific repetitive patterns) to make even better decisions. Fig. 2 presents the connectivity between the SDN controller to each of the ToR switches and *Chopin*’s nodes. The **distributed scheduler** is embedded within the ToR switches and is based only on local measurements. It reacts quickly to local changes in traffic and may tear down connections if they become unmatched and establish new connections for new “hot” ToR switch pairs. The tear down and connection establishment are made by updates sent from the ToR switch to its *Chopin* node, see Fig. 2.

The centralized scheduler and the distributed scheduler are discussed in details in Section 4 and Section 5 respectively.

Moreover, by combining these two schedulers, we can strike an optimized trade-off and realize both fast reactions and global and long-term network optimizations, accounting for demand uncertainty. In particular, unlike many existing solutions, which consider only one scheduler, *Chopin* is flexible and performs better than both.

3.1 The Hybrid⁴ Topology

Chopin can be used together with any fast switching circuit technology (as in [22,23,62]), and implemented within the existing datacenter hardware. We distinguish between two entities in the ToR switches: the electric switch itself (for brevity, we will simply refer to this switch as the ToR switch), and the Chopin node which resides in the switch, serving as the entry point to the optical network. This modular Chopin structure enables us to support existing ToR switches, by directing one of its upstream ports to the Chopin nodes. When clear from the context, we use the terms, ToR switch and Chopin node, interchangeably.

The optical network can be any non-blocking topology, where the only constraint on establishing a circuit between two ToR switches is the availability of a transceiver in the corresponding Chopin node (namely, its optical degree).

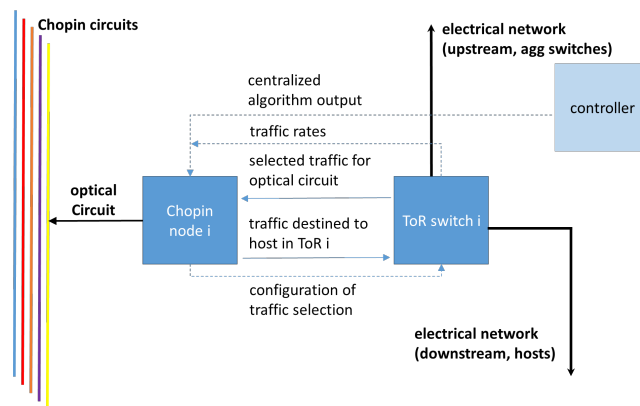
Specifically, we assume each Chopin node has an optical degree of k and optical circuits are symmetric. This implies, that at any given time, a Chopin node can send and receive data *from at most k Chopin nodes*. For any given time t , we denote by $dest_i(t)$ the set of Chopin nodes connected to the Chopin node i . We observe that as circuits are symmetric, if $j \in dest_i(t)$ then it also holds that $i \in dest_j(t)$.

3.2 Problem Formulation

At the heart of Chopin lies the desire to improve network performance and throughput by avoiding scheduling bottlenecks. As Chopin is deployed between ToR switches, the scheduler is oblivious to intra-rack traffic or delays.

We first need to introduce some preliminaries. Let n be the number of ToR switches in the network and assume that time is slotted, where in each time-slot the distributed scheduler can be invoked (e.g., the length of each time-slot is 1 ms). Let $X_{i,j}(t)$ be the total amount of traffic sent from rack i to rack j at time-slot t . Now let $Y_{i,j}(t)$ be an indicator variable to describe whether a pair of ToR switches is connected through a Chopin circuit at time interval t : $Y_{i,j}(t) = 1$ if and only if $j \in dest_i(t)$ (and 0 otherwise). If $Y_{i,j}(t) = 1$ then $Y_{j,i}(t) = 1$ as connections through Chopin are symmetric.

⁴ We note that the term *hybrid* can have a different meaning in some networking contexts, e.g., indicating a combination of the LOCAL model with the node-capacitated clique model [5].



■ **Figure 2** Chopin's design.

Let $C_t \subseteq S \times S$ be a symmetric relation with all ToR switch pairs that are connected through a Chopin circuit at time interval t (i.e., $(i, j) \in C(t)$ if and only if $j \in \text{dest}_i(t)$).

We aim to maximize optical circuit throughput, a standard objective in such topologies [7, 14, 22, 24, 54, 59, 78, 79], namely $\sum_t \sum_i \sum_j X_{i,j}(t) \cdot Y_{i,j}(t)$. This relieves the electrically switched network part and reduces the overall latency. This is done by updating the set C_t , based on local and centralized decisions.

Note that, as optical circuit capacities are typically very high, we assume that the capacity of an optical circuit is always larger than the total amount of traffic sent between two racks (namely $X_{i,j}(t)$). In case this does not hold, and the two racks are connected through an optical link, one can send traffic through the optical circuit up to its capacity while the remaining traffic is sent through the electrically switched network.

3.3 Schedulers and Definitions

First, a *centralized* scheduler has a global view of the network and, in some cases, even auxiliary information given by the network administrator. This, on one hand, enables the scheduler to perform more informed decisions. But on the other hand, when using a centralized scheduler, it can take much longer to gather, compute, and spread the information across the datacenter. In our model, we assume the centralized scheduler works every T time-slots (which we call the *centralized scheduler epoch*) and uses slightly outdated traffic information: at time t , only the measurements $\{X_{i,j}(t') | t' < t - \Delta, \text{ for every } i, j\}$ can be used, where Δ is the *centralized algorithm delay*: the time it takes it to gather all information and make decisions. For example, if the optical degree is 1 (i.e. $k = 1$), the centralized scheduler may use algorithms such as maximum weight matching to optimize the throughput that goes through the optical circuits.

As T becomes larger, centralized scheduler decisions can deteriorate, as the input on which decisions are based is outdated toward the end of the epoch. Thus, we additionally consider a distributed scheduler that is more fine-grained and runs every time-slot, benefiting from a reduced computation time and avoiding the delays involved in the centralized scheduler; it changes the pairs of connected switches based on local information only and by exchanging messages between ToR switches in two rounds. Specifically, the distributed scheduler of node i at time-slot t may use traffic measurements on its node until the computation starts:

$$\{X_{i,j}(t') | j \neq i, t' < t - \delta\} \cup \{X_{j,i}(t') | j \neq i, t' < t - \delta\},$$

where $\delta < \Delta$ is the distributed scheduler delay. In addition, the distributed scheduler is aware of the information sent to it by other nodes throughout the rounds of computation. Importantly, for each pair (i, j) that was optically connected through Chopin at time interval $t - 1$ (namely, $Y_{i,j}(t - 1) = 1$), the distributed scheduler at node i knows what information was used to establish this connection (e.g., what is the rate reported to the centralized scheduler upon its last invocation) and decides whether the information is stale or not. Table 1 summarizes Chopin schedulers' notations.

4 Chopin's Centralized Scheduler

The centralized scheduler is implemented on top of a centralized SDN controller, which is (logically) connected to each of the ToR switches and the Chopin nodes. Upon a request from the centralized scheduler, the controller collects traffic measurements across the network (namely, counters at ToR switches, current status of Chopin nodes). Based on these measurements, it computes the next optical circuit allocation.

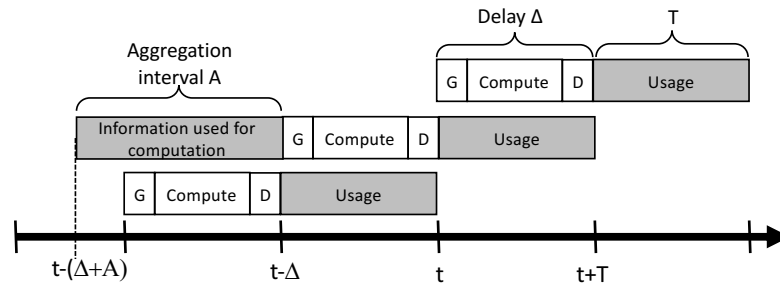
■ **Table 1** Chopin’s schedulers’ notations.

Notation	Meaning
n	The number of ToR switches.
$dest_i(t)$	The set of racks optically connected to rack i at time-slot t
$X_{i,j}(t)$	The total amount of traffic sent from rack i to rack j at time slot t
$Y_{i,j}(t)$	Indicator variable. $Y_{i,j}(t) = 1$ iff $j \in dest_i(t)$
C_t	The set of rack pairs optically connected at time slot t
K	Optical degree, the number of available circuits per Chopin node.
Δ	Centralized scheduler delay
δ	Distributed scheduler delay
α	Chopin threshold for keeping centralized decisions
A	Centralized scheduler aggregation interval
a	Distributed scheduler aggregation interval
T	Centralized scheduler epoch

Recall that the *delay* Δ is the time it takes to send all the information to the controller, run the centralized algorithm, and send the decisions back to the nodes. The centralized algorithm works in epochs of length T , where decisions arrive at the nodes at the beginning of each epoch. Assume an epoch starts at time t . Then, these decisions will be used by nodes until time $t + T$ (or until altered locally by the distributed scheduler). Furthermore, these decisions are based on information gathered in the interval $[0, t - \Delta]$. However, if traffic changes quickly (DC traffic is often bursty [6]), this information may be outdated quickly. This motivates us to define an *aggregation interval* A for the centralized algorithm, considering only the interval $[t - (\Delta + A), t - \Delta]$, see Fig. 3.

Notice that the delay Δ is an important factor for the performance of Chopin. The delay describes the response time of the central scheduler and consists of several steps: contacting tens to hundreds of nodes [20, 44], (2) receiving thousands of flow entry statistics, estimating optical circuit utilization, contacting all nodes again, and updating all rules with new parameters if needed.

Considering common SDN controllers’ capability to handle a few thousands of messages per second [81], we estimate the delay to be in the order of hundreds of milliseconds in most configurations [49] [40]. Furthermore, the computation time of our algorithms can be in the order of tens of milliseconds for hundreds of ToR switches (e.g., when running maximum weight matching-like algorithms, as reported in [24]). Due to these delays, fast changes in the network (occurring within a few milliseconds [66]), may not be detected by the centralized scheduler in a timely manner. Also, the reconfiguration time (approximately 11 μ sec [31] [63]) is likely negligible compared to a centralized reconfiguration cycle. These observations motivate usage of another scheduling layer, to adapt to traffic in an online manner.


 ■ **Figure 3** Centralized scheduler timing parameters, where “G” stands for the gathering period and “D” for the disseminate period. Similar parameters are used by the *distributed scheduler*, with delay of δ .

Our high-level goal is to maximize the overall throughput over the optical network. First recall that allocations are constrained by the optical ToR switch degree k : each ToR switch can be optically connected to at most k other ToR switches. Accordingly, our centralized algorithm essentially needs to solve a weighted b -matching problem, with $b = k$. Specifically, we consider an undirected graph whose nodes are the ToR switches and the weight of each edge (i, j) is the total traffic between i and j in the relevant interval:

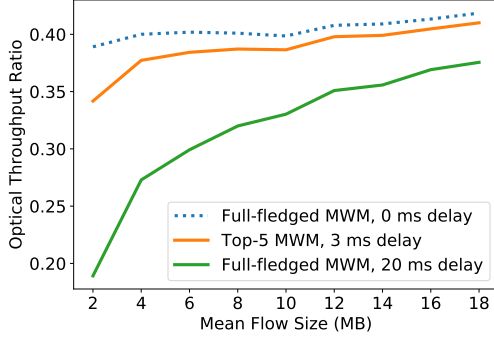
$$w_{ij} = \sum_{t'=t-(\Delta+A)}^{t-\Delta} X_{i,j}(t') + X_{j,i}(t').$$

While b -matching algorithms are strongly polynomial [4], their running time can still be prohibitively high in practice [28, 50, 61]. This can lead to high delays Δ and in turn, to a significantly reduced overall performance of the system. Thus, we propose to approximate the problem: we compute a maximum weight matching (using Edmond’s MWM algorithm [29]), subtract the weights of the matching’s edges from the graph, and run maximum weight matching again with the new, smaller weights. As in [70], this process is repeated k times, resulting in k matchings. Hence each node is connected to at most k other nodes, as required. We refer to Khan et al. [47] for a further discussion on efficiently approximating b -matchings.

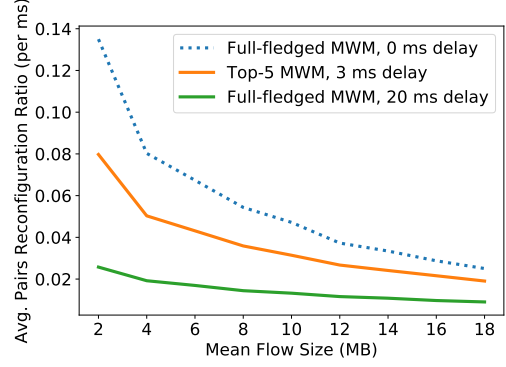
We further reduce computation time by considering only the top- m live flows per ToR switch (instead of all possible pairs between the nodes). Due to the sparse nature of datacenter traffic matrices, even small values of m provide a highly accurate approximation: there is almost no performance degradation compared to a full-fledged MWM (§5). Note that the top- m flows per switch can be efficiently calculated in each switch since there are only n possible flows and maintaining n counters at line rate is supported by switches.

Moreover, focusing only on a constant number of top flows per node enables Chopin to scale with an increasing number of nodes. It also decreases both the MWM computation and the network reconfiguration times, allowing more frequent centralized scheduler reconfigurations. As the frequency of centralized scheduler invocations significantly affects the performance, by considering only m flow, we can improve the scheduler’s performance. For example, when $m = 5$ and the number of ToR switches is 80, the time it takes to compute MWM based on top-5 live flows per ToR is 3 ms, while full-fledged MWM takes at least 20 ms. Fig. 4 compares the performance of both algorithms under the pFabric traffic pattern we have described (similar results hold for other traffic patterns as well) and shows that having more frequent reconfigurations is more significant than having slightly better matchings. Our centralized scheduler, based on top-5 live flows with reconfiguration every 3 ms, achieves almost the same results as an idealized online optimal algorithm, that computes full-fledged MWM every 1 ms. Finally, we observe that the optical throughput ratio improves as the mean flow size increases (and the gap between the algorithms shrinks), since longer flows imply that flow information is still relevant even after a long time when computations are infrequent.

In order to explain the throughput differences in Fig. 4, we analyzed the number of reconfigurations in each scenario. We consider the average number of reconfigured pairs in each run, out of the total number of pairs $(n/2)$. Fig. 5 presents the reconfiguration average ratio per 1 ms, as a function of the mean flow size. As expected, as the mean flow size increases (and the flows are longer), the number of reconfigurations decreases. Moreover, it shows that the number of reconfigurations is decreasing rapidly when the epoch time is 20 ms. This can be attributed to the fact that short lived connections have less impact on the 20ms long measurements and are less likely to be matched.



■ **Figure 4** Comparison between a centralized scheduler which operates every 3 ms and computes the MWM of the top-5 live flows, to a centralized scheduler which operates every 20 ms and computes full-fledged MWM. For brevity only pFabric results are shown.



■ **Figure 5** Average reconfiguration ratio per 1 ms of three scenarios: centralized scheduler which computes MWM over top-5 live flows every 3 ms, a centralized scheduler which operates every 20 ms and computes full-pledged MWM. For brevity only results for the pFabric traffic pattern are shown.

5 Chopin's Distributed Scheduler

The distributed scheduler is a distributed control algorithm, embedded inside each Chopin node. Each ToR switch is connected to a single Chopin node and sends flows to the latter (e.g., by connecting one of its ports to the Chopin node). The traffic that is sent through this port is configured either by our centralized algorithm (as described in Section 4) or by the distributed scheduler that runs on the Chopin node.

The Chopin node is responsible of sending traffic destined for the ToR switch from one of the optical circuits. Each ToR switch in turn is connected to a single Chopin node. We refer to the illustration in Fig. 2 for an overview. Supplemental pseudo-code of Chopin's distributed algorithm appears in Algorithm 1 in the Appendix.

At the beginning of each centralized scheduler epoch, every Chopin node keeps track of the traffic rate according to which its circuit was selected. Namely, for an epoch that starts at time t , if a circuit was established between Node i and Node j , both Node i and Node j compute and store:

$$R_{i,j}(t) = \frac{1}{A} \sum_{t'=t-(\Delta+A)}^{t-\Delta} X_{i,j}(t') + X_{j,i}(t').$$

The nodes use these rates to determine if traffic demands stay steady during the epoch. Specifically, we define a *threshold* $\alpha \geq 0$, and compute the rate in each time-slot $\hat{t} \in [t, t+T]$ based only on local information available at the nodes:

$$r_{i,j}(\hat{t}) = \frac{1}{a} \sum_{t'=\hat{t}-(\delta+a)}^{\hat{t}-\delta} X_{i,j}(t') + X_{j,i}(t'),$$

where δ is the distributed scheduler delay, and a is its aggregation interval. Only if $r_{i,j}(\hat{t}) > \alpha \cdot R_{i,j}(t)$, then the circuit is marked as matched and the algorithm keeps it connected through this epoch. Otherwise, it strives to replace it with a better connection, as described

next. We observe that for $\alpha = 0$, all existing connections are kept matched (namely, Chopin just runs the centralized scheduler, and may improve only when its computed b-matching changes). As the threshold increases, it enables the distributed scheduler to tear down almost every centrally-computed connection and to create new ones, based on the current ToR switch traffic. The distributed scheduler itself tries to establish as many circuits as possible to increase the overall traffic through the optical circuit connections. In a nutshell, each Chopin Node i sends requests to a predetermined number of other nodes needs (this number is denoted by variable `max_reqs`), for which it observes the most bi-directional traffic. These nodes, denoted by `req_nodes`, do not include those kept matched to Node i ; moreover, `max_reqs` $>$ k to allow utilizing all the circuits connected to Node i . After a Chopin node i sends its requests, it waits to receive requests from other nodes. We distinguish between:

1. Request from a node j that is in the `req_nodes` set: This means that both nodes i and j consider the traffic between them in their top `max_reqs` links. This makes Node j a candidate for a match with Node i .
2. Request from a node j that is not in the `req_nodes` set: This means that while Node j considers Node i in its top `max_reqs` links, Node i has `max_reqs` other links with larger traffic. This request should be denied.

We wait until all requests are received at Node i : This is indicated by a time-out event, that can be set, for example to half the aggregation interval a (requests are timestamped, so requests that arrive after the time-out will simply be ignored.). After all requests are received, there will be at most `max_reqs` candidates for matching. However, the number of free circuits (the optical degree minus the number of matched circuit) may be smaller. Therefore, we choose only the top ones so as not to exceed the number of available links. We thus send a **grant** message to all of them and **deny** messages to others.

In the last phase of our algorithm, each node waits until all its requests are either granted or denied. It then connects with all nodes that (i) *it has granted*, and (ii) *a grant message was received from them*. It disconnects all other links, except those made by the centralized algorithm and above the threshold. Note that rate measurements used in an epoch are performed in parallel with the decision making of the previous epoch.

6 Evaluation

Chopin aims to maximize the circuit throughput (online), without compromising the datacenter latency, by combining centralized and distributed schedulers. Therefore, in our evaluation, we focus on each of these schedulers' parameters as well as on their contribution to the overall DC performance. The performance is evaluated on several parameters, including the centralized scheduler epoch T , aggregation intervals A (for the centralized scheduler) and a (for the distributed scheduler), as well as the corresponding delays Δ and δ .

6.1 Methodology

Topology. We have analyzed Chopin's performance through synthetic simulations, for which we generate traffic according to known datacenter traffic patterns [10, 38, 66]. We used *NetworkX* for topology creation, as well as for matching computations. Our simulation code is available at [67].

Specifically, we have considered real-world datacenter topologies (3-tier) with 8 and 16 aggregation switches, 80 racks and 160 racks, respectively, where each rack contains 10 hosts (i.e., up to 1,600 hosts in the network). In addition to the electrical network, we assume a non-blocking optical circuit switch, which connects to each Chopin node k times at 10 Gbps, we vary the value of k between 1, 2, and 4.

25:12 Chopin: Combining Distributed and Centralized Schedulers

Real datacenter’s data plane parameters were used. The link capacities are 1 Gbps between servers and ToR switches, 10 Gbps between ToRs and aggregation-level switches as in [10], and 40 Gbps between the aggregation-level switches and cores, as in [44]. The reconfiguration time is approximately $11\mu\text{sec}$ [31,63], as discussed in Section 4. As the host’s traffic contains hundreds of Mbps on average at all times [38], we analyzed average host demand levels of 200 Mbps.

Chopin’s evaluation focuses on increasing the optical throughput. Optimizing Chopin’s optical throughput adds some approximation to it, in three aspects: (i) partial maximal weight matching computation, (ii) higher optical degree of Chopin nodes, and (iii) approximated maximal weight matching for higher optical degrees, as discussed next.

Adding several optical routes per Chopin node improves its optical throughput by using higher connectivity between Chopin nodes. This can be achieved, e.g., by a wavelength-selective switch (WSS) module at each ToR switch, which is a customized 1×4 -port Nistica full-fledged 100 WSS module (as suggested in [21]). This implementation enables each Chopin node to connect other Chopin nodes by up to 4 optical links.

This becomes less attractive for a larger number of channels (namely, greater than 4) because of the additional noise (e.g., the multiplexer enables additive noise funneling from each of the sources into the reconfigurable optical add/drop multiplexer ROADM network) [71]. Furthermore, recent studies show that using 1×8 ports increases the system costs by a factor of 10 compared to 1×4 ports [82]. Thus, for cost-effective systems, where several optical switches are recommended, we analyze Chopin’s performance where each Chopin node has up to 4 connections to other Chopin nodes (i.e., “optical degree $k = 4$ ”).

Traffic patterns. We generate the traffic flow based on previous studies of traffic characteristics of datacenter networks [10, 45, 65]. Flows are TCP [1] with Poisson flow arrival times [38], whose size distribution follows one of three well-known flow size distributions: (i) HULL [3]; (ii) pFabric [2, 51, 60]; and (iii) VL2 [32].

The distribution of flow arrival time to the ToR switches is modeled as a Poisson process, where the servers use the network heavily, constantly transmitting and receiving several hundreds of Mbps data on average all the time [38]. Such a traffic pattern matches the common inflow rate in today’s datacenters serving a variety of applications, such as video and job-task managers [41, 52].

The dispersion pattern in the simulation was based on the observation that traffic is either rack-distributed or destined for one $\approx 1\%–10\%$ of the hosts, spread across most of the source’s cluster (tens of racks) [45, 66]. The inter-rack demand per host was set to approximately 150 Mbps, based on [38].

Chopin’s optical circuits throughput is analyzed w.r.t.:

- Different flow traffic distributions (HULL, VL2 and pFabric).
- Different scheduler policies: all-distributed/-centralized, and in-between (varying threshold α levels).

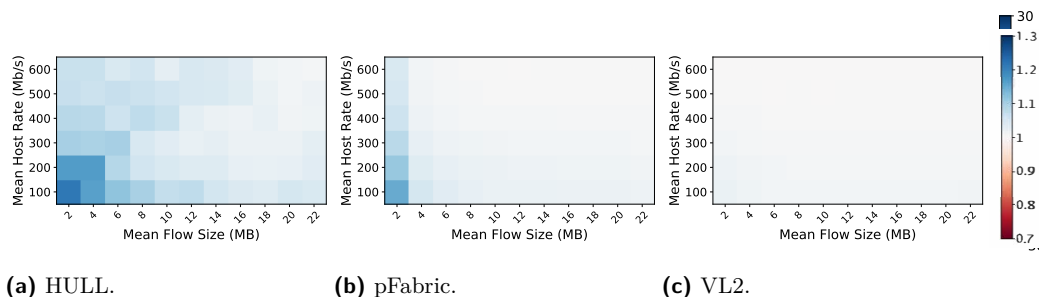
We found that each flow size distribution has special properties, with respect to flow length and flow size variance. These properties have a significant influence on the performance of both the centralized and the distributed scheduler:

- The *HULL* flow distribution is a Pareto distribution where almost all flows are mice⁵ ($< 10\text{KB}$). Moreover, flow variance is low. Therefore, the centralized scheduler throughput is low, as there are not many elephant flows, and the differences between the flow carried by the optical links is small compared to the others.

⁵ We define mice and elephant flows based on the distinction made by [10].

■ **Table 2** Throughput ratio for optical degree 4.

Distribution	HULL	pFabric	VL2
Mean	Low (100 KB)	Medium (1.7MB)	High (12 MB)
Variance	Low	Medium	High
Centralized perf.	0.42	0.7	0.77
Distributed perf.	0.49	0.75	0.77
Chopin	0.5	0.76	0.78



■ **Figure 6** Comparison between *Chopin* and the *centralized* scheduler under different traffic patterns (generated by scaling realistic flow size distributions and Poisson flow arrival times under different rates), when the optical ToR switch’s connectivity is 4. The color represents the ratio between the optical throughput of Chopin and the centralized scheduler. As the blue cells become darker, Chopin more strongly outperforms the centralized scheduler.

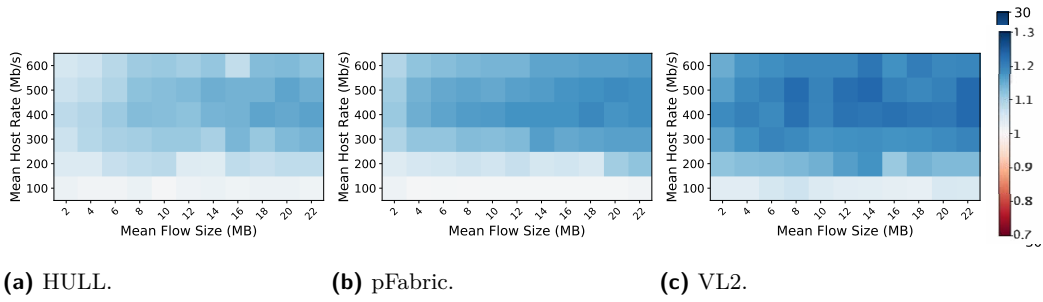
- The *VL2* distribution creates many elephant flows, with high variance. Therefore, the centralized scheduler can optimize the traffic and the distributed scheduler can make decisions which improves the throughput through the optical circuits.
- The *pFabric* distribution includes some elephant flows (but medium mean). With medium variance the distributed scheduler operates as for *VL2*, but centralized is slightly less effective, due to shorter flows.

The properties of these traffic patterns and their impact on the scheduler performance are described in Table 2, for an optical degree of 4. Notice that Hull, as a Pareto distribution with $\alpha = 1.05$, mean=100KB [3] has unbounded variance. pFabric has mean value of approximately 1.7 MB [51] and variance of 3.9MB, and VL2 [32] has mean value of 12 MB with variance of 85MB.

6.2 Scheduler Implementation

The Chopin scheduler consists of centralized scheduler and distributed scheduler. The centralized, described in Section 4, aims to find a Maximum Weight Matching (MWM) solution.

However, due to its complexity, especially as the optical degree (k) increases, the centralized scheduler suffers from large running times. In order to reduce delays, two approximations were introduced. First, MWM with degree k is computed as an iterative Edmond’s MWM algorithm. Second, the centralized scheduler considers only the top- m live flows per ToR switch (instead of all possible pairs between the n nodes). We found top-5 MWM running time to be within 1% of the MWM over all pairs, since MWM complexity (which is the core of our b-matching solution) scales linearly with the number of to-be-matched edges.



■ **Figure 7** Comparison between *Chopin* and the *distributed* scheduler under the same settings as in Fig. 6. The color represents the ratio between the optical throughput of *Chopin* and the distributed scheduler. As the blue cells become darker, *Chopin* more strongly outperforms the distributed scheduler.

Moreover, as each node reports only its top-5 nodes to the controller, the report can be sent by a single 200 bit packet. Considering 100 switches reporting to a controller with 1Gbps network card, and control plane latency of 0.05 ms, all reports can be sent within 0.07 ms. The reconfiguration commands (for at most 4 links per ToR switch) will have similar latency.

Lastly, we consider the actual update time of the switch internal configuration after the reconfiguration message arrives. However, it is considered as negligible, assuming an optimized implementation with time complexity dominated by TCAM update time which is approximately in the 0.025 ms range [43]. Therefore, the total reconfiguration latency based on top-5 nodes can be bounded by 3 ms.

6.3 Scheduler Evaluation Benchmarks

We evaluate *Chopin* with respect to the following centralized and distributed schedulers.

Centralized schedulers are designed for long term datacenter flows. The realistic centralized scheduler was analyzed through different values of the centralized scheduler epoch T , and with delay Δ equals to T . Namely, in the third epoch, the scheduler uses matching results based on data collected in the first epoch, i.e., data from two epochs ago, recall Fig. 3 (characterized both *centralized scheduler* and *Chopin centralized scheduler*). Similarly to Veisllari et al. [77], we consider an *optimal scheduler*, which runs MWM, *with access to future traffic knowledge*. For each 1 ms interval it uses the optimal matching computed as MWM of that interval. Therefore, it is an upper bound for datacenter performance.

We also consider an *online optimal scheduler*, which has no knowledge of the future but it does not suffer from any delay. For each 1 ms interval it uses an allocation computed as the MWM of the previous interval.

Distributed schedulers are designed for bursts and short datacenter flows. According to Roy et al. [66], 90% of the time, 50% of the heavy flows change within 1 ms. Therefore, the distributed scheduler should operate repeatedly in high frequency. *Chopin's distributed scheduler* is set to operate every 1 ms, which is the length of a time-slot in our model. Furthermore, as in the centralized scheduler, both aggregation interval and delay (a and δ respectively) are set to equal the time between two invocations (namely, 1 ms). In addition, the performance of a *distributed scheduler* (unrelated to a centralized scheduler) with the same properties was also analyzed. Moreover, as discussed in §5, a major factor of the *Chopin* distributed scheduler is the threshold α , the level under which the centralized allocation can be changed by the distributed scheduler.

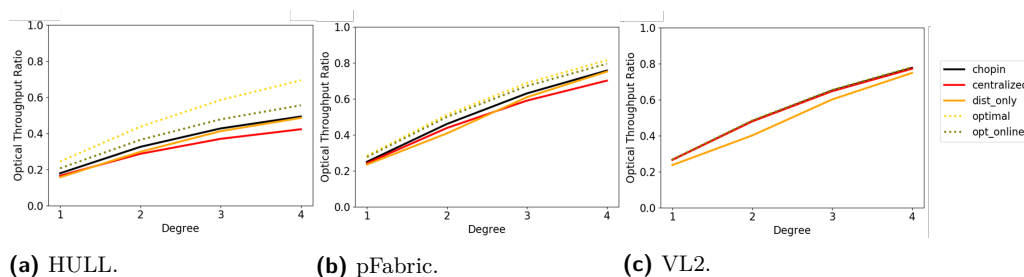


Figure 8 Throughput through the optical circuits, for different optical degrees and flow size distributions.

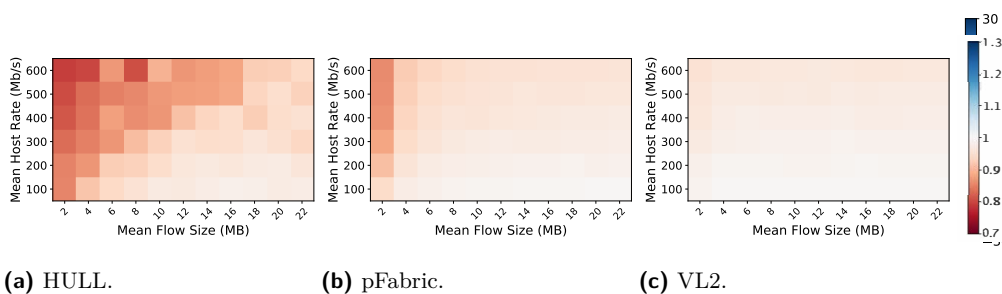


Figure 9 Comparison between *Chopin* and the *online optimal* scheduler under different traffic patterns (each generated by scaling well-known realistic flow size distributions and assuming Poisson flow arrival times under different rates), when the optical ToR switch’s connectivity is 4. As the red cells become darker, the online optimal scheduler performs better than Chopin.

As the threshold decreases, Chopin’s performance is closer to a centralized scheduler. Similarly, as the threshold increases, Chopin’s performance is closer to being distributed. Therefore, we evaluate Chopin for different threshold levels, between 0.1 to 1.3, to capture Chopin’s performance scheduling between distributed and centralized scheduling.

6.4 Centralized-Distributed Trade-off

How can we find an optimal tradeoff between the centralized scheduler, which provides accurate solutions but relies on outdated information, and the distributed scheduler which relies on more recent information but provides approximate solutions (due to locality)?

This trade-off was analyzed in two related ways: (i) optimal threshold, and (ii) optimal reconfiguration number. The threshold α is the parameter which enables the distributed scheduler to change the centralized matching, and therefore, to adapt the traffic changes in small time intervals. For example, a circuit allocation between ToR pair with high throughput on previous intervals, should be torn down if the flow rate reduces drastically. We found that Chopin’s optimal threshold α is between $\approx 0.4 - 0.7$, depending on the traffic pattern. For the HULL traffic pattern, it achieves higher performance with $\alpha = 0.4$, while for DCTCP and VL2 traffic, the optimal threshold is approximately 0.7. Moreover, across this range, the performance across all the traffic patterns were the highest, with low deviation.

6.5 On the Benefit of Hybrid Scheduling

To analyze Chopin’s (the hybrid scheduler) improvement over distributed and centralized schedulers, we consider the optical throughput ratio. Fig. 6 and Fig. 7 describe Chopin’s improvement ratio for each of the flow patterns, compared to centralized and distributed schedulers (respectively), when considering a centralized compute epoch of 3 ms, see §6.2.

The results show that Chopin outperforms the centralized scheduler for *every* traffic pattern. We found that Chopin’s optical throughput ratio is higher than the centralized scheduler optical throughput ratio by up to 20% in the HULL distribution, 15% for pFabric pattern, and 2% for datacenters with a VL2 flow size distribution (as shown in Fig. 6a). Moreover, Chopin also achieves a higher throughput ratio compared to the distributed scheduler across all traffic patterns. Specifically, Chopin increases the optical throughput ratio of the distributed scheduler by up to 16% in the HULL distribution, 20% in the pFabric and 23% in VL2 traffic (see Fig. 7b). Therefore, Chopin outperforms both centralized and distributed schedulers.

6.6 Optical Degree Improvement

Next, we examine the improvement as a function of Chopin nodes’ optical degree. Therefore, we have focused on the optimal threshold for each of the flow patterns, where the centralized scheduler epoch is 3 ms, as discussed in Section 4.

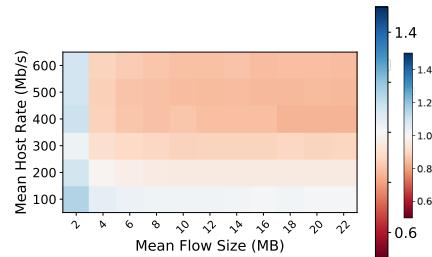
Fig. 8 presents the ratio between the throughput through optical circuits to the overall throughput (electrical and optical networks combined). This *optical throughput ratio* changes with the optical degree and flow patterns, as shown in Fig. 8. In each flow pattern, all the schedulers were considered. It is shown that as the degree increases, the throughput among all the schedulers improved, and that Chopin’s throughput is higher than both the centralized and the distributed schedulers. Moreover, as the number of elephant flows increases (as in VL2), Chopin’s throughput is getting closer to the optimal. It is consistent with Chopin’s aim to carry elephant flows over optical circuits. Therefore, flow patterns with high number of elephant flows benefit more from using Chopin.

6.7 Chopin VS Online Optimal Scheduler

We compared between Chopin performance, and online optimal scheduler performance (where centralized updates are being sent to Chopin nodes every 1 ms instead of 3 ms respectively). Fig. 9 shows that even if Chopin’s centralized scheduler updates were sent every 1 ms (such as in the online optimal scheduler), there is no significant improvement for the VL2 flow pattern (see Fig. 9c). In other words, Chopin is closer to the optimal scheduler as the flows become larger, because as the mean flow is longer, the changes over small time intervals (such as 1 ms) become minor. Therefore, in these cases, the added value of high frequent scheduling updates, even with the “future” information (as in the optimal scheduler), decreases. Chopin can benefit from higher frequent centralized updates mostly in HULL distribution, (where the flows are usually shorter), by approximately 20%.

6.8 Sensitivity Analysis

We further analyzed our results by considering different sizes of networks, e.g. with 100 ToR switches and for 160 ToR switches, where there are 10 hosts per ToR switch, and for different rates per host (100–600 Mbps). Across all the networks that were examined, for each of the traffic patterns, we observe that on certain conditions the distributed scheduler outperforms



■ **Figure 10** Comparison between centralized and distributed schedulers, as in Fig. 1, but with 160 ToR switches. For brevity we only present pFabric results.

the centralized scheduler and vice versa, with respect to higher optical throughput. For instance, see the heatmap for a network with 160 ToR switches, each with an optical degree of 4 connections, under the pFabric traffic pattern in Fig. 10. It shows that under the pFabric distribution, when the mean flow is larger than 5 MB, the centralized scheduler achieves higher performance compared to the distributed one. This phenomenon was also observed for the 80 ToR network, as in Fig. 1b.

Moreover, Chopin’s performance can scale. We demonstrate its effectiveness over a concrete network topology (specified in Section 6.1), but faster links with higher demand will create the same bottleneck and resolve with Chopin in the same way.

7 Conclusion

Chopin aims to combine the benefits of centralized scheduling with distributed scheduling, to provide high throughput and fast reaction. While centralized and distributed scheduling has also been combined in all-static non-hybrid networks, e.g., Facebook’s Express Backbone [42], hybrid networks with optical circuits pose structurally different challenges. In particular, we find that distributed decisions benefit from being closer in time to the measurements they are based on, which is more critical than the rate of decisions.

We believe that our work opens several interesting avenues for future research. In particular, while we achieve significant performance gains, our approach is more complex than the state-of-the-art and it would be useful to simplify it further. Furthermore, our distributed schedulers use the same threshold for all nodes as a homogeneous strategy. While this succinct representation is sufficient for the settings described in this paper, it can be interesting to explore heterogeneity, e.g., to increase the threshold on very congested racks. Finally, the trade-off between an elephant flow’s duration and the time before it starts to route through optical circuits can be considered for future optimization.

References

- 1 Mohammad Alizadeh. Empirical traffic generator. Cisco DC Repositories, 2015.
- 2 Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Suddipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *ACM SIGCOMM*, 2010.
- 3 Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *NSDI*. USENIX Association, 2012.
- 4 Richard P. Anstee. A polynomial algorithm for b-matchings: An alternative approach. *Inf. Process. Lett.*, 24(3):153–157, 1987.

- 5 John Augustine, Kristian Hinnenthal, Fabian Kuhn, Christian Scheideler, and Philipp Schneider. Shortest paths in a hybrid network model. In *SODA*, pages 1280–1299. SIAM, 2020.
- 6 Chen Avin, Manya Ghobadi, Chen Griner, and Stefan Schmid. On the complexity of traffic traces and implications. In *Proc. ACM SIGMETRICS*, 2020.
- 7 Navid Hamed Azimi, Zafar Ayyub Qazi, Himanshu Gupta, Vyas Sekar, Samir R. Das, Jon P. Longtin, Himanshu Shah, and Ashish Tanwer. Firefly: a reconfigurable wireless data center fabric using free-space optics. In *SIGCOMM*. ACM, 2014.
- 8 Hitesh Ballani, Paolo Costa, Raphael Behrendt, Daniel Cletheroe, István Haller, Krzysztof Jozwik, Fotini Karinou, Sophie Lange, Kai Shi, Benn Thomsen, and Hugh Williams. Sirius: A flat datacenter network with nanosecond optical switching. In *SIGCOMM*, pages 782–797. ACM, 2020.
- 9 Alkida Balliu, Sebastian Brandt, Juho Hirvonen, Dennis Olivetti, Mikaël Rabie, and Jukka Suomela. Lower bounds for maximal matchings and maximal independent sets. *J. ACM*, 68(5):39:1–39:30, 2021.
- 10 T. Benson, A. Akella, and D.A. Maltz. Network traffic characteristics of data centers in the wild. In *ACM IMC*, pages 267–280, 2010.
- 11 André Berger, James Gross, Tobias Harks, and Simon Tenbusch. Constrained resource assignments: Fast algorithms and applications in wireless networks. *Management Science*, 62, November 2015.
- 12 Maciej Besta and Torsten Hoefler. Slim fly: A cost effective low-diameter network topology. In *IEEE SC*, pages 348–359, 2014.
- 13 Sayan Bhattacharya, Deeparnab Chakrabarty, and Monika Henzinger. Deterministic dynamic matching in $O(1)$ update time. *Algorithmica*, 82(4):1057–1080, 2020.
- 14 Li Chen, Kai Chen, Zhonghua Zhu, Minlan Yu, George Porter, Chunming Qiao, and Shan Zhong. Enabling wide-spread communications on optical fabric with megaswitch. In *USENIX NDSI*, 2017.
- 15 Charles Clos. A study of non-blocking switching network. *Bell System Technology Journal*, 32(2):406–424, 1953.
- 16 Shibsankar Das. A modified decomposition algorithm for maximum weight bipartite matching and its experimental evaluation. *Sci. Ann. Comput. Sci.*, 30(1):39–67, 2020.
- 17 Sushovan Das, Afsaneh Rahbar, Xinyu Crystal Wu, Zhuang Wang, Weitao Wang, Ang Chen, and T. S. Eugene Ng. Shufflecast: An optical, data-rate agnostic, and low-power multicast architecture for next-generation compute clusters. *IEEE/ACM Trans. Netw.*, 30(5):1970–1985, 2022.
- 18 Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *USENIX ATC*, 2015.
- 19 Nikhil Devanur, Janardhan Kulkarni, Gireeja Ranade, Manya Ghobadi, Ratul Mahajan, and Amar Phanishayee. Stable matching algorithm for an agile reconfigurable data center interconnect. Technical Report 2016-1140, MSR, June 2016.
- 20 Fahad Dogar, Thomas Karagiannis, Hitesh Ballani, and Antony Rowstron. Decentralized task-aware scheduling for data center networks. *ACM SIGCOMM CCR*, 44, August 2014.
- 21 N. Farrington, A. Forencich, G. Porter, P. C. Sun, J. E. Ford, Y. Fainman, G. C. Papen, and A. Vahdat. A multiport microsecond optical circuit switch for data center networking. *IEEE Phot. Techn. L.*, 25(16):1589–92, August 2013.
- 22 Nathan Farrington, Alex Forencich, Pang-Chen Sun, Shaya Fainman, Joe Ford, Amin Vahdat, George Porter, and George C. Papen. A 10 us hybrid optical-circuit/electrical-packet network for datacenters. In *OFC/NFOEC*. OSA, 2013.
- 23 Nathan Farrington, George Porter, Yeshiaihu Fainman, George Papen, and Amin Vahdat. Hunting mice with microsecond circuit switches. In *ACM HotNets*, 2012.

- 24 Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Hajabdolali Bazzaz, Vikram Subramanya, Yashaiah Fainman, George Papen, and Amin Vahdat. Helios: a hybrid electrical/optical switch architecture for modular data centers. In *SIGCOMM*. ACM, 2010.
- 25 Thomas Fenz, Klaus-Tycho Foerster, Stefan Schmid, and Anaïs Villedieu. Efficient non-segregated routing for reconfigurable demand-aware networks. *Comput. Commun.*, 164:138–147, 2020.
- 26 Klaus-Tycho Foerster, Maciej Pacut, and Stefan Schmid. On the complexity of non-segregated routing in reconfigurable data center architectures. *Comput. Commun. Rev.*, 49(2):2–8, 2019.
- 27 Klaus-Tycho Foerster and Stefan Schmid. Survey of reconfigurable data center networks: Enablers, algorithms, complexity. *SIGACT News*, 50(2):62–79, 2019.
- 28 Harold N. Gabow. Data structures for weighted matching and extensions to b -matching and f -factors. *ACM Trans. Algorithms*, 14(3):39:1–39:80, 2018.
- 29 Zvi Galil. Efficient algorithms for finding maximum matching in graphs. *ACM Comput. Surv.*, 18(1):23–38, March 1986.
- 30 Manya Ghobadi, Ratul Mahajan, Amar Phanishayee, Pierre-Alexandre Blanche, Houman Rastegarfar, Madeleine Glick, and Daniel Kilper. Design of mirror assembly for an agile reconfigurable data center interconnect. Technical Report 2016-1139, MSR, June 2016.
- 31 Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Nikhil Devanur, Janardhan Kulkarni, Gireeja Ranade, Pierre-Alexandre Blanche, Houman Rastegarfar, Madeleine Glick, and Daniel Kilper. Projector: Agile reconfigurable data center interconnect. In *ACM SIGCOMM*, pages 216–229, 2016.
- 32 A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A scalable and flexible data center network. *ACM SIGCOMM*, 39(4):51–62, 2009.
- 33 A. Grieco, G. Porter, and Y. Fainman. Integrated space-division multiplexer for application to data center networks. *IEEE J. Sel. Top. Quant. El.*, 22(6), 2016.
- 34 Chen Griner, Stefan Schmid, and Chen Avin. Cachenet: Leveraging the principle of locality in reconfigurable network design. *Computer Networks*, 204:108648, 2022.
- 35 Chen Griner, Johannes Zerwas, Andreas Blenk, Manya Ghobadi, Stefan Schmid, and Chen Avin. Cerberus: The power of choices in datacenter topology design - A throughput perspective. *Proc. ACM Meas. Anal. Comput. Syst.*, 5(3):38:1–38:33, 2021.
- 36 Matthew Nance Hall, Klaus-Tycho Foerster, Stefan Schmid, and Ramakrishnan Durairajan. A survey of reconfigurable optical networks. *Opt. Switch. Netw.*, 41:100621, 2021.
- 37 Daniel Halperin, Srikanth Kandula, Jitendra Padhye, Paramvir Bahl, and David Wetherall. Augmenting data center networks with multi-gigabit wireless links. In *SIGCOMM*, pages 38–49. ACM, 2011.
- 38 Y. Han, J.H. Yoo, and J.W.K. Hong. Poisson shot-noise process based flow-level traffic matrix generation for data center networks. In *IFIP/IEEE IM*, May 2015.
- 39 Kathrin Hanauer, Monika Henzinger, Stefan Schmid, and Jonathan Trummer. Fast and heavy disjoint weighted matchings for demand-aware datacenter topologies. In *INFOCOM*, pages 1649–1658. IEEE, 2022.
- 40 Keqiang He, Junaid Khalid, Aaron Gember-Jacobson, Sourav Das, Chaithan Prakash, Aditya Akella, Li Erran Li, and Marina Thottan. Measuring control plane latency in sdn-enabled switches. In *ACM SIGCOMM, SOSR '15*, pages 25:1–25:6, 2015.
- 41 Netflix help center. Internet connection speed recommendations, 2018. URL: <https://help.netflix.com/en/node/306>.
- 42 Mikel Jimenez and Henry Kwik. Building Express Backbone: Facebook’s new long-haul network, May 2017. URL: <https://engineering.fb.com/data-center-engineering/building-express-backbone-facebook-s-new-long-haul-network/>.
- 43 Mikel Jimenez and Henry Kwik. Ternary Content Addressable Memory (TCAM) Search IP for SDNet - SmartCORE IP Product Guide. Technical report, Xilinx, November 2017. URL: https://www.xilinx.com/support/documentation/ip_documentation/tcam/pg190-tcam.pdf.

- 44 S. Kandula, J. Padhye, and P. Bahl. Flyways to de-congest data center networks. In *ACM HotNets*, 2009.
- 45 S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of data center traffic: Measurements & analysis. In *ACM IMC*, pages 202–208, 2009.
- 46 Simon Kassing, Asaf Valadarsky, Gal Shahaf, Michael Schapira, and Ankit Singla. Beyond fat-trees without antennae, mirrors, and disco-balls. In *SIGCOMM*, pages 281–294. ACM, 2017.
- 47 Arif M. Khan, Alex Pothan, Md. Mostofa Ali Patwary, Nadathur Rajagopalan Satish, Narayanan Sundaram, Fredrik Manne, Mahantesh Halappanavar, and Pradeep Dubey. Efficient approximation algorithms for weighted b-matching. *SIAM J. Sci. Comput.*, 38(5), 2016.
- 48 Viatcheslav Korenwein. The practical power of data reduction for maximum-cardinality matching. Masterthesis, TU Berlin, January 2018. Master thesis. URL: <http://fpt.akt.tu-berlin.de/publications/theses/ma-viatcheslav-korenwein.pdf>.
- 49 M. Kuzniar, P. Peresini, and D. Kostic. What you need to know about sdn control and data planes. Technical report, EPFL, 2014.
- 50 Adam N. Letchford, Gerhard Reinelt, and Dirk Oliver Theis. Odd minimum cut sets and b-matchings revisited. *SIAM J. Discret. Math.*, 22(4):1480–1487, 2008.
- 51 Z. Li, W. Bai, K. Chen, D. Han, Y. Zhang, D. Li, and H. Yu. Rate-aware flow scheduling for commodity data center networks. In *IEEE INFOCOM*, pages 1–9, 2017. doi:10.1109/INFOCOM.2017.8057082.
- 52 Xiao Ling, Yi Yuan, Dan Wang, Jiangchuan Liu, and Jiahai Yang. Joint scheduling of mapreduce jobs with servers. *J. Parallel Distrib. Comput.*, 90(C):52–66, April 2016.
- 53 He Liu, Feng Lu, Alex Forencich, Rishi Kapoor, Malveeka Tewari, Geoffrey M. Voelker, George Papen, Alex C. Snoeren, and George Porter. Circuit switching under the radar with reactor. In *USENIX NSDI*, pages 1–15, April 2014.
- 54 He Liu, Feng Lu, Alex Forencich, Rishi Kapoor, Malveeka Tewari, Geoffrey M. Voelker, George Papen, Alex C. Snoeren, and George Porter. Circuit switching under the radar with reactor. In *USENIX NSDI*, pages 1–15, 2014.
- 55 Zvi Lotker, Boaz Patt-Shamir, and Seth Pettie. Improved distributed approximate matching. *J. ACM*, 62(5):38:1–38:17, 2015.
- 56 Zvi Lotker, Boaz Patt-Shamir, and Adi Rosén. Distributed approximate matching. *SIAM J. Comput.*, 39(2):445–460, 2009.
- 57 Long Luo, Klaus-Tycho Foerster, Stefan Schmid, and Hongfang Yu. Optimizing multicast flows in high-bandwidth reconfigurable datacenter networks. *J. Netw. Comput. Appl.*, 203:103399, 2022.
- 58 William M. Mellette, Rajdeep Das, Yibo Guo, Rob McGuinness, Alex C. Snoeren, and George Porter. Expanding across time to deliver bandwidth efficiency and low latency . In *NSDI*. USENIX Association, 2020.
- 59 William M. Mellette, Rob McGuinness, Arjun Roy, Alex Forencich, George Papen, Alex C. Snoeren, and George Porter. Rotornet: A scalable, low-complexity, optical datacenter network. In *SIGCOMM*. ACM, 2017.
- 60 Alizadeh Mohammad, Yang Shuang, Sharif Milad, Katti Sachin, McKeown Nick, Prabhakar Balaji, and Shenker Scott. pfabric: Minimal near-optimal datacenter transport. *ACM SIGCOMM*, 43(4):435–446, 2013.
- 61 Matthias Müller-Hannemann and Alexander Schwartz. Implementing weighted b-matching algorithms: Insights from a computational study. *ACM Journal of Experimental Algorithmics*, 5:8, 2000.
- 62 George Porter, Richard Strong, Nathan Farrington, Alex Forencich, Pang Chen-Sun, Tajana Rosing, Yeshaiah Fainman, George Papen, and Amin Vahdat. Integrating microsecond circuit switching into the data center. *ACM SIGCOMM*, 43(4):447–458, 2013.

- 63 George Porter, Richard D. Strong, Nathan Farrington, Alex Forencich, Pang-Chen Sun, Tajana Rosing, Yeshaiah Fainman, George Papen, and Amin Vahdat. Integrating microsecond circuit switching into the data center. In *SIGCOMM*, pages 447–458. ACM, 2013.
- 64 Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Muhammad Mukarram Bin Tariq, Rui Wang, Jianan Zhang, Virginia Beaugard, Patrick Conner, Steve D. Gribble, Rishi Kapoor, Stephen Kratzer, Nanfang Li, Hong Liu, Karthik Nagaraj, Jason Ornstein, Samir Sawhney, Ryohei Urata, Lorenzo Vicisano, Kevin Yasumura, Shidong Zhang, Junlan Zhou, and Amin Vahdat. Jupiter evolving: transforming google’s datacenter network via optical circuit switches and software-defined networking. In *SIGCOMM*, pages 66–85. ACM, 2022.
- 65 Y. Qiao, Z. Hu, and J. Luo. Efficient traffic matrix estimation for data center networks. In *IFIP Networking*, pages 1–9, May 2013.
- 66 Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network’s (datacenter) network. In *SIGCOMM*. ACM, 2015.
- 67 Neta Rozen-Schiff, David Hay, Stefan Schmid, and Klaus-Tycho Foerster. Chopin implementation code. https://bitbucket.org/NetaRS/sched_analytics/src/master/, October 2020.
- 68 Stefan Schmid, Chen Avin, Christian Scheideler, Michael Borokhovich, Bernhard Haeupler, and Zvi Lotker. Splaynet: Towards locally self-adjusting networks. *IEEE/ACM Trans. Netw.*, 24(3):1421–1433, 2016.
- 69 Ankit Singla, Chi-Yao Hong, Lucian Popa, and Philip Brighten Godfrey. Jellyfish: Networking data centers, randomly. In *USENIX NSDI*, volume 12, 2012.
- 70 Ankit Singla, Atul Singh, and Yan Chen. OSA: An optical switching architecture for data center networks with unprecedented flexibility. In *USENIX NSDI*, 2012.
- 71 T. A. Strasser and J. L. Wagener. Wavelength-selective switches for roadm applications. *IEEE J. Sel. Top. Quant. El.*, 16(5), 2010.
- 72 Xiaoye Steven Sun and T. S. Eugene Ng. When creek meets river: Exploiting high-bandwidth circuit switch in scheduling multicast data. In *ICNP*, pages 1–6. IEEE Computer Society, 2017.
- 73 Jukka Suomela. Survey of local algorithms. *ACM Comput. Surv.*, 45(2):24:1–24:40, 2013.
- 74 Akhilesh S. Thyagaturu, Anu Mercian, Michael P. McGarry, Martin Reisslein, and Wolfgang Kellerer. Software defined optical networks (sdons): A comprehensive survey. *IEEE Commun. Surv. Tutorials*, 18(4):2738–2786, 2016.
- 75 Gerard J Tortora and Bryan H Derrickson. *Principles of anatomy and physiology*. John Wiley & Sons, 2018.
- 76 Asaf Valadarsky, Gal Shahaf, Michael Dinitz, and Michael Schapira. Xpander: Towards optimal-performance datacenters. In *ACM CoNEXT*, 2016.
- 77 R. Veislari, S. Bjornstad, and N. Stol. Scheduling techniques in an integrated hybrid node with electronic buffers. In *ONDM*, April 2012.
- 78 Shaileshh Bojja Venkatakrishnan, Mohammad Alizadeh, and Pramod Viswanath. Costly circuits, submodular schedules and approximate carathéodory theorems. In *SIGMETRICS*. ACM, 2016.
- 79 Guohui Wang, David G. Andersen, Michael Kaminsky, Konstantina Papagiannaki, T. S. Eugene Ng, Michael Kozuch, and Michael P. Ryan. c-through: part-time optics in data centers. In *SIGCOMM*, pages 327–338. ACM, 2010.
- 80 Yiting Xia, Xiaoye Steven Sun, Simbarashe Dzinamarira, Dingming Wu, Xin Sunny Huang, and T. S. Eugene Ng. A tale of two topologies: Exploring convertible data center network architectures with flat-tree. In *SIGCOMM*, pages 295–308. ACM, 2017.
- 81 Bing Xiong, Kun Yang, Jinyuan Zhao, Wei Li, and Keqin Li. Performance evaluation of openflow-based software-defined networks based on queueing model. *Comput. Netw.*, 102(C):172–185, 2016.

25:22 Chopin: Combining Distributed and Centralized Schedulers

- 82 Haining Yang, Brian Robertson, Peter Wilkinson, and Daping Chu. Low-cost cdc roadm architecture based on stacked wavelength selective switches. *J. Opt. Commun. Netw.*, 9(5):375–384, May 2017.
- 83 Johannes Zerwas, Chen Avin, Stefan Schmid, and Andreas Blenk. Exec: Experimental framework for reconfigurable networks based on off-the-shelf hardware. In *ANCS*, pages 66–72. ACM, 2021.
- 84 Johannes Zerwas, Wolfgang Kellerer, and Andreas Blenk. What you need to know about optical circuit reconfigurations in datacenter networks. In *ITC*, pages 1–9. IEEE, 2021.
- 85 Danyang Zhuo, Qiao Zhang, Vincent Liu, Arvind Krishnamurthy, and Thomas Anderson. Rack-level congestion control. In *ACM HotNets*, 2016.

A Chopin's Distributed Scheduler Algorithm

In Algorithm 1, we provide the pseudo-code of Chopin's distributed algorithm.

Algorithm 1 Chopin Distributed Algorithm Code for Node i .

max_reqs : The number of allowed requests per ToR switch
 cur_nodes : The nodes currently connected with i
 centralized_nodes : The nodes matched to i by the centralized scheduler in its last invocation
 $\text{received_reqs} \leftarrow \emptyset$

Upon the beginning of a distributed scheduler epoch:

- 1: **function** START:
- 2: $\text{matched_nodes} \leftarrow \emptyset$
- 3: **for** $p \in (\text{cur_nodes} \cap \text{centralized_nodes})$ **do**
- 4: **if** $r_{i,p} \geq \alpha \cdot R_{i,p}$ **then**
- 5: $\text{matched_nodes.add}(p)$
- 6: $\text{req_nodes} \leftarrow ([n] \setminus \{i\}) \setminus \text{matched_nodes}$ $\triangleright n$ denotes the number of Chopin nodes in the network
- 7: $\text{req_nodes} \leftarrow \text{GET_TOP_NODES}(\text{req_nodes}, \text{max_reqs})$ \triangleright Top max_reqs nodes, out of req_nodes , with the most bi-directional traffic with ToR switch i .
- 8: $\text{grants} \leftarrow \emptyset$; $\text{denies} \leftarrow \emptyset$
- 9: $\text{SEND_REQUESTS}(\text{req_nodes})$ \triangleright Send **request** to all nodes in req_nodes .

Upon receiving a **request** message from src_id :

- 10: **function** REQUEST_HANDLER(src_id):
- 11: $\text{received_reqs.add}(\text{src_id})$

Upon a timeout event (implying the request phase has ended):

- 12: **function** REQUEST_TIMEOUT_HANDLER:
- 13: $\text{nodes} \leftarrow \text{req_nodes} \cap \text{received_reqs}$
- 14: $\text{free_links} \leftarrow k - |\text{matched_nodes}|$
- 15: $\text{granted} \leftarrow \text{GET_TOP_NODES}(\text{nodes}, \text{free_links})$
- 16: $\text{rejected} \leftarrow \text{received_reqs} \setminus \text{granted}$
- 17: $\text{SEND_DENIES}(\text{rejected})$ \triangleright Send **deny** message to all nodes in rejected set.
- 18: $\text{SEND_GRANTS}(\text{granted})$ \triangleright Send **grant** message to all nodes in granted set.
- 19: $\text{grant_sent} \leftarrow \text{true}$
- 20: $\text{TRY_EXECUTE_DECISIONS}()$

Upon receiving a **grant** message from src_id :

- 21: **function** GRANT_HANDLER(src_id):
- 22: $\text{grants.add}(\text{src_id})$
- 23: $\text{TRY_EXECUTE_DECISIONS}()$

Upon receiving a **deny** message from src_id :

- 24: **function** DENY_HANDLER(src_id):
- 25: $\text{denies.add}(\text{src_id})$
- 26: $\text{TRY_EXECUTE_DECISIONS}()$
- 27: **function** TRY_EXECUTE_DECISIONS:
- 28: **if** $\text{denies} \cup \text{grants} \neq \text{req_nodes}$ **or not** grant_sent **then**
- 29: **return** \triangleright Not all grant/deny were received
- 30: $\text{new_nodes} \leftarrow \text{granted} \cap \text{grants}$
- 31: **for** $p \in (\text{cur_nodes} \setminus \text{new_nodes}) \setminus \text{matched_nodes}$ **do**
- 32: $\text{DISCONNECT}(p)$
- 33: **for** $p \in (\text{new_nodes} \setminus \text{cur_nodes} \setminus \text{matched_nodes})$ **do**
- 34: $\text{CONNECT}(p)$
- 35: $\text{received_reqs} \leftarrow \emptyset$; $\text{grant_sent} \leftarrow \text{false}$

A Modular Approach to Construct Signature-Free BRB Algorithms Under a Message Adversary

Timothé Albouy ✉

Univ Rennes, Inria, CNRS, IRISA, France

Davide Frey ✉

Univ Rennes, Inria, CNRS, IRISA, France

Michel Raynal ✉

Univ Rennes, Inria, CNRS, IRISA, France

François Taïani ✉

Univ Rennes, Inria, CNRS, IRISA, France

Abstract

This paper explores how reliable broadcast can be implemented without signatures when facing a dual adversary that can both corrupt processes and remove messages. More precisely, we consider an asynchronous n -process message-passing system in which up to t processes are Byzantine and where, at the network level, for each message broadcast by a correct process, an adversary can prevent up to d processes from receiving it (the integer d defines the power of the message adversary). So, unlike previous works, this work considers that not only can computing entities be faulty (Byzantine processes), but, in addition, that the network can also lose messages. To this end, the paper adopts a modular strategy and first introduces a new basic communication abstraction denoted $k2\ell$ -cast, which simplifies quorum engineering, and studies its properties in this new adversarial context. Then, the paper deconstructs existing signature-free Byzantine-tolerant asynchronous broadcast algorithms and, with the help of the $k2\ell$ -cast communication abstraction, reconstructs versions of them that tolerate both Byzantine processes and message adversaries. Interestingly, these reconstructed algorithms are also more efficient than the Byzantine-tolerant-only algorithms from which they originate.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Asynchronous system, Byzantine processes, Communication abstraction, Delivery predicate, Fault-Tolerance, Forwarding predicate, Message adversary, Message loss, Modularity, Quorum, Reliable broadcast, Signature-free algorithm, Two-phase commit

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2022.26

Related Version *Full Version:* <https://arxiv.org/abs/2204.13388>

Funding This work has been partially supported by the French ANR projects ByBloS (ANR-20-CE25-0002-01) and PriCLESS (ANR-10-LABX-07-81) devoted to the design of modular distributed computing building blocks.

1 Introduction

Context: reliable broadcast and message adversaries. Reliable broadcast (RB) is a fundamental abstraction in distributed computing that lies at the core of many higher-level constructions (including distributed memories, distributed agreement, and state machine replication). Essentially, RB requires that non-faulty (i.e., correct) processes agree on the set of messages they deliver so that this set includes at least all the messages that correct processes have broadcast.



© Timothé Albouy, Davide Frey, Michel Raynal, and François Taïani;
licensed under Creative Commons License CC-BY 4.0

26th International Conference on Principles of Distributed Systems (OPODIS 2022).

Editors: Eshcar Hillel, Roberto Palmieri, and Etienne Rivière; Article No. 26; pp. 26:1–26:23

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In a failure-free system, implementing reliable broadcast on top of an asynchronous network is relatively straightforward [27]. If processes may fail, and in particular if failed processes may behave arbitrarily (a failure known as Byzantine [21, 26]), implementing reliable broadcast becomes far from trivial as Byzantine processes may collude to fool correct processes [29]. An algorithm that solves reliable broadcast in the presence of Byzantine processes is known as implementing BRB (Byzantine reliable broadcast).

BRB in asynchronous networks (in which no bound is known over message delays) has been extensively studied over the last forty years [1, 2, 7, 11, 12, 18, 20, 23, 22, 25, 29]. Existing BRB algorithms typically assume they execute over a *reliable* point-to-point network, i.e., a network in which sent messages are eventually received. This is a reasonable assumption as most unreliable networks can be made reliable using re-transmissions and acknowledgments (e.g. a timeout-free version of the TCP protocol).

This work takes a drastic turn away from this usual assumption and explores how BRB might be provided when processes execute on an *unreliable* network that might lose point-to-point messages. Our motivation is threefold: First, in volatile networks (e.g., mobile networks or networks under attack), processes might remain disconnected over long periods (e.g., weeks or months), leading in practice to considerable delays (a.k.a. tail latencies) when using re-transmissions. Because most asynchronous Byzantine-tolerant algorithms exploit intersecting quorums, these tail latencies can potentially limit the performance of BRB algorithms drastically, a well-known phenomenon in systems research [14, 15, 34]. Second, re-transmissions require that correct processes be eventually able to receive messages and cannot, therefore, model the permanent disconnection of correct processes. Finally, this question is interesting in its own right, as it opens up novel trade-offs between algorithm guarantees and network requirements, with potential application to the design of reactive distributed algorithms tolerant to both processes and network failures.

The impact of network faults on distributed algorithms has been studied in several works, in particular using the concept of message adversaries (MA). Message adversaries were initially introduced by N. Santoro and P. Widmayer in [31, 32]¹, and then used (sometimes implicitly) in many works (e.g., [4, 3, 13, 17, 30, 28, 32, 33]). Initially proposed for synchronous networks, an MA may suppress point-to-point network messages according to rules that define its power. For instance, a tree MA in a synchronous network might suppress any message except those transiting on an (unknown) spanning tree of the network, with this spanning tree possibly changing in each round.

The message losses that an MA causes differ fundamentally from Byzantine faults. This is because an MA can affect the messages sent by any correct process, and can change the processes it targets during an execution, in contrast to Byzantine corruptions that are tied to a set of fixed processes (which is why MA faults are sometimes dubbed *transient* or *mobile*). For instance, it may be tempting to think that Byzantine-fault-tolerant (BFT) algorithms inherently tolerate message losses from correct processes because they can only afford to wait for at most $n - t$ messages (where n is the total number of processes, and t the upper bound on Byzantine processes). In an asynchronous network, a BFT algorithm could therefore miss up to t messages from correct processes, if those are delayed by the scheduler. This scenario only applies, however, in the particular circumstance where the t faulty processes

¹ Where the terminology *communication failure model* and *ubiquitous faults* is used instead of MA. While we consider only message losses, the work of Santoro and Widmayer also considers message additions and corruptions.

send messages that are received and accepted as valid by correct recipients. This caveat is fundamental. If the faulty processes remain silent or send contradicting messages (if they are Byzantine), then a BFT algorithm cannot afford to lose t messages from correct processes.

Content of the paper. This paper combines a Message Adversary with Byzantine processes, and studies the signature-free implementation of Byzantine Reliable Broadcast (BRB) in an asynchronous, fully connected network subject to this MA and to at most t Byzantine faults. The MA models lossy connections by preventing up to d point-to-point messages from reaching their recipient every time a correct process seeks to communicate with the rest of the network.²

To limit as much as possible our working assumptions, we further assume that the computability power of the adversary is unbounded (except for the cryptography-based algorithm presented in Section 6), which precludes the use of signatures. (We do assume, however, that each point-to-point communication channel is authenticated.)³

This represents a particularly challenging environment, as the MA may target different correct processes every time the network is used or focus indefinitely on the same (correct) victims. Further, the Byzantine processes may collude with the MA for maximal impact.

For clarity, in the remainder of the paper, we simply call *messages* the point-to-point network messages used internally by a BRB algorithm. (The MA may suppress these messages.) We distinguish these messages from the messages the BRB algorithm seeks to disseminate, which we call “*application messages*” (*app-messages* for short). In such a context, the paper presents the following results.

- It first introduces a new modular abstraction, named $k2\ell$ -cast, which appears to be a foundational building block to implement BRB abstractions (with or without the presence of an MA). This communication abstraction systematically dissociates the predicate used to forward (network) messages from the predicate that triggers the delivery of an app-message, and lies at the heart of the work presented in the paper. When proving the $k2\ell$ -cast communication abstraction, the paper presents an in-depth analysis of the power of an adversary that controls at most t Byzantine processes and an MA of power d .
- Then, the paper deconstructs two signature-free BRB algorithms (Bracha’s [11] and Imbs and Raynal’s [20] algorithms) and reconstructs versions of them that tolerate *both* Byzantine processes and MA. Interestingly, when considering Byzantine failures only, these deconstructed versions use smaller quorum sizes and are, therefore, more efficient than their initial counterparts.

So, this paper is not only the first to present signature-free BRB algorithms in the context of asynchrony and MA but also the first to propose an intermediary communication abstraction that allows us to obtain efficient BRB algorithms. For clarity, we give in Table 1 the list of acronyms and notations used in this paper.

Roadmap. The paper is composed of 7 sections and one appendix. Section 2 describes the underlying computing model. Section 3 presents the $k2\ell$ -cast abstraction and its properties. Section 4 defines the MA-tolerant BRB communication abstraction. Section 5 shows that

² A close but different notion was introduced by Dolev in [16], which considers static κ -connected networks. If the adversary selects statically, for each correct sender, d correct processes that do not receive any of this sender’s messages, the proposed model includes Dolev’s model where $\kappa = n - d$.

³ Let us mention that the problem of designing an MA-tolerant BRB has been solved in [4] by leveraging digital signatures within a monolithic algorithm. Finding a signature-free counterpart remained, however, an open question, which we answer positively in this paper using a modular strategy.

■ **Table 1** Acronyms and notations.

Acronyms	Meaning
BRB	Byzantine-tolerant reliable broadcast
MA	Message adversary
MBRB	Message adversary- and Byzantine-tolerant reliable broadcast
Notations	Meaning
n	total nb of processes in the network
t	upper bound on the nb of Byzantine processes
d	power of the message adversary
c	effective nb of correct processes in a run ($n - t \leq c \leq n$)
k	minimal nb of correct processes that $k2\ell$ -cast a message
ℓ	minimal nb of correct processes that $k2\ell$ -deliver a message
k'	minimal nb of correct $k2\ell$ -casts if there is a correct $k2\ell$ -delivery
δ	true iff no-duplicity is guaranteed, false otherwise
q_d	size of the $k2\ell$ -delivery quorum
q_f	size of the forwarding quorum
<i>single</i>	true iff only a single message can be endorsed, false otherwise

thanks to the $k2\ell$ -cast abstraction, existing BRB algorithms can give rise to MA-tolerant BRB algorithms which, when $d = 0$, are more efficient than the BRB algorithms they originate from. Section 6 presents a signature-based implementation of $k2\ell$ -cast that possesses optimal guarantees. Finally, Section 7 concludes the paper. Due to page limitations, some proofs and a numerical evaluation of the $k2\ell$ -cast abstraction are presented in appendices of this paper and its extended version [5].

2 Computing Model

Process model. The system is composed of n asynchronous sequential processes denoted p_1, \dots, p_n . Each process p_i has a distinct identity, known to other processes. For simplicity and without loss of generality, we assume that i is the identity of p_i .

In terms of faults, up to $t \geq 0$ processes can be Byzantine, where a Byzantine process is a process whose behavior does not follow the code specified by its algorithm [21, 26]. Byzantine processes may collude to fool non-Byzantine processes (also called correct processes). In this model, the premature stop (crash) of a process is a Byzantine failure. In the following, given an execution, c denotes the effective number of processes that behave correctly in that execution. We always have $n - t \leq c \leq n$. While this number remains unknown to correct processes, it is used to analyze and characterize (more precisely than using its worse value $n - t$) the guarantees provided by the proposed algorithms.

Finally, the processes have no access to random numbers, and their computability power is unbounded. Hence, the algorithms presented in the paper are deterministic and signature-free (except the signature-based algorithm presented in Section 6).

Communication model. Processes communicate by exchanging messages through a fully connected asynchronous point-to-point network, assumed to be reliable in the sense it neither corrupts, duplicates, nor creates messages. As far as messages losses are concerned, the network is under the control of an adversary (see below) that can suppress messages.

Let MSG be a message type and v the associated value. A process can invoke the best-effort broadcast macro-operation denoted `ur_broadcast(MSG(v))`, which is a shorthand for “**for all** $j \in \{1, \dots, n\}$ **do send** MSG(v) **to** p_j **end for**”. Correct processes are assumed to invoke

`ur_broadcast` to send messages. When they do, we say that the messages are *ur-broadcast* and *received*. The operation `ur_broadcast(MSG(v))` is not reliable. For example, if the invoking process crashes during its invocation, an arbitrary subset of processes receive the message `MSG(v)`. Moreover, due to its very nature, a Byzantine process can send fake messages without using the macro-operation `ur_broadcast`.

Message adversary. Let d be an integer constant such that $0 \leq d < n - t$. The communication network is controlled by an MA (as defined in Section 1), which eliminates messages `ur-broadcast` by correct processes, so these messages are lost. More precisely, when a correct process invokes `ur_broadcast(MSG(v))`, the MA is allowed to arbitrarily suppress up to d copies of the message `MSG(v)` intended to correct processes⁴. This means that, although the sender is correct, up to d correct processes may miss the message `MSG(v)`. The extreme case $d = 0$ corresponds to the case where no message is lost.

As an example, let us consider a set D of correct processes, where $1 \leq |D| \leq d$, such that during some period of time, the MA suppresses all the messages sent to them. It follows that, during this period of time, this set of processes appears as being input-disconnected from the other correct processes. Note that the size and the content of D can vary with time and are never known by the correct processes.

3 $k2\ell$ -Cast Abstraction

Signature-free BRB algorithms [9, 11, 20] often rely on successive waves of internal messages (e.g. the ECHO or READY messages of Bracha’s algorithm [11]) to provide safety and liveness. Each wave is characterized by a threshold-based predicate that triggers the algorithm’s next phase when fulfilled (e.g. enough ECHO messages for the same app-message m).

In this section, we introduce, implement, and prove a new modular abstraction, called $k2\ell$ -cast, that encapsulates a wave/thresholding mechanism that is both Byzantine- and MA-tolerant. As previously announced, we then use this abstraction to reconstruct MA-tolerant BRB algorithms in Section 5 from two existing BRB algorithms [11, 20].

3.1 Definition

$k2\ell$ -cast (for k -to- ℓ -cast) is a many-to-many communication abstraction⁵. Intuitively, it relates the number k of correct processes that send a message m (we say that these processes $k2\ell$ -cast m) with the number ℓ of correct processes that deliver m (we say that they $k2\ell$ -deliver m). Both k and ℓ are subject to thresholding constraints: enough correct processes must $k2\ell$ -cast a message for it to be $k2\ell$ -delivered at least once; and as soon as one (correct) $k2\ell$ -delivery occurs, some minimal number of correct processes are guaranteed to $k2\ell$ -deliver as well.

⁴ Note that this message adversary is not limited to algorithms that use the `ur_broadcast` macro-operation. The same adversary can be equivalently defined for an operation `ur_multicast` that sends a message to a dynamically defined subset of processes (be it multiple recipients or only one in the case of unicast), by stipulating that the MA can still suppress up to d copies of this message. In this case, the most robust way for correct processes to disseminate a message is to send it to all processes, i.e. to fall back on a `ur_broadcast` operation.

⁵ An example of this family is the binary reliable broadcast introduced in [24], which is defined by specific delivery properties – not including MA-tolerance – allowing binary consensus to be solved efficiently with the help of a common coin.

More formally, $k2\ell$ -cast is a multi-shot abstraction, i.e. each app-message m that is $k2\ell$ -cast or $k2\ell$ -delivered is associated with an identity id . (Typically, such an identity is a pair consisting of a process identity and a sequence number.) It provides two operations, $k2\ell_cast$ and $k2\ell_deliver$, whose behavior is defined by the values of four parameters: three integers k' , k , ℓ , and a Boolean δ . This behavior is captured by the following six properties:

- Safety:
 - $k2\ell$ -VALIDITY. If a correct process p_i $k2\ell$ -delivers an app-message m with identity id , then at least k' correct processes $k2\ell$ -cast m with identity id .
 - $k2\ell$ -NO-DUPLICATION. A correct process $k2\ell$ -delivers at most one app-message m with identity id .
 - $k2\ell$ -CONDITIONAL-NO-DUPLICITY. If the Boolean δ is `true`, then no two different correct processes $k2\ell$ -deliver different app-messages with the same identity id .
- Liveness⁶:
 - $k2\ell$ -LOCAL-DELIVERY. If at least k correct processes $k2\ell$ -cast an app-message m with identity id and no correct process $k2\ell$ -casts an app-message $m' \neq m$ with identity id , then at least one correct process $k2\ell$ -delivers the app-message m with identity id .
 - $k2\ell$ -WEAK-GLOBAL-DELIVERY. If a correct process $k2\ell$ -delivers an app-message m with identity id , then at least ℓ correct processes $k2\ell$ -deliver an app-message m' with identity id (each of them possibly different from m).
 - $k2\ell$ -STRONG-GLOBAL-DELIVERY. If a correct process $k2\ell$ -delivers an app-message m with identity id , and no correct process $k2\ell$ -casts an app-message $m' \neq m$ with identity id , then at least ℓ correct processes $k2\ell$ -deliver the app-message m with identity id .

This specification is *parameterized* in the sense that each tuple (k', k, ℓ, δ) defines a specific communication abstraction with different guarantees. This versatility explains why the $k2\ell$ -cast abstraction can be used to produce highly compact reconstructions of existing BRB algorithms, rendering them MA-tolerant in the process (using four and three lines of pseudo-code respectively, see Section 5). Despite this versatility, however, we will see in Section 3.2 that $k2\ell$ -cast can be implemented using a single (parameterized) algorithm, underscoring the fundamental commonalities of MA-tolerant BRB algorithms.

Intuitively, the parameters k' , k , and ℓ hobble the disruption power of the Byzantine/MA adversary by setting limits on the number of correct processes that are either required or guaranteed to be involved in one communication “wave” (corresponding to one identity id). k' sets the minimal number of correct processes that must $k2\ell$ -cast for any $k2\ell$ -delivery to occur: it thus limits the ability of the Byzantine/MA adversary to trigger spurious $k2\ell$ -deliveries. The role of k is symmetrical. It guarantees that some $k2\ell$ -delivery will necessarily occur if k correct processes $k2\ell$ -cast some message. It thus prevents the adversary from silencing correct processes as soon as some critical mass of them participates. Finally, ℓ captures a “quite-a-few-or-nothing” guarantee that mirrors the traditional “all-or-nothing” delivery guarantee of traditional BRB. As soon as one correct $k2\ell$ -delivery occurs (for some identity id), then ℓ correct processes must also $k2\ell$ -deliver (with the same identity).

The fourth parameter, δ , is a flag that when `true` enforces agreement between $k2\ell$ -deliveries. When $\delta = \text{true}$, the $k2\ell$ -CONDITIONAL-NO-DUPLICITY property implies that all the app-messages m' involved in the $k2\ell$ -WEAK-GLOBAL-DELIVERY property are equal to m .

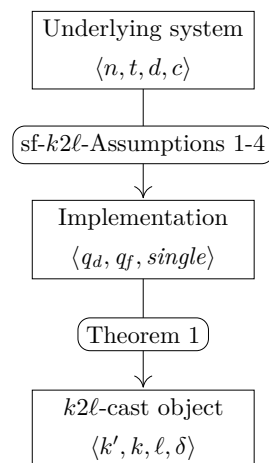
⁶ The liveness properties comprise a *local* delivery property that provides a necessary condition for the $k2\ell$ -delivery of an app-message by at least *one* correct process, and two *global* delivery properties that consider the collective behavior of correct processes.

■ **Algorithm 1** Signature-free $k2\ell$ -cast (code for p_i).

```

object SigFreeK2LCast( $q_d, q_f, single$ ) is
(1) operation  $k2\ell\_cast(m, id)$  is
(2)   if (ENDORSE( $-, id$ ) not already ur-broadcast)
(3)     then ur_broadcast(ENDORSE( $m, id$ ))
(4)   end if.
(5) when ENDORE( $m, id$ ) is received do
    % ----- forwarding step -----
(6)   if (ENDORSE( $m, id$ ) received from at least  $q_f$  processes
         $\wedge$  (( $\neg single \wedge$  ENDORE( $m, id$ ) not already ur-broadcast)
             $\vee$  ENDORE( $-, id$ ) not already ur-broadcast))
(7)     then ur_broadcast(ENDORSE( $m, id$ ))
(8)   end if;
    % ----- delivery step -----
(9)   if (ENDORSE( $m, id$ ) received from at least  $q_d$  processes
         $\wedge$  ( $-, id$ ) not already  $k2\ell$ -delivered)
(10)    then  $k2\ell\_deliver(m, id)$ 
(11)  end if.
end object.

```



■ **Figure 1** From the system parameters to a $k2\ell$ -cast implementation.

3.2 A Signature-Free Implementation of $k2\ell$ -Cast

Among the many possible ways of implementing $k2\ell$ -cast, this section presents a quorum-based⁷ signature-free implementation⁸ of the abstraction. To overcome the disruption caused by Byzantine processes and message losses from the MA, our algorithm uses the ur-broadcast primitive (cf. our communication model in Sec. 2) to accumulate and forward ENDORE messages before deciding whether to deliver. Forwarding and delivery are triggered by *two thresholds* (a pattern also found, for instance, in Bracha’s BRB algorithm [11]):

- A first threshold, q_d , triggers the delivery of an app-message m when enough ENDORE messages supporting m have been received.
- A second threshold, q_f , which is lower than q_d , controls how ENDORE messages are forwarded during the algorithm’s execution.

Forwarding, which is controlled by q_f , amplifies how correct processes react to ENDORE messages, and is instrumental to ensure the algorithm’s liveness. As soon as some critical “mass” of agreeing ENDORE messages accumulates within the system, forwarding triggers a chain reaction which guarantees that a minimum number of correct processes eventually $k2\ell$ -deliver the corresponding app-message.

More concretely, our algorithm provides an object (SigFreeK2LCast, Alg. 1), instantiated using the function SigFreeK2LCast($q_d, q_f, single$), using three input parameters:

- q_d : the number of matching ENDORE messages that must be received from distinct processes in order to $k2\ell$ -deliver an app-message.

⁷ In this paper, a quorum is a set of processes that (at the implementation level) ur-broadcast the same message. This definition takes quorums in their ordinary sense. In a deliberative assembly, a quorum is the minimum number of members that must vote the same way for an irrevocable decision to be taken. Let us notice that this definition does not require quorum intersection. However, if quorums have a size greater than $\frac{n+t}{2}$, the intersection of any two quorums contains, despite Byzantine processes, at least one correct process [11, 29].

⁸ Another $k2\ell$ -cast implementation, which uses digital signatures and allows to reach optimal values for k and ℓ , is presented in Section 6.

- q_f : the number of matching ENDORSE messages that must be received from distinct processes for the local p_i to endorse the corresponding app-message (if it has not yet).
- $single$: a Boolean that controls whether a given correct process can endorse different app-messages for the same identity id ($single = \text{false}$), or not ($single = \text{true}$).

The algorithm provides the operations $k2l_cast$ and $k2l_deliver$. Given an app-message m with identity id , the operation $k2l_cast(m, id)$ ur-broadcasts $ENDORSE(m, id)$ provided p_i has not yet endorsed any different app-message for the same identity id (lines 2-4). When p_i receives a message $ENDORSE(m, id)$, it executes two steps. If the forwarding quorum q_f has been reached, p_i first retransmits $ENDORSE(m, id)$ (Forwarding step, lines 6-8). Then, if the $k2l$ -delivery quorum q_d is attained, p_i $k2l$ -delivers m (Delivery step, lines 9-11).

For brevity, we define $\alpha = n + q_f - t - d - 1$. Given an execution defined by the system parameters n , t , d , and c , Alg. 1 requires the following assumptions to hold for the input parameters q_f and q_d of a $k2l$ -cast instance (a global picture linking all parameters is presented in Fig. 1). The prefix “sf” stands for signature-free.

- sf- $k2l$ -Assumption 1: $c - d \geq q_d \geq q_f + t \geq 2t + 1$,
- sf- $k2l$ -Assumption 2: $\alpha^2 - 4(q_f - 1)(n - t) \geq 0$,
- sf- $k2l$ -Assumption 3: $\alpha(q_d - 1) - (q_f - 1)(n - t) - (q_d - 1)^2 > 0$,
- sf- $k2l$ -Assumption 4: $\alpha(q_d - 1 - t) - (q_f - 1)(n - t) - (q_d - 1 - t)^2 \geq 0$.

In particular, the safety of Alg. 1 algorithm relies solely on sf- $k2l$ -Assumption 1, while its liveness relies on all four of them. sf- $k2l$ -Assumption 2 through 4 constrain the solutions of a second-degree inequality resulting from the combined action of the MA, the Byzantine processes, and the message-forwarding behavior of Alg. 1. We show in the extended version that, in practical cases, these assumptions can be satisfied by a bound of the form $n > \lambda t + \xi d + f(t, d)$, where $\lambda, \xi \in \mathbb{N}$ and $f(t, 0) = f(0, d) = 0$. Together, the assumptions allow Alg. 1 to provide a $k2l$ -cast abstraction (with values of the parameters k' , k , ℓ , and δ defining a specific $k2l$ -cast instance) as stated by the following theorem.

► **Theorem 1** (*$k2l$ -CORRECTNESS*). *If sf- $k2l$ -Assumptions 1–4 are verified, Alg. 1 implements $k2l$ -cast with the following guarantees:*

- $k2l$ -VALIDITY with $k' = q_f - n + c$,
- $k2l$ -NO-DUPLICATION,
- $k2l$ -CONDITIONAL-NO-DUPLICITY with $\delta = \left(q_f > \frac{n+t}{2} \right) \vee \left(single \wedge q_d > \frac{n+t}{2} \right)$,
- $k2l$ -LOCAL-DELIVERY with $k = \left\lfloor \frac{c(q_f-1)}{c-d-q_d+q_f} \right\rfloor + 1$,
- $\left\{ \begin{array}{ll} \text{if } single = \text{false,} & k2l\text{-WEAK-GLOBAL-DELIVERY} \\ \text{if } single = \text{true,} & k2l\text{-STRONG-GLOBAL-DELIVERY} \end{array} \right\}$ with $\ell = \left\lceil c \left(1 - \frac{d}{c-q_d+1} \right) \right\rceil$.

3.3 Proof of Algorithm 1

The proofs of the $k2l$ -cast safety properties stated in Theorem 1 ($k2l$ -VALIDITY, $k2l$ -NO-DUPLICATION, and $k2l$ -CONDITIONAL-NO-DUPLICITY) are fairly straightforward. To save space, these proofs are provided in the extended version.

The proofs of the $k2l$ -cast liveness properties ($k2l$ -LOCAL-DELIVERY, $k2l$ -WEAK-GLOBAL-DELIVERY, $k2l$ -STRONG-GLOBAL-DELIVERY) are sketched informally below (Lemmas 2-10). Their full development can be found in Appendix A.

When seeking to violate the liveness properties of $k2\ell$ -cast, the attacker can use the MA to control in part how many ENDORSE messages are received by each correct process, thus interfering with the quorum mechanisms defined by q_d and q_f . To analyze the joint effect of this interference with Byzantine faults, our proofs consider seven well-chosen subsets of correct processes (A, B, C, U, F, NF , and NB , depicted in Fig. 2a).

These subsets are defined for an execution of Alg. 1 in which k_I correct processes $k2\ell$ -cast (m, id) (the I in k_I is for “Initial”), and ℓ_e correct processes receive at least q_d message ENDORSE(m, id). The first three subsets, A, B , and C , partition correct processes based on the number of ENDORSE(m, id) messages they receive.

- A contains the ℓ_e correct processes that receive at least q_d ENDORSE(m, id) messages (be it from correct or from Byzantine processes), and thus $k2\ell$ -deliver some message.⁹
- B contains the correct processes that receive at least q_f but less than q_d ENDORSE(m, id) messages and thus do not $k2\ell$ -deliver (m, id) .
- C contains the remaining correct processes that receive less than q_f ENDORSE(m, id) messages. They neither forward nor deliver any message for identity id (since $q_f \leq q_d$).

In our proofs, we count how many messages ENDORSE(m, id) ur-broadcast by correct processes are received by the processes of A (resp. B and C). We note these quantities w_A^c , w_B^c , and w_C^c , and use them to bootstrap our proofs using bounds on messages (see below).

The last four subsets intersect with A, B and C , and distinguish correct processes based on the ur-broadcast operations they perform.

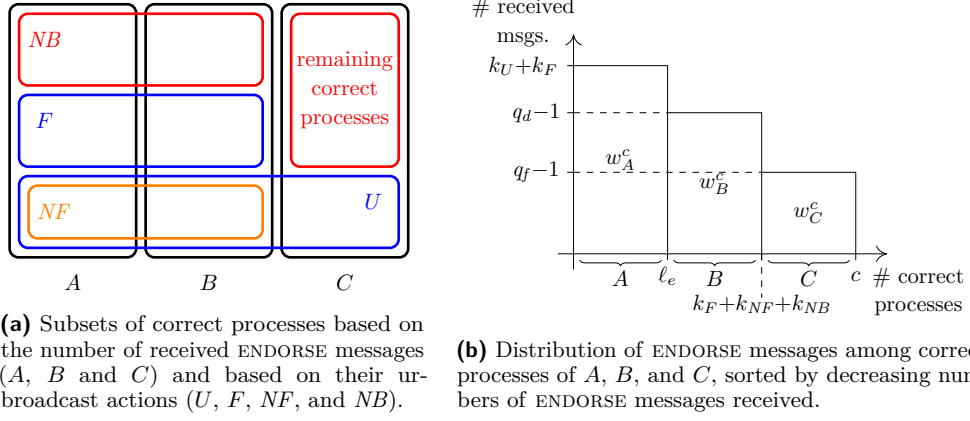
- U consists of the correct processes that ur-broadcast ENDORSE(m, id) at line 3.
- F denotes the correct processes of $A \cup B$ that ur-broadcast ENDORSE(m, id) at line 7 (i.e., they perform forwarding).
- NF denotes the correct processes of $A \cup B$ that ur-broadcast ENDORSE(m, id) at line 3.
- NB denotes the correct processes of $A \cup B$ that never ur-broadcast ENDORSE(m, id), be it at line 3 or at line 7. These processes have received at least q_f messages ENDORSE(m, id), but do not forward ENDORSE(m, id), because they have already ur-broadcast ENDORSE(m', id) at line 3 or at line 7 for an app-message $m' \neq m$.

Proof strategy. We note $k_U = |U|$, $k_F = |F|$, $k_{NF} = |NF|$, $k_{NB} = |NB|$. Observe that $k_U \leq k_I$ and $k_{NF} \leq k_U$, since all (correct) processes in U and NF invoke $k2\ell_cast$. Also, $(k_U + k_F)$ represents the total number of correct processes that ur-broadcast a message ENDORSE(m, id). Fig. 2b illustrates how these quantities constrain the distribution of ENDORSE messages across A, B and C . Our core proof strategy consists in bounding the areas shown in Fig. 2b. (For instance, observe that $w_A^c \leq |A| \times (k_U + k_F)$, since each of the ℓ_e correct processes in A can receive at most one ENDORSE message from each of the $(k_U + k_F)$ correct processes that send them.) This reasoning on bounds yields a polynomial involving $\ell_e = |A|$, k_I , and k_U , whose roots can then be constrained to yield the liveness guarantees required by the $k2\ell$ -cast specification.

Observation. In the same way we have bounded w_A^c , we can also bound w_B^c by observing that there are $(k_{NF} + k_{NB} + k_F - \ell_e)$ processes in B and that each can receive at most $q_d - 1$ ENDORSE messages. Similarly, we can bound w_C^c by observing that the $(c - k_{NF} - k_{NB} - k_F)$ processes of C can receive at most $q_f - 1$ ENDORSE messages. Thus:

⁹ Because of the condition at line 9, these processes do not necessarily $k2\ell$ -deliver (m, id) , but all do $k2\ell$ -deliver an app-message for identity id .

26:10 Signature-Free BRB Algorithms Under a Message Adversary



■ **Figure 2** Subsets of correct processes and distribution of ENDORSE messages among them.

$$w_A^c \leq (k_U + k_F)\ell_e, \quad (1)$$

$$w_B^c \leq (q_d - 1)(k_{NF} + k_{NB} + k_F - \ell_e), \quad (2)$$

$$w_C^c \leq (q_f - 1)(c - k_{NF} - k_{NB} - k_F). \quad (3)$$

Moreover, the MA cannot suppress more than d copies of each individual ENDORSE message ur-broadcast to the c correct processes. Thus, the total number of ENDORSE messages received by correct processes ($w_A^c + w_B^c + w_C^c$) is such that:

$$w_A^c + w_B^c + w_C^c \geq (k_U + k_F)(c - d). \quad (4)$$

► **Lemma 2.** $\ell_e \times (k_U + k_F - q_d + 1) \geq (k_U + k_F)(c - d - q_d + q_f) - c(q_f - 1) - k_{NB}(q_d - q_f)$.

Proof sketch. We get this result by combining (1), (2), (3) and (4), and using sf- $k2\ell$ -Assumption 1 with the fact that $k_{NF} \leq k_U$. (Full derivations in Appendix A.) ◀

► **Lemma 3.** *If no correct process $k2\ell$ -casts (m', id) with $m' \neq m$, then no correct process forwards $ENDORSE(m', id)$ at line 7 (and then $k_{NB} = 0$). (Proof in Appendix A.)*

► **Lemma 4** ($k2\ell$ -LOCAL-DELIVERY). *If at least $k = \lfloor \frac{c(q_f - 1)}{c - d - q_d + q_f} \rfloor + 1$ correct processes $k2\ell$ -cast an app-message m with identity id and no correct process $k2\ell$ -casts any app-message m' with identity id such that $m \neq m'$, then at least one correct process p_i $k2\ell$ -delivers m with identity id .*

Proof sketch. From the hypotheses, Lemma 3 helps us determine that $k_{NB} = 0$. Then, the property is proved by contraposition, by assuming that no correct process $k2\ell$ -delivers (m, id) , which leads us to $\ell_e = 0$. Using prior information and sf- $k2\ell$ -Assumption 1, we can rewrite the inequality of Lemma 2 to get the threshold of $k2\ell$ -casts above which there is at least one $k2\ell$ -delivery. (Full derivations in Appendix A.) ◀

► **Lemma 5.** $(single = false) \implies (k_{NB} = 0)$. (Proof in Appendix A.)

► **Lemma 6.** *If at least one correct process $k2\ell$ -delivers (m, id) and $x = k_U + k_F$ (the number of correct processes that ur-broadcast $ENDORSE(m, id)$ at line 3 or 7), then $x \geq q_d - t$ and $x^2 - x(c - d + q_f - 1 - k_{NB}) \geq -(c - k_{NB})(q_f - 1)$.*

Proof sketch. We prove this lemma by counting the total number of messages (sent by Byzantine or correct processes) that are received by the processes of A , and by using (1), (3) (4), and sf- $k2\ell$ -Assumption 1. (Full derivations in Appendix A.) ◀

► **Lemma 7.** *If $k_{NB} = 0$, and at least one correct process $k2\ell$ -delivers (m, id) , then $k_U + k_F \geq q_d$.*

Proof sketch. Given that $k_{NB} = 0$, we can rewrite the inequality of Lemma 6, which gives us a second-degree polynomial (where $x = k_U + k_F$ is the unknown variable). We compute its roots and show that the smaller one contradicts Lemma 6, and that the larger one is greater than or equal to q_d . The fact that x must be greater than or equal to the larger root to satisfy Lemma 6 proves the lemma. (Full derivations in Appendix A.) ◀

► **Lemma 8.** *If $k_{NB} = 0$ and $k_U + k_F \geq q_d$, then at least $\left\lceil c \left(1 - \frac{d}{c - q_d + 1}\right) \right\rceil$ correct processes $k2\ell$ -deliver some app-message with identity id (not necessarily m).*

Proof sketch. From the hypotheses, we can rewrite the inequality of Lemma 2 to get a lower bound on ℓ_e . Using sf- $k2\ell$ -Assumption 3, we can determine that this lower bound is decreasing with the number of ur-broadcasts by correct processes ($x = k_U + k_F$). Hence, this lower bound is minimum when x is maximum, that is, when $x = c$. This gives us the minimum number of correct processes that $k2\ell$ -deliver under the given hypotheses. (Full derivations in Appendix A.) ◀

► **Lemma 9** ($k2\ell$ -WEAK-GLOBAL-DELIVERY). *If $single = \text{false}$, and a correct process $k2\ell$ -delivers an app-message m with identity id , then at least $\ell = \left\lceil c \left(1 - \frac{d}{c - q_d + 1}\right) \right\rceil$ correct processes $k2\ell$ -deliver an app-message m' with identity id (each possibly different from m).*

Proof sketch. As $single = \text{false}$ and one correct process $k2\ell$ -delivers (m, id) , Lemmas 5 and 7 apply, and we have $k_{NB} = 0$ and $k_U + k_F \geq q_d$. This provides the prerequisites for Lemma 8, which concludes the proof. (Full derivations in Appendix A.) ◀

► **Lemma 10** ($k2\ell$ -STRONG-GLOBAL-DELIVERY). *If $single = \text{true}$, and a correct process $k2\ell$ -delivers an app-message m with identity id , and no correct process $k2\ell$ -casts an app-message $m' \neq m$ with identity id , then at least $\ell = \left\lceil c \left(1 - \frac{d}{c - q_d + 1}\right) \right\rceil$ correct processes $k2\ell$ -deliver m with identity id .*

Proof sketch. As $single = \text{true}$, Lemma 3 holds and implies that $k_{NB} = 0$. As above, Lemma 7 and Lemma 8 hold, yielding the lemma. (Full derivations in Appendix A.) ◀

4 BRB in the Presence of Message Adversary (MBRB): Definition

Before using the $k2\ell$ -cast abstraction to reconstruct MA-tolerant BRB algorithms, we first specify what a Byzantine- and MA-tolerant broadcast should precisely achieve. We call such a broadcast an MBR-broadcast (for Message-adversarial Byzantine Reliable Broadcast), or MBRB for short. The MBRB abstraction provides two matching operations, `mbrb_broadcast` and `mbrb_deliver`. It is a multishot abstraction, i.e., it associates an identity $\langle sn, i \rangle$ (sequence number, sender identity) with each app-message, and assumes that correct processes never reuse the same sequence number for different `mbrb_broadcast` invocations.

When, at the application level, a process p_i invokes `mbrb_broadcast(m, sn)`, where m is the app-message, we say it “mbrb-broadcasts (m, sn)”. Similarly, when the invocation of `mbrb_deliver` by p_i returns the tuple (m, sn, j) to the client application (where p_j is the sender process), we say it “mbrb-delivers (m, sn, j)”. So, the app-messages are *mbrb-broadcast* and *mbrb-delivered*. Because of the MA, we cannot always guarantee that an app-message mbrb-delivered by a correct process is eventually received by all correct processes. Hence, in the MBR-broadcast specification, we introduce a variable ℓ_{MBRB} (reminiscent of the ℓ of $k2\ell$ -cast) which indicates the strength of the global delivery guarantee of the primitive: if one correct process mbrb-delivers an app-message, then ℓ_{MBRB} correct processes eventually mbrb-deliver this app-message¹⁰. MBRB is defined by the following properties:

- Safety:
 - MBRB-VALIDITY. If a correct process p_i mbrb-delivers an app-message m from a correct process p_j with sequence number sn , then p_j mbrb-broadcast m with sequence number sn .
 - MBRB-NO-DUPLICATION. A correct process p_i mbrb-delivers at most one app-message from a process p_j with sequence number sn .
 - MBRB-NO-DUPLICITY. No two distinct correct processes mbrb-deliver different app-messages from a process p_i with the same sequence number sn .
- Liveness:
 - MBRB-LOCAL-DELIVERY. If a correct process p_i mbrb-broadcasts an app-message m with sequence number sn , then at least one correct process p_j eventually mbrb-delivers m from p_i with sequence number sn .
 - MBRB-GLOBAL-DELIVERY. If a correct process p_i mbrb-delivers an app-message m from a process p_j with sequence number sn , then at least ℓ_{MBRB} correct processes mbrb-deliver m from p_j with sequence number sn .

It is implicitly assumed that a correct process does not use the same sequence number twice. Let us observe that, as at the implementation level, the MA can always suppress all the messages sent to a fixed set D of d processes, these mbrb-delivery properties are the strongest that can be implemented. More generally, the best-guaranteed value for ℓ_{MBRB} is $c - d$. So, the previous specification boils down to Bracha’s specification [11] for $\ell_{MBRB} = c$.

5 $k2\ell$ -Cast in Action: From Classical BRB to MA-Tolerant BRB (MBRB) Algorithms

This section uses $k2\ell$ -cast to reconstruct two signature-free BRB algorithms [11, 20] initially introduced in a pure Byzantine context (i.e., without any MA). This reconstruction produces Byzantine-MA-tolerant versions of the initial algorithms that implement the MBRB specification of Section 4. Moreover, when $d = 0$, our two reconstructed BRB algorithms are strictly more efficient than the original algorithms that gave rise to them (they terminate earlier).

More precisely, the original and reconstructed versions of Bracha’s BRB are identical in terms of communication cost, time complexity, and t -resilience (when $d = 0$). The same comparison holds for the original and reconstructed versions of Imbs and Raynal’s BRB. However, both reconstructed BRB algorithms use smaller quorums than their original versions, and therefore require fewer messages to progress. In an actual network, this means a lower latency in practice, as practical networks typically exhibit a long tail distribution of latencies (a phenomenon well-studied by system and networking researchers [14, 15, 34]).

¹⁰If there is no MA (i.e. $d = 0$), we should have $\ell_{MBRB} = c \geq n - t$.

■ **Algorithm 2** $k2\ell$ -cast-based reconstruction of Bracha’s BRB algorithm (code of p_i).

init: $obj_E \leftarrow \text{SigFreeK2LCast}(q_d = \lfloor \frac{n+t}{2} \rfloor + 1, q_f = t+1, \text{single} = \text{true});$
 $obj_R \leftarrow \text{SigFreeK2LCast}(q_d = 2t+d+1, q_f = t+1, \text{single} = \text{true}).$

(1) **operation** $\text{mbrb_broadcast}(m, sn)$ **is** $\text{ur_broadcast}(\text{INIT}(m, sn)).$

(2) **when** $\text{INIT}(m, sn)$ **is received from** p_j **do** $obj_E.k2\ell_cast(\text{ECHO}(m), (sn, j)).$

(3) **when** $(\text{ECHO}(m), (sn, j))$ **is** $obj_E.k2\ell_delivered$ **do** $obj_R.k2\ell_cast(\text{READY}(m), (sn, i)).$

(4) **when** $(\text{READY}(m), (sn, j))$ **is** $obj_R.k2\ell_delivered$ **do** $\text{mbrb_deliver}(m, sn, j).$

To help readers familiar with the initial algorithms, we use the same message types (INIT, ECHO, READY, WITNESS) as in the original publications. It has been shown in [4] that the MBRB problem can be solved if and only if $n > 3t + 2d$.

5.1 Bracha’s BRB algorithm reconstructed

Reconstructed version. Bracha’s BRB algorithm comprises three phases. When a process invokes $\text{brb_broadcast}(m, sn)$, it disseminates the app-message m an INIT message (first phase). The reception of this message by a correct process triggers its participation in a second phase implemented by the exchange of messages tagged ECHO. Finally, when a process has received ECHO messages from “enough” processes, it enters the third phase, in which READY messages are exchanged, at the end of which it brb -delivers the app-message m . Alg. 2 is a reconstructed version of the Bracha’s BRB, which assumes $n > 3t + 2d + 2\sqrt{td}$.

The algorithm requires two instances of $k2\ell$ -cast, denoted obj_E and obj_R , associated with the ECHO messages and the READY messages, respectively. For both these objects, the Boolean single is set to **true**. For the quorums, we have the following:

- obj_E : $q_f = t + 1$ and $q_d = \lfloor \frac{n+t}{2} \rfloor + 1$,
- obj_R : $q_f = t + 1$ and $q_d = 2t + d + 1$.

The integer sn is the sequence number of the app-message m mbrb -broadcast by p_i . The identity of m is consequently the pair $\langle sn, i \rangle$.

Alg. 2 provides $\ell_{MBRB} = \left\lceil c \left(1 - \frac{d}{c-2t-d} \right) \right\rceil$ under:

- B87-Assumption (for Bracha 1987): $n > 3t + 2d + 2\sqrt{td}$;
- its proof of correctness can be found in the extended version.

Comparison (Table 2). When $d = 0$, both Bracha’s algorithm and its reconstruction use the same quorum size for the READY phase. The quorums of the ECHO phase are however different (Table 2). As the algorithm requires $n > 3t$, we define $\Delta = n - 3t$ as the slack between the lower bound on n and the actual value of n . When considering the forwarding threshold q_f , we have $\lfloor \frac{n+t}{2} \rfloor + 1 = 2t + \lfloor \frac{\Delta}{2} \rfloor + 1 > t + 1$. As a result, the reconstruction of Bracha’s algorithm always uses a lower forwarding threshold for ECHO messages than the original. It therefore forwards messages more rapidly and reaches the delivery quorum faster.

■ **Table 2** Bracha’s original version vs. $k2\ell$ -cast-based reconstruction when $d = 0$.

Threshold	Original version (ECHO phase)	$k2\ell$ -cast-based version (obj_E)
Forwarding q_f	$\lfloor \frac{n+t}{2} \rfloor + 1$	$t + 1$
Delivery q_d	$\lfloor \frac{n+t}{2} \rfloor + 1$	$\lfloor \frac{n+t}{2} \rfloor + 1$

■ **Algorithm 3** $k2\ell$ -cast-based reconstruction of Imbs and Raynal’s BRB algorithm (code of p_i).

init: $obj_w \leftarrow \text{SigFreeK2LCast}(q_d = \lfloor \frac{n+3t}{2} \rfloor + 3d + 1, q_f = \lfloor \frac{n+t}{2} \rfloor + 1, \text{single} = \text{false})$.

(1) **operation** $\text{mbrb_broadcast}(m, sn)$ is $\text{ur_broadcast}(\text{INIT}(m, sn))$.

(2) **when** $\text{INIT}(m, sn)$ is received **from** p_j **do** $obj_w.k2\ell_cast(\text{WITNESS}(m), (sn, j))$.

(3) **when** $(\text{WITNESS}(m), (sn, j))$ is $obj_w.k2\ell_delivered$ **do** $\text{mbrb_deliver}(m, sn, j)$.

5.2 Imbs and Raynal’s BRB algorithm reconstructed

Reconstructed version. Imbs and Raynal’s BRB is another BRB implementation, which achieves an optimal good-case latency (only two communication steps) at the cost of a non-optimal t -resilience. Its reconstructed version requires $n > 5t + 12d + \frac{2td}{t+2d}$.

The algorithm requires a single $k2\ell$ -cast object, denoted obj_w , associated with the WITNESS message, and which is instantiated with $q_f = \lfloor \frac{n+t}{2} \rfloor + 1$ and $q_d = \lfloor \frac{n+3t}{2} \rfloor + 3d + 1$, and the Boolean $single = \text{false}$. Similarly to Bracha’s reconstructed BRB, an identity of app-message in this algorithm is a pair $\langle sn, i \rangle$ containing a sequence number sn and a process identity i .

Alg. 3 provides $\ell_{MRRB} = \left\lceil c \left(1 - \frac{d}{c - \lfloor \frac{n+3t}{2} \rfloor - 3d} \right) \right\rceil$ under:

- IR16-Assumption (for Imbs-Raynal 2016): $n > 5t + 12d + \frac{2td}{t+2d}$; (where $t + d > 0$) its proof of correctness can be found in the extended version.

Comparison (Table 3). Table 3 compares Imbs and Raynal’s original algorithm against its $k2\ell$ -cast reconstruction for $d = 0$. Recall that this algorithm saves one communication step with respect to Bracha’s at the cost of a weaker t -tolerance, i.e., it requires $n > 5t$. As for Bracha, let us define the slack between n and its minimum as $\Delta = n - 5t$, we have $\Delta \geq 1$.

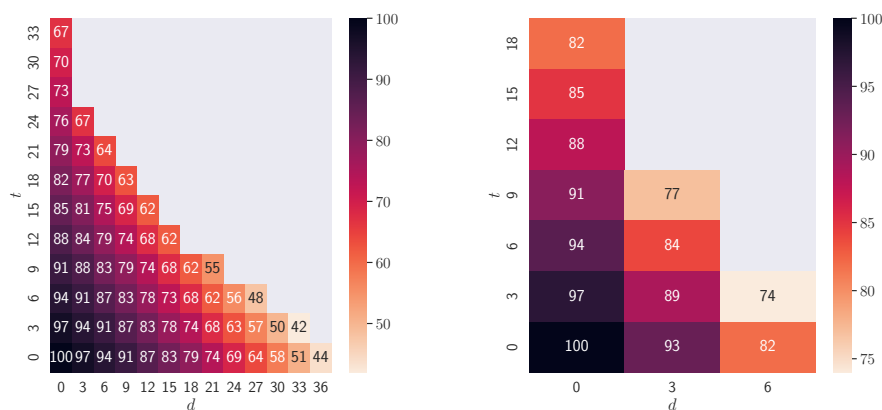
- Let us first consider the size of the forwarding quorum (first line of the table). We have $n - 2t = 3t + \Delta$ and $\lfloor \frac{n+t}{2} \rfloor + 1 = 3t + \lfloor \frac{\Delta}{2} \rfloor + 1$. When $\Delta > 2$, we always have $\Delta > \lfloor \frac{\Delta}{2} \rfloor + 1$, it follows that the forwarding predicate of the reconstructed version is equal or weaker than the one of the original version.
- The same occurs for the size of the delivery quorum (second line of the table). We have $n - t = 4t + \Delta$ and $\lfloor \frac{n+3t}{2} \rfloor + 1 = 4t + \lfloor \frac{\Delta}{2} \rfloor + 1$. So both reconstructed quorums are lower than those of the original version when $\Delta > 2$, making the reconstructed algorithm quicker as soon as $n \geq 5t + 3$. The two versions behave identically for $5t + 3 \geq n \geq 5t + 2$ ($\Delta \in \{1, 2\}$).

■ **Table 3** Imbs and Raynal’s original version vs. $k2\ell$ -cast-based reconstruction when $d = 0$.

Threshold	Original version (WITNESS phase)	$k2\ell$ -cast-based version (obj_w)
Forwarding q_f	$n - 2t$	$\lfloor \frac{n+t}{2} \rfloor + 1$
Delivery q_d	$n - t$	$\lfloor \frac{n+3t}{2} \rfloor + 1$

5.3 Numerical evaluation of the MRRB algorithms

Fig. 3 provides a numerical evaluation of the delivery guarantees of both $k2\ell$ -cast-based MRRB algorithms (Algs. 2 and 3) in the presence of Byzantine processes and an MA. Results were obtained for $n = 100$ and $c = n - t$, and show the values of ℓ_{MRRB} for different values of



(a) Reconstructed Bracha MBRB (Alg. 2). (b) Reconstructed Imbs-Raynal MBRB (Alg. 3).

■ **Figure 3** Values of ℓ_{MBRB} for the reconstructed BRB algorithms when varying t and d ($n = 100$ and $c = n - t$) within the ranges that satisfy B87-Assumption and IR16-Assumption.

t and d . For instance, Fig. 3a shows that with 6 Byzantine processes and an MA suppressing up to 9 ur-broadcast messages, Alg. 2 ensures the MBRB-GLOBAL-DELIVERY property with $\ell_{MBRB} = 83$. The figures illustrate that the reconstructed Bracha algorithm performs in a broader range of parameter values, mirroring the bounds on n , t , and d captured by B87-Assumption and IR16-Assumption. Nonetheless, both algorithms exhibit values of ℓ_{MBRB} that can support real-world applications in the presence of an MA.

6 A Signature-Based Implementation of $k2\ell$ -Cast

This section presents an implementation of $k2\ell$ -cast based on digital signatures. The underlying model is the same as that of Section 2 (page 4), except that the computing power of the attacker is now bounded, which allows us to leverage asymmetric cryptography.

6.1 Algorithm

The signature-based algorithm is described in Alg. 4. It uses an asymmetric cryptosystem to sign messages and verify their authenticity. Every process has a public/private key pair. Public keys are known to everyone, but private keys are only known to their owner. (Byzantine processes may exchange their private keys.) Each process also knows the mapping between process indexes and associated public keys, and each process can produce a unique, valid signature for a given message, and check if a signature is valid.

It is a simple algorithm that ensures that an app-message must be $k2\ell$ -cast by at least k correct processes to be $k2\ell$ -delivered by at least ℓ correct processes. For the sake of simplicity, we say that a correct process p_i “ur-broadcasts a set of signatures” if it ur-broadcasts a BUNDLE($m, id, sigs_i$) in which $sigs_i$ contains the signatures at hand. A correct process p_i ur-broadcasts an app-message m with identity id at line 5 or line 11.

- If this occurs at line 5, p_i includes in the message it ur-broadcasts all the signatures it has already received for (m, id) plus its own signature.
- If this occurs at line 11, p_i has just received a message containing a set of signatures $sigs$ for the pair (m, id) . The process p_i then aggregates in $sigs_i$ the valid signatures it just received with the ones it did know about beforehand (line 10).

This algorithm simply assumes: (the prefix “sb” stands for signature-based)

- sb- $k2\ell$ -Assumption 1: $c > 2d$,
- sb- $k2\ell$ -Assumption 2: $c - d \geq q_d \geq t + 1$.

■ **Algorithm 4** $k2\ell$ -cast implementation with signatures (code for p_i).

```

object SigBasedK2LCast( $q_d$ ) is
(1) operation  $k2\ell\_cast(m, id)$  is
(2)   if  $((-, id)$  not already signed by  $p_i$ ) then
(3)      $sig_i \leftarrow$  signature of  $(m, id)$  by  $p_i$ ;
(4)      $sigs_i \leftarrow$  {all valid signatures for  $(m, id)$  ur-broadcast by  $p_i$ }  $\cup$   $\{sig_i\}$ ;
(5)     ur_broadcast(BUNDLE( $m, id, sigs_i$ ));
(6)     check_delivery()
(7)   end if.
(8) when BUNDLE( $m, id, sigs$ ) is received do
(9)   if ( $sigs$  contains valid signatures for  $(m, id)$  not already ur-broadcast by  $p_i$ ) then
(10)     $sigs_i \leftarrow$  {all valid signatures for  $(m, id)$  ur-broadcast by  $p_i$ }
         $\cup$  {all valid signatures for  $(m, id)$  in  $sigs$ };
(11)    ur_broadcast(BUNDLE( $m, id, sigs_i$ ));
(12)    check_delivery()
(13)  end if.
(14) internal operation check_delivery() is
(15)  if ( $p_i$  ur-broadcast at least  $q_d$  valid signatures for  $(m, id)$ 
         $\wedge (-, id)$  not already  $k2\ell$ -delivered)
(16)    then  $k2\ell\_deliver(m, id)$ 
(17)  end if.
end object.

```

Thanks to digital signatures, processes can relay the messages of other processes in Alg. 4. The algorithm, however, does not use forwarding in the same way Alg. 1 did: there is no equivalent of q_f here, that is, the only way to “endorse” an app-message (which, in this case, is equivalent to signing this app-message) is to invoke the $k2\ell_cast$ operation. Furthermore, only one app-message can be endorsed by a correct process for a given identity (which is the equivalent of $single = \mathbf{true}$ in the signature-free version).

Although this implementation of $k2\ell$ -cast provides better guarantees than Alg. 1, using it to reconstruct signature-free BRB algorithms would be counter-productive. This is because signatures allow for MA-tolerant BRB algorithms that are more efficient in terms of round and message complexity than those that can be constructed using $k2\ell$ -cast [4].

However, a signature-based $k2\ell$ -cast does make sense in contexts in which many-to-many communication patterns are required [9], and, we believe, opens the path to novel ways to handle local state resynchronization resilient to Byzantine failures and message adversaries. For instance, we are using the following algorithm in our own work to design churn-tolerant money transfer systems tolerating Byzantine failures and temporary disconnections.

6.2 Guarantees

The proof of the following theorem can be found in the extended version.

► **Theorem 11** ($k2\ell$ -CORRECTNESS). *If sb- $k2\ell$ -Assumption 1 and 2 are verified, Alg. 4 implements $k2\ell$ -cast with the following guarantees: (i) $k' = q_d - n + c$, (ii) $k = q_d$, (iii) $\ell = c - d$, and (iv) $\delta = q_d > \frac{n+t}{2}$.*

7 Conclusion

This paper discussed reliable broadcast in asynchronous systems where an adversary can control some Byzantine processes and can suppress messages. Its starting point was the design of generic reliable broadcast abstractions suited to applications that do not require total order on the delivery of application messages (distributed money transfers are such applications [8, 10, 19]). However, the ability to thwart an adversary controlling Byzantine processes and a message adversary is new. This approach can be applied to the design of a wide range of quorum-based distributed algorithms other than reliable broadcast. For instance, we conjecture that $k2\ell$ -cast could benefit self-stabilizing and self-healing distributed systems [6], where a critical mass of messages from other processes is needed in order to re-synchronize the local state of a given process.

References

- 1 I. Abraham, K. Nayak, L. Ren, and Z. Xiang. Good-case latency of Byzantine broadcast: a complete categorization. In *Proc. 40th ACM Symposium on Principles of Distributed Computing (PODC'21)*, pages 331–341. ACM Press, 2021.
- 2 I. Abraham, L. Ren, and Z. Xiang. Good-case and bad-case latency of unauthenticated Byzantine broadcast: A complete categorization. In *Proc. 25th Int'l Conference on Principles of Distributed Systems (OPODIS'21)*, pages 5:1–5:20. LIPIcs, 2021.
- 3 Y. Afek and E. Gafni. Asynchrony from synchrony. In *Proc. 14th Int'l Conference on Distributed Computing and Networking (ICDCN'13)*, pages 225–239. Springer, 2021.
- 4 T. Albouy, D. Frey, M. Raynal, and F. Taïani. Byzantine-tolerant reliable broadcast in the presence of silent churn. In *Proc. 23th Int'l Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'21)*, pages 21–33. Springer, 2021. Extended version: [arXiv:2205.09992](https://arxiv.org/abs/2205.09992).
- 5 T. Albouy, D. Frey, M. Raynal, and F. Taïani. A modular approach to construct signature-free BRB algorithms under a message adversary, 2022. [arXiv:2204.13388](https://arxiv.org/abs/2204.13388).
- 6 K. Altisen, S. Devismes, S. Dubois, and F. Petit. *Introduction to distributed self-stabilizing algorithms*. Morgan & Claypool, 2019.
- 7 H. Attiya and J. Welch. *Distributed computing: fundamentals, simulations and advanced topics*. Wiley-Interscience, 2004.
- 8 A. Auvolat, D. Frey, M. Raynal, and F. Taïani. Money transfer made simple: a specification, a generic algorithm, and its proof. *Bulletin of EATCS (European Association of Theoretical Computer Science)*, 132:22–43, 2020.
- 9 A. Auvolat, M. Raynal, and F. Taïani. Byzantine-tolerant set-constrained delivery broadcast. In *Proc. 23rd Int'l Conference on Principles of Distributed Systems (OPODIS'19)*, pages 6:1–6:23. LIPIcs, 2019.
- 10 M. Baudet, G. Danezis, and A. Sonnino. Fastpay: high-performance Byzantine fault tolerant settlement. In *Proc. 2nd ACM Conference on Advances in Financial Technologies (AFT'20)*, pages 163–177. ACM Press, 2020.
- 11 G. Bracha. Asynchronous Byzantine agreement protocols. *Information & Computation*, 75(2):130–143, 1987.
- 12 C. Cachin, R. Guerraoui, and L. Rodrigues. *Reliable and secure distributed programming*. Springer, 2011.
- 13 B. Charron-Bost and A. Schiper. The heard-of model: computing in distributed systems with benign faults. *Distributed Computing*, 22(1):49–71, 2009.
- 14 X. Chen, H. Song, J. Jiang, C. Ruan, C. Li, S. Wang, G. Zhang, R. Cheng, and H. Cui. Achieving low tail-latency and high scalability for serializable transactions in edge computing. In *Proc. 16th European Conference on Computer Systems (EuroSys'21)*, pages 210–227. ACM Press, 2021.

- 15 D. Didona and W. Zwaenepoel. Size-aware sharding for improving tail latencies in in-memory key-value stores. In *Proc. 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*, pages 79–94. USENIX Association, 2019.
- 16 D. Dolev. The Byzantine generals strike again. *Journal of Algorithms*, 3:14–20, 1982.
- 17 C. Dwork, D. Peleg, N. Pippenger, and E. Upfal. Fault tolerance in networks of bounded degree. *SIAM Journal of Computing*, 17(5):975–988, 1988.
- 18 R. Guerraoui, J. Komatovic, P. Kuznetsov, Y.A. Pignolet, D.A. Seredinschi, and A. Tonkikh. Dynamic Byzantine reliable broadcast. In *Proc. 24th Int'l Conference on Principles of Distributed Systems (OPODIS'20)*, pages 23:1–23:18. LIPIcs, 2020.
- 19 R. Guerraoui, P. Kuznetsov, M. Monti, M. Pavlovic, and D.A. Seredinschi. The consensus number of a cryptocurrency. In *Proc. 38th ACM Symposium on Principles of Distributed Computing (PODC'19)*, pages 307–316. ACM Press, 2019.
- 20 D. Imbs and M. Raynal. Trading t -resilience for efficiency in asynchronous Byzantine reliable broadcast. *Parallel Processing Letters*, 26(4):1650017:1–1650017:8, 2016.
- 21 L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.
- 22 D. Malkhi and M.K. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.
- 23 A. Maurer, X. Défago, and S. Tixeuil. Communicating reliably in multi-hop dynamic networks despite Byzantine failures. In *Proc. 34th Symposium on Reliable Distributed Systems (SRDS'15)*, pages 238–245. IEEE Press, 2015.
- 24 A. Mostéfaoui, H. Moumen, and M. Raynal. Signature-free asynchronous byzantine consensus with $t < n/3$ and $O(n^2)$ messages. In *Proc. 33th ACM Symposium on Principles of Distributed Computing (PODC'14)*, pages 2–9. ACM Press, 2014.
- 25 K. Nayak, L. Ren, E. Shi, N.H. Vaidya, and Z. Xiang. Improved extension protocols for Byzantine broadcast and agreement. In *Proc. 34rd Int'l Symposium on Distributed Computing (DISC'20)*, pages 28:1–28:17. LIPIcs, 2020.
- 26 M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27:228–234, 1980.
- 27 M. Raynal. *Distributed algorithms for message-passing systems*. Springer, 2013.
- 28 M. Raynal. Message adversaries. In *Encyclopedia of Algorithms*, pages 1272–1276. Springer, 2016.
- 29 M. Raynal. *Fault-tolerant message-passing distributed systems: an algorithmic approach*. Springer, 2018.
- 30 M. Raynal and J. Stainer. Synchrony weakened by message adversaries vs asynchrony restricted by failure detectors. In *Proc. 32nd ACM Symposium on Principles of Distributed Computing (PODC'13)*, pages 166–175. ACM Press, 2013.
- 31 N. Santoro and P. Widmayer. Time is not a healer. In *Proc. 6th Annual Symposium on Theoretical Aspects of Computer Science (STACS'89)*, pages 304–316. Springer, 1989.
- 32 N. Santoro and P. Widmayer. Agreement in synchronous networks with ubiquitous faults. *Theoretical Computer Science*, 384(2-3):232–249, 2007.
- 33 L. Tseng, Q. Zhang, S. Kumar, and Y. Zhang. Exact consensus under global asymmetric Byzantine links. In *Proc. 40th IEEE Int'l Conference on Distributed Computing Systems (ICDCS 2020)*, pages 721–731. IEEE Press, 2020.
- 34 L. Yang, S.J. Park, M. Alizadeh, S. Kannan, and D. Tse. DispersedLedger: high-throughput Byzantine consensus on variable bandwidth networks. In *Proc. 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI'22)*, pages 493–512. USENIX Association, 2022.

A

Liveness Proof of the Signature-Free $k2\ell$ -cast Implementation (Algorithm 1)

► **Lemma 2.** $\ell_e \times (k_U + k_F - q_d + 1) \geq (k_U + k_F)(c - d - q_d + q_f) - c(q_f - 1) - k_{NB}(q_d - q_f)$.

Proof. Combining (1), (2), (3) and (4) yields:

$$\begin{aligned} (k_U + k_F)\ell_e + (q_d - 1)(k_{NF} + k_{NB} + k_F - \ell_e) + \\ (q_f - 1)(c - k_{NF} - k_{NB} - k_F) &\geq (k_U + k_F)(c - d), \\ \ell_e \times (k_U + k_F - q_d + 1) &\geq (k_U + k_F)(c - d) - (q_d - 1)(k_{NF} + k_{NB} + k_F) - \\ &\quad (q_f - 1)(c - k_{NF} - k_{NB} - k_F), \\ &\geq (k_U + k_F)(c - d) - (q_d - q_f)(k_{NF} + k_{NB} + k_F) - c(q_f - 1). \end{aligned}$$

Using sf- $k2\ell$ -Assumption 1, we have $q_d - q_f \geq 0$. By definition, we also have $k_{NF} \leq k_U$, which yields:

$$\begin{aligned} \ell_e \times (k_U + k_F - q_d + 1) &\geq (k_U + k_F)(c - d) - (q_d - q_f)(k_U + k_F + k_{NB}) - c(q_f - 1), \\ &\geq (k_U + k_F)(c - d - q_d + q_f) - c(q_f - 1) - k_{NB}(q_d - q_f). \quad \blacktriangleleft \end{aligned}$$

► **Lemma 3.** *If no correct process $k2\ell$ -casts (m', id) with $m' \neq m$, then no correct process forwards $ENDORSE(m', id)$ at line 7 (and then $k_{NB} = 0$).*

Proof. Assume there is a correct process that ur-broadcasts $ENDORSE(m', id)$ at line 7 with $m' \neq m$. Let us consider the first such process p_i . To execute line 7, p_i must first receive q_f messages $ENDORSE(m', id)$ from distinct processes. Since $q_f > t$ (sf- $k2\ell$ -Assumption 1), at least one of these processes, p_j , is correct. Since p_i is the first correct process to forward $ENDORSE(m', id)$ at line 7, the $ENDORSE(m', id)$ message of p_j must come from line 3, and p_j must have $k2\ell$ -cast (m', id) . We have assumed that no correct process $k2\ell$ -cast $m' \neq m$, therefore $m' = m$. Contradiction.

We conclude that, under these assumptions, no correct process ur-broadcasts $ENDORSE(m', id)$ with $m' \neq m$, be it at line 3 (by assumption) or at line 7 (shown by this proof). As a result, $k_{NB} = 0$. ◀

► **Lemma 4** ($k2\ell$ -LOCAL-DELIVERY). *If at least $k = \left\lfloor \frac{c(q_f - 1)}{c - d - q_d + q_f} \right\rfloor + 1$ correct processes $k2\ell$ -cast an app-message m with identity id and no correct process $k2\ell$ -casts any app-message m' with identity id such that $m \neq m'$, then at least one correct process p_i $k2\ell$ -delivers m with identity id .*

Proof. Let us assume that no correct process $k2\ell$ -casts (m', id) with $m' \neq m$. No correct process therefore ur-broadcasts $ENDORSE(m', id)$ with $m' \neq m$ at line 3. Lemma 3 also applies and no correct process forwards $ENDORSE(m', id)$ with $m' \neq m$ at line 7 either, so $k_{NB} = 0$. Because no correct process ur-broadcasts $ENDORSE(m', id)$ with $m' \neq m$ whether at line 3 or 7, a correct process receives at most t messages $ENDORSE(m', id)$ (all coming from Byzantine processes). As by sf- $k2\ell$ -Assumption 1, $t < q_d$, no correct process $k2\ell$ -delivers (m', id) with $m' \neq m$ at line 10.

We now prove the contraposition of the Lemma. Let us assume no correct process $k2\ell$ -delivers (m, id) . Since, by our earlier observations, no correct process $k2\ell$ -delivers (m', id) with $m' \neq m$ either, the condition at line 9 implies that no correct process ever receives at least q_d $ENDORSE(m, id)$, and therefore $\ell_e = 0$. By Lemma 2 we have $c(q_f - 1) \geq (k_U + k_F)(c - d - q_d + q_f)$. sf- $k2\ell$ -Assumption 1 implies that $c - d - q_d \geq 0 \iff c - d - q_d + q_f > 0$

(as $q_f \geq t + 1 \geq 1$), leading to $k_U + k_F \leq \frac{c(q_f-1)}{c-d-q_d+q_f}$. Because of the condition at line 2, a correct process p_j that has $k2\ell$ -cast (m, id) but has not ur-broadcast $\text{ENDORSE}(m, id)$ at line 3 has necessarily ur-broadcast $\text{ENDORSE}(m, id)$ at line 7. We therefore have $k_I \leq k_U + k_F$, which gives $k_I \leq \frac{c(q_f-1)}{c-d-q_d+q_f}$. By contraposition, if $k_I > \frac{c(q_f-1)}{c-d-q_d+q_f}$, then at least one correct process must $k2\ell$ -deliver (m, id) . Hence, we have $k = \left\lfloor \frac{c(q_f-1)}{c-d-q_d+q_f} \right\rfloor + 1$. ◀

► **Lemma 5.** ($single = \text{false}$) $\implies (k_{NB} = 0)$.

Proof. Let us consider a correct process $p_i \in A \cup B$. If we assume $p_i \notin F$, p_i never executes line 7 by definition. Because $p_i \in A \cup B$, p_i has received at least q_f messages $\text{ENDORSE}(m, id)$, and therefore did not fulfill the condition at line 6 when it received its q_f^{th} message $\text{ENDORSE}(m, id)$. As $single = \text{false}$ by Lemma assumption, to falsify this condition, p_i must have had already ur-broadcast $\text{ENDORSE}(m, id)$ when this happened. Because p_i never executes line 7, this implies that p_i ur-broadcasts $\text{ENDORSE}(m, id)$ at line 3, and therefore $p_i \in NF$. This reasoning proves that $A \cup B \setminus F \subseteq NF$. As the sets F , NF and NB partition $A \cup B$, this shows that $NB = \emptyset$, and $k_{NB} = |\emptyset| = 0$. ◀

► **Lemma 6.** *If at least one correct process $k2\ell$ -delivers (m, id) and $x = k_U + k_F$ (the number of correct processes that ur-broadcast $\text{ENDORSE}(m, id)$ at line 3 or 7), then $x \geq q_d - t$ and $x^2 - x(c - d + q_f - 1 - k_{NB}) \geq -(c - k_{NB})(q_f - 1)$.*

Proof. Let us write w_A^b the total number of $\text{ENDORSE}(m, id)$ messages from Byzantine processes received by the processes of A , and $w_A = w_A^c + w_A^b$ the total of number $\text{ENDORSE}(m, id)$ messages received by the processes of A , whether these ENDORSE messages originated from correct or Byzantine senders. By definition, $w_A^b \leq t\ell_e$ and $w_A \geq q_d\ell_e$. By combining these two inequalities with (1) on w_A^c we obtain:

$$\begin{aligned} q_d\ell_e \leq w_A &= w_A^c + w_A^b \leq (k_U + k_F)\ell_e + t\ell_e = (t + k_U + k_F)\ell_e, \\ q_d &\leq t + k_U + k_F, && \text{(as } \ell_e > 0) \\ q_d - t &\leq k_U + k_F = x. && (5) \end{aligned}$$

This proves the first inequality of the lemma. The processes in $A \cup B$ each receive at most $k_U + k_F$ distinct $\text{ENDORSE}(m, id)$ messages from correct processes, so we have $w_A^c + w_B^c \leq (k_{NF} + k_F + k_{NB})(k_U + k_F)$. Combined with the inequalities (3) on w_C^c and (4) on $w_A^c + w_B^c + w_C^c$ that remain valid in this case, we now have:

$$\begin{aligned} (k_{NF} + k_F + k_{NB})(k_U + k_F) + (q_f - 1)(c - k_{NF} - k_{NB} - k_F) &\geq (k_U + k_F)(c - d), \\ (k_{NF} + k_F + k_{NB})(k_U + k_F - q_f + 1) &\geq (k_U + k_F)(c - d) - c(q_f - 1). \end{aligned} \quad (6)$$

Let us determine the sign of $(k_U + k_F - q_f + 1)$. We derive from (5):

$$\begin{aligned} k_U + k_F - q_f + 1 &\geq q_d - t - q_f + 1 \\ &\geq 1 > 0. && \text{(as } q_d - q_f \geq t \text{ by sf-}k2\ell\text{-Assumption 1)} \end{aligned}$$

As $(k_U + k_F - q_f + 1)$ is positive and we have $k_U \geq k_{NF}$ by definition, we can transform (6) into:

$$\begin{aligned} (k_U + k_F + k_{NB})(k_U + k_F - q_f + 1) &\geq (k_U + k_F)(c - d) - c(q_f - 1), \\ (x + k_{NB})(x - q_f + 1) &\geq x(c - d) - c(q_f - 1), && \text{(as } x = k_U + k_F) \\ x^2 - x(c - d + q_f - 1 - k_{NB}) &\geq -(c - k_{NB})(q_f - 1). && \blacktriangleleft \end{aligned}$$

► **Lemma 7.** *If $k_{NB} = 0$, and at least one correct process $k2\ell$ -delivers (m, id) , then $k_U + k_F \geq q_d$.*

Proof. By Lemma 6 we have:

$$x^2 - x(c - d + q_f - 1 - k_{NB}) \geq -(c - k_{NB})(q_f - 1), \quad (7)$$

As (7) holds for all, values of $c \in [n - t, n]$, we can in particular consider $c = n - t$. Moreover, as by hypothesis, $k_{NB} = 0$, we have.

$$\begin{aligned} x^2 - x(n - t - d + q_f - 1) + (q_f - 1)(n - t) &\geq 0, \\ x^2 - \alpha x + (q_f - 1)(n - t) &\geq 0. \end{aligned} \quad (\text{by definition of } \alpha) \quad (8)$$

Let us first observe that the discriminant of the second-degree polynomial in (8) is non negative, i.e. $\alpha^2 - 4(q_f - 1)(n - t) \geq 0$ by sf- $k2\ell$ -Assumption 2. This allows us to compute the two real-valued roots as follows:

$$r_0 = \frac{\alpha}{2} - \frac{\sqrt{\alpha^2 - 4(q_f - 1)(n - t)}}{2} \quad \text{and} \quad r_1 = \frac{\alpha}{2} + \frac{\sqrt{\alpha^2 - 4(q_f - 1)(n - t)}}{2}.$$

Thus (8) is satisfied if and only if $x \leq r_0 \vee x \geq r_1$.

■ Let us prove $r_0 \leq q_d - 1 - t$. We need to show that:

$$\begin{aligned} \frac{\alpha}{2} - \frac{\sqrt{\alpha^2 - 4(q_f - 1)(n - t)}}{2} &\leq q_d - 1 - t \\ \frac{\alpha}{2} - (q_d - 1) + t &\leq \frac{\sqrt{\alpha^2 - 4(q_f - 1)(n - t)}}{2} \\ \frac{\sqrt{\alpha^2 - 4(q_f - 1)(n - t)}}{2} &\geq \frac{\alpha}{2} - (q_d - 1) + t \\ \sqrt{\alpha^2 - 4(q_f - 1)(n - t)} &\geq \alpha - 2(q_d - 1) + 2t. \end{aligned}$$

The inequality is trivially satisfied if $\alpha - 2(q_d - 1) + 2t < 0$. For all other cases, we need to verify that:

$$\begin{aligned} \alpha^2 - 4(q_f - 1)(n - t) &\geq (\alpha - 2(q_d - 1) + 2t)^2, \\ \alpha^2 - 4(q_f - 1)(n - t) &\geq \alpha^2 + 4(q_d - 1)^2 + 4t^2 - 4\alpha(q_d - 1) + 4\alpha t - 8t(q_d - 1), \\ -4(q_f - 1)(n - t) &\geq 4(q_d - 1)^2 + 4t^2 - 4\alpha(q_d - 1) + 4\alpha t - 8t(q_d - 1), \\ -(q_f - 1)(n - t) &\geq (q_d - 1)^2 + t^2 - \alpha(q_d - 1) + \alpha t - 2t(q_d - 1), \\ -(q_f - 1)(n - t) &\geq (q_d - 1 - t)^2 - \alpha(q_d - 1 - t), \end{aligned}$$

and thus $\alpha(q_d - 1 - t) - (q_f - 1)(n - t) - (q_d - 1 - t)^2 \geq 0$, which is true by sf- $k2\ell$ -Assumption 4.

■ Let us prove $r_1 > q_d - 1$. We want to show that:

$$\frac{\alpha}{2} + \frac{\sqrt{\alpha^2 - 4(q_f - 1)(n - t)}}{2} > q_d - 1$$

Let us rewrite the inequality as follows:

$$\begin{aligned} \alpha + \sqrt{\alpha^2 - 4(q_f - 1)(n - t)} &> 2(q_d - 1) \\ \sqrt{\alpha^2 - 4(q_f - 1)(n - t)} &> 2(q_d - 1) - \alpha \end{aligned}$$

26:22 Signature-Free BRB Algorithms Under a Message Adversary

The inequality is trivially satisfied if $2(q_d - 1) - \alpha < 0$. For all other cases, we can take the squares as follows:

$$\begin{aligned}\alpha^2 - 4(q_f - 1)(n - t) &> (2(q_d - 1) - \alpha)^2, \\ \alpha^2 - 4(q_f - 1)(n - t) &> 4(q_d - 1)^2 + \alpha^2 - 4\alpha(q_d - 1), \\ -4(q_f - 1)(n - t) &> 4(q_d - 1)^2 - 4\alpha(q_d - 1), \\ 4\alpha(q_d - 1) - 4(q_f - 1)(n - t) - 4(q_d - 1)^2 &> 0, \\ \alpha(q_d - 1) - (q_f - 1)(n - t) - (q_d - 1)^2 &> 0,\end{aligned}$$

which is true by sf- $k2\ell$ -Assumption 3.

We now know that $r_0 \leq q_d - 1 - t$ and that $r_1 > q_d - 1$. In addition, as $x \leq r_0 \vee x \geq r_1$, we have $x \leq q_d - t - 1 \vee x > q_d - 1$. But Lemma 6 states that $x \geq q_d - t$, which is incompatible with $x \leq q_d - t - 1$. So we are left with $x > q_d - 1$, which implies, as q_d and x are integers that $x \geq q_d$, thus proving the lemma for $c = n - t$.

Let us now consider the set E_0 of all executions in which t processes are Byzantine, and therefore $c = n - t$, and a set E_c of executions in which there are fewer Byzantine processes, and thus $c > n - t$ correct processes. We show that $E_c \subseteq E_0$ in that a Byzantine process can always simulate the behavior of a correct process. In particular, if the simulated correct process is not subject to the message adversary, the simulating Byzantine process simply operates like a correct process. If, on the other hand, the simulated correct process misses some messages as a result of the message adversary, the Byzantine process can also simulate missing such messages. As a result, the executions that can happen when $c > n - t$ can also happen when $c = n - t$. Thus our result proven for $c = n - t$ can be extended to all possible values of c . ◀

► **Lemma 8.** *If $k_{NB} = 0$ and $k_U + k_F \geq q_d$, then at least $\left\lceil c \left(1 - \frac{d}{c - q_d + 1}\right) \right\rceil$ correct processes $k2\ell$ -deliver some app-message with identity id (not necessarily m).*

Proof. As $k_{NB} = 0$ and $k_U + k_F \geq q_d$, we can rewrite the inequality of Lemma 2 into:

$$\ell_e \times (k_U + k_F - q_d + 1) \geq (k_U + k_F)(c - d - q_d + q_f) - c(q_f - 1).$$

From $k_U + k_F \geq q_d$ we derive $k_U + k_F - q_d + 1 > 0$, and we transform the above inequality into:

$$\ell_e \geq \frac{(k_U + k_F)(c - d - q_d + q_f) - c(q_f - 1)}{k_U + k_F - q_d + 1}.$$

Let us now focus on the case in which $c = n - t$, we obtain:

$$\ell_e \geq \frac{(k_U + k_F)(n - t - d - q_d + q_f) - (n - t)(q_f - 1)}{k_U + k_F - q_d + 1}.$$

The right side of the inequality is of the form:

$$\ell_e \geq \frac{\phi x - \beta}{x - \gamma} = \phi + \frac{\phi\gamma - \beta}{x - \gamma} \tag{9}$$

with:

$$\begin{aligned}x &= k_U + k_F, \\ \gamma &= q_d - 1, \\ \alpha &= n - t - d + q_f - 1, \\ \phi &= n - t - d - q_d + q_f, \\ \beta &= c(q_f - 1).\end{aligned}$$

Since, by hypothesis, $x = k_U + k_F \geq q_d$, we have:

$$x - \gamma = k_U + k_F - q_d + 1 > 0. \quad (10)$$

We also have:

$$\begin{aligned} \phi\gamma - \beta &= (\alpha - \gamma)\gamma - c(q_f - 1) = \alpha\gamma - \gamma^2 - c(q_f - 1), \\ &= \alpha(q_d - 1) - (q_d - 1)^2 - (n - t)(q_f - 1) > 0, \quad (\text{by sf-}k2\ell\text{-Assumption 3}) \\ \phi\gamma - \beta &> 0. \end{aligned} \quad (11)$$

Injecting (10) and (11) into (9), we conclude that $\phi + \frac{\phi\gamma - \beta}{x - \gamma}$ is a *decreasing hyperbole* defined over $x \in]\gamma, \infty]$ with *asymptotic value* ϕ when $x \rightarrow \infty$. As x is a number of correct processes, $x \leq c$. The decreasing nature of the right-hand side of (9) leads us to: $\ell_e \geq \phi + \frac{\phi\gamma - \beta}{c - \gamma} = \frac{\phi c - \beta}{c - \gamma} \geq \frac{c(c - d - q_d + q_f) - c(q_f - 1)}{c - q_d + 1} \geq c \times \frac{c - d - q_d + 1}{c - q_d + 1} = c \left(1 - \frac{d}{c - q_d + 1}\right)$.

Since ℓ_e is a positive integer, we conclude that at least $\ell_{\min} = \left\lceil c \left(1 - \frac{d}{c - q_d + 1}\right) \right\rceil$ correct processes receive at least q_d message $\text{ENDORSE}(m, id)$ at line 9. As each of these processes either $k2\ell$ -delivers (m, id) when this first happens, or has already $k2\ell$ -delivered another app-message $m' \neq m$ with identity id , we conclude that at least ℓ_{\min} correct processes $k2\ell$ -deliver some app-message (whether it be m or $m' \neq m$) with identity id when $c = n - t$. The reasoning for extending this result to any value of $c \in [n - t, n]$ is identical to the one at the end of the proof of Lemma 7 just above. ◀

► **Lemma 9** (*k2ℓ-WEAK-GLOBAL-DELIVERY*). *If $single = \mathbf{false}$, and a correct process $k2\ell$ -delivers an app-message m with identity id , then at least $\ell = \left\lceil c \left(1 - \frac{d}{c - q_d + 1}\right) \right\rceil$ correct processes $k2\ell$ -deliver an app-message m' with identity id (each possibly different from m).*

Proof. Let us assume $single = \mathbf{false}$, and one correct process $k2\ell$ -delivers (m, id) . By Lemma 5, $k_{NB} = 0$. The prerequisites for Lemma 7 are verified, and therefore $k_U + k_F \geq q_d$. This provides the prerequisites for Lemma 8, from which we conclude that at least $\ell = \left\lceil c \left(1 - \frac{d}{c - q_d + 1}\right) \right\rceil$ correct processes $k2\ell$ -deliver an app-message m' with identity id , which concludes the proof of the lemma. ◀

► **Lemma 10** (*k2ℓ-STRONG-GLOBAL-DELIVERY*). *If $single = \mathbf{true}$, and a correct process $k2\ell$ -delivers an app-message m with identity id , and no correct process $k2\ell$ -casts an app-message $m' \neq m$ with identity id , then at least $\ell = \left\lceil c \left(1 - \frac{d}{c - q_d + 1}\right) \right\rceil$ correct processes $k2\ell$ -deliver m with identity id .*

Proof. Let us assume that (i) $single = \mathbf{true}$, (ii) no correct process $k2\ell$ -casts (m', id) with $m' \neq m$, and (iii) one correct process $k2\ell$ -delivers (m, id) . Lemma 3 holds and implies that $k_{NB} = 0$. From there, as above, Lemmas 7 and 8 hold, and at least $\ell = \left\lceil c \left(1 - \frac{d}{c - q_d + 1}\right) \right\rceil$ correct processes $k2\ell$ -deliver an app-message for identity id .

By hypothesis, no correct process ur-broadcasts $\text{ENDORSE}(m', id)$ at line 3 with $m' \neq m$. Similarly, because of Lemma 3, no correct process ur-broadcasts $\text{ENDORSE}(m', id)$ at line 7 with $m' \neq m$. As a result, a correct process can receive at most receive t messages $\text{ENDORSE}(m', id)$ at line 9 (all from Byzantine processes). As $q_d > t$ (by sf- $k2\ell$ -Assumption 1), the condition of line 9 never becomes true for $m' \neq m$, and as result no correct process delivers an app-message $m' \neq m$ with identity id . All processes that $k2\ell$ -deliver an app-message with identity id , therefore, $k2\ell$ -deliver m , which concludes the lemma. ◀

Design of Self-Stabilizing Approximation Algorithms via a Primal-Dual Approach

Yuval Emek ✉

Technion – Israel Institute of Technology, Haifa, Israel

Yuval Gil ✉

Technion – Israel Institute of Technology, Haifa, Israel

Noga Harlev ✉

Technion – Israel Institute of Technology, Haifa, Israel

Abstract

Self-stabilization is an important concept in the realm of fault-tolerant distributed computing. In this paper, we propose a new approach that relies on the properties of linear programming duality to obtain self-stabilizing approximation algorithms for distributed graph optimization problems. The power of this new approach is demonstrated by the following results:

- A self-stabilizing $2(1 + \varepsilon)$ -approximation algorithm for minimum weight vertex cover that converges in $O(\log \Delta / (\varepsilon \log \log \Delta))$ synchronous rounds.
- A self-stabilizing Δ -approximation algorithm for maximum weight independent set that converges in $O(\Delta + \log^* n)$ synchronous rounds.
- A self-stabilizing $((2\rho + 1)(1 + \varepsilon))$ -approximation algorithm for minimum weight dominating set in ρ -arboricity graphs that converges in $O((\log \Delta) / \varepsilon)$ synchronous rounds.

In all of the above, Δ denotes the maximum degree. Our technique improves upon previous results in terms of time complexity while incurring only an additive $O(\log n)$ overhead to the message size. In addition, to the best of our knowledge, we provide the first self-stabilizing algorithms for the weighted versions of minimum vertex cover and maximum independent set.

2012 ACM Subject Classification Theory of computation → Approximation algorithms analysis; Theory of computation → Distributed algorithms

Keywords and phrases self-stabilization, approximation algorithms, primal-dual

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2022.27

Funding This research was supported by VATAT Fund to the Technion Artificial Intelligence Hub (Tech.AI).

Acknowledgements We thank Laurent Feuilloley for a helpful and insightful discussion.

1 Introduction

Distributed networks have become ubiquitous in modern engineering reality. One of the major challenges that arise when dealing with large-scale systems is handling fault recovery. The notion of *self-stabilization* was introduced by Dijkstra [10] to accommodate this challenge. Self-stabilization is characterized by the ability of a distributed system that starts from an arbitrary state to converge into a correct state within a finite time. The initial arbitrary state of the system can capture any finite number of faults, thus making self-stabilization an adaptable fault-tolerance approach.

In the realm of *distributed computing*, classic optimization problems continue to draw much research attention, and new distributed approximation algorithms are always in demand. While an abundance of recent studies have been dedicated to distributed approximation algorithms [3, 4, 11, 14, 23], most of them operate in a *fault-free* environment, i.e., they are assumed to start from some designated initial state.



© Yuval Emek, Yuval Gil, and Noga Harlev;
licensed under Creative Commons License CC-BY 4.0

26th International Conference on Principles of Distributed Systems (OPODIS 2022).

Editors: Eshcar Hillel, Roberto Palmieri, and Etienne Rivière; Article No. 27; pp. 27:1–27:19

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

As a step toward bridging the gap between distributed optimization problems and self-stabilization, in this paper we introduce a new general technique that facilitates the design of self-stabilizing approximation algorithms. We consider distributed algorithms that work in the synchronous *message passing* model. Our technique is based on the *primal-dual* methodology, which is known to be highly useful in the context of approximation algorithms [32]. Given a fault-free approximation algorithm, the technique converts it into a self-stabilizing algorithm with the same approximation and runtime guarantees. Moreover, the conversion induces only an additive $O(\log n)$ overhead to the message size, where n is the number of nodes in the graph. Since the fault-free algorithms used in the context of this paper have a message size of $O(\log n)$ (under common assumptions), we get that all of the self-stabilizing algorithms developed in this paper also have a message size of $O(\log n)$.

In Section 4, we demonstrate the power of our new technique by applying it to three recent fault-free algorithms. This leads to new self-stabilizing approximation algorithms for minimum weight vertex cover, maximum weight independent set, and minimum weight dominating set. To the best of our knowledge, these are the first self-stabilizing algorithms for the weighted versions of minimum vertex cover and maximum independent set, and the first sub-linear time algorithm for minimum weight dominated set.

1.1 Model

Consider an undirected graph $G = (V, E)$ and denote $n = |V|$ and $m = |E|$. For a node $v \in V$, we stick to the convention that $N_G(v)$ denotes the set of v 's neighbors in G and that $\deg_G(v)$ denotes v 's degree in G . When G is clear from context, we may omit it from our notation and use $N(v)$ and $\deg(v)$ instead of $N_G(v)$ and $\deg_G(v)$, respectively. Let $E(v) = \{e \in E : v \in e\}$ denote the set of edges in E incident on node $v \in V$.

Following a common convention in the realm of *distributed graph algorithms*, additional input components such as node/edge weights and edge orientations, are passed to the nodes of graph G by means of an *input assignment* $\ell : V \rightarrow \{0, 1\}^*$ which assigns to each node $v \in V$ an *input label* $\ell(v)$. The input label $\ell(v)$ encodes graph attributes relating to v and its incident edges. Moreover, we assume that $\ell(v)$ includes a *port numbering*, i.e., a bijection between v 's incident edges and the set $\{1, \dots, \deg(v)\}$ of ports. Unless stated otherwise, when we refer to an ordered list $u_1, \dots, u_{\deg(v)}$ of v 's neighbors, it is assumed that the list is ordered by v 's port numbers. We refer to the pair $G_\ell = \langle G, \ell \rangle$ as a *labeled graph*.

In this paper, we focus on algorithms that operate in a *message passing* framework in which the nodes of a given labeled graph G_ℓ are associated with identical state machines that update their state concurrently in synchronous *rounds*. In each round, every node $v \in V$ carries out the following operations: (1) v performs local computation and updates its state as a function of its current state, its input label $\ell(v)$, and possibly random coin tosses; (2) v sends messages to its neighbors; and (3) v receives messages sent to it in the current round by its neighbors. We define the *global state* of G_ℓ to be the n -sized vector encoding the states of all nodes in G .

The state of each node $v \in V$ also includes a designated *output register* $\text{out}(v) \in \{0, 1\}^* \cup \{\perp\}$ in which v maintains its *output*. If $\text{out}(v) = \perp$ we say that v is *undecided*, otherwise, we say that v is *decided*. For a labeled graph G_ℓ , we define a *configuration* of G_ℓ as an n -sized vector $c : V \rightarrow \{0, 1\}^* \cup \{\perp\}$ assigning an output value $c(v)$ to each node $v \in V$. We refer to the 3-tuple $G_{\ell,c} = \langle G, \ell, c \rangle$ consisting of a graph $G = (V, E)$, an input assignment $\ell : V \rightarrow \{0, 1\}^*$, and a configuration $c : V \rightarrow \{0, 1\}^* \cup \{\perp\}$ of G_ℓ , as a *configured graph*.

A *distributed problem* Π is a collection of configured graphs $G_{\ell,c}$. In the context of a distributed problem Π , a labeled graph G_ℓ is said to be *valid* if there exists a configuration c such that $G_{\ell,c} \in \Pi$, in which case we say that c is *feasible* for G_ℓ . Given a distributed problem Π , we may slightly abuse notation and write $G_\ell \in \Pi$ to denote that G_ℓ is valid.

Consider a distributed problem Π . Given a valid labeled graph $G_\ell \in \Pi$, the goal of an algorithm **Alg** for Π is to converge to a feasible configuration within a finite number of rounds, in which case we say that **Alg** is *correct*. When considering an algorithm **Alg** that operates in a *fault-free* environment, the initial state of each node $v \in V$ is assumed to be determined locally by **Alg**. More formally, for each valid labeled graph $G_\ell \in \Pi$, the initial state of each node $v \in V$ is defined to be the value $\text{init}_{\text{Alg}}(\ell(v))$ obtained by a function $\text{init}_{\text{Alg}} : \{0,1\}^* \rightarrow \{0,1\}^*$. In contrast, *self-stabilizing* algorithms do not determine the initial state of the nodes. That is, we say that an algorithm **Alg** for Π is self-stabilizing if for any valid labeled graph $G_\ell \in \Pi$, algorithm **Alg** is guaranteed to converge to a feasible configuration starting from any initial global state. The *runtime* of an algorithm is defined to be the number of rounds required until convergence.

For many distributed problems, the quality of a feasible configuration can be measured by means of an *objective function* that one wishes to minimize/maximize. Formally, we define a *distributed minimization problem* (resp., *distributed maximization problem*) Ψ as a pair $\langle \Pi, f \rangle$, where Π is a distributed problem, and $f : \Pi \rightarrow \mathbb{R}$ is an objective function that assigns an objective value $f(G_{\ell,c})$ to any configured graph $G_{\ell,c} \in \Pi$. For an approximation parameter $\alpha \geq 1$, we say that a configuration c is an α -*approximation* for a valid labeled graph $G_\ell \in \Pi$ if the following conditions hold: (1) $G_{\ell,c} \in \Pi$, i.e., c is feasible (with respect to Π) for G_ℓ ; and (2) $f(G_{\ell,c}) \leq \alpha \cdot f(G_{\ell,c'})$ (resp., $f(G_{\ell,c}) \geq f(G_{\ell,c'})/\alpha$) for any feasible configuration c' . We often use the general term *distributed optimization problem* to refer to distributed minimization problems as well as distributed maximization problems. We say that an algorithm **Alg** α -*approximates* a distributed optimization problem Ψ if it solves the distributed problem $\Pi_{\Psi,\alpha} = \{G_{\ell,c} \mid c \text{ is an } \alpha\text{-approximation for } G_\ell\}$.

1.2 Related Work

The notion of self-stabilization was introduced in the seminal paper of Dijkstra [10] and is studied extensively since then. Special interest is given to self-stabilizing graph algorithms, which have natural applications in distributed systems. Awerbuch and Varghese [2] provided a compiler that transforms deterministic synchronous distributed algorithms into self-stabilizing algorithms with the same running time. Note, however, that this held only under the *LOCAL* model and the size of the node states may be unbounded. See [26] for more details on this compiler.

For the unweighted vertex cover problem, a 2-approximation can be achieved by finding a maximal matching. Hsu and Huang [21] presented a self-stabilizing maximal matching algorithm in the *shared memory* model with running time $O(n^3)$, where n is the number of nodes in the graph. Later, this algorithm was reanalyzed to show that its running time is up-bounded by $O(n^2)$ [29], and then an $O(m+n)$ was shown by [20], where m is the number of edges in the graph. The algorithm of Hsu and Haung assumes *sequential adversary*, which means that exactly one node is scheduled for execution at each round. Gradinariu and Tixeuil [17] provided a general scheme to transform an algorithm under a sequential adversary into an algorithm that works under a *distributed adversary*, which selects a subset of the nodes to be executed at each round. Combined with the algorithm of Hsu and Huang, this scheme yields a time complexity of $O(\Delta m)$, where Δ is the maximum degree of the graph. Chattopadhyay et al. [7] and Manne et al. [27] gave self-stabilizing algorithms for

maximal matching with quadratic runtime in more general models. Cohen et al. [9] proposed a randomized self-stabilizing algorithm for computing a maximal matching with a time complexity of $O(n^2)$ rounds with high probability.

Kiniwa [24] devised a self-stabilizing vertex cover algorithm that achieves a $(2 - 1/\Delta)$ -approximation. This algorithm, which works in the shared memory model, is the first with an approximation ratio less than 2. Turau and Hauck [31] presented a self-stabilizing vertex cover algorithm that computes a $(3 - 2/(\Delta + 1))$ -approximation and stabilizes in $O(n + m)$ rounds.

For the minimal dominating set (MDS) problem, Hedetniemi et al. [19] presented a self-stabilizing algorithm under a sequential adversary with a time complexity of $O(n^2)$. Xu et al. [34] proposed a synchronous MDS self-stabilizing algorithm that converges in $O(n)$ rounds. Self-stabilizing MDS algorithms with a linear time complexity under a distributed adversary are presented in [30, 15, 8]. For the minimum weight dominating set (MWDS) problem, Wang et al. [33] were the first to propose a self-stabilizing algorithm that works for general graphs. Their algorithm converges in $O(n^2)$ rounds under a sequential adversary.

For the unweighted MaxIS problem, one can obtain a Δ -approximation by finding a *maximal independent set (MIS)*. The first self-stabilizing algorithm for the MIS problem was introduced by Shukla et al. [28]. Their algorithm converges in $O(n)$ rounds under a sequential adversary. Under a distributed adversary, Ikeda et al. [22] provided an algorithm that converges in $O(n^2)$ rounds, and Goddard et al. [16] proposed a synchronous algorithm that converges in $O(n)$ rounds. Later, Turau [30] designed the first linear time asynchronous MIS algorithm assuming a distributed adversary. Recently, an improved self-stabilizing linear-time asynchronous MIS algorithm was suggested by Arapoglu and Dagdeviren [1], assuming a distributed adversary as well. Blair and Manne [6] suggested a generic mapping from sequential tree algorithms to self-stabilizing tree algorithms. Among other algorithms, this mapping yields a MaxIS algorithm that requires $O(n^2)$ rounds under the *read-write atomicity* assumption.

We refer the interested reader to [18, 12] for extensive surveys on self-stabilizing algorithms and the different models.

2 Preliminaries

Linear Programming and Duality. A *linear program (LP)* consists of a linear objective function to be optimized (i.e., minimized or maximized) subject to linear inequality constraints. Formally, a minimization (resp., maximization) LP is $\min\{\mathbf{c}^T \mathbf{x} \mid \mathbf{A} \mathbf{x} \geq \mathbf{b} \wedge \mathbf{x} \geq \mathbf{0}\}$ (resp., $\max\{\mathbf{c}^T \mathbf{x} \mid \mathbf{A} \mathbf{x} \leq \mathbf{b} \wedge \mathbf{x} \geq \mathbf{0}\}$), where $\mathbf{x} = \{x_j\} \in \mathbb{R}^s$ is a vector of variables and $\mathbf{A} = \{a_{i,j}\} \in \mathbb{R}^{r \times s}$, $\mathbf{b} = \{b_i\} \in \mathbb{R}^r$, and $\mathbf{c} = \{c_j\} \in \mathbb{R}^s$ are a matrix and vectors of coefficients, respectively. An *integer linear program (ILP)* is an LP with integer variables. An *LP relaxation* of an ILP is the LP obtained from the ILP by relaxing its integrality constraints.

Every LP has a corresponding *dual program*, and in this context, we refer to the original LP as the *primal program*. Specifically, for a minimization (resp., maximization) LP, its dual program is a maximization (resp., minimization) LP, formulated as $\max\{\mathbf{b}^T \mathbf{y} \mid \mathbf{A}^T \mathbf{y} \leq \mathbf{c} \wedge \mathbf{y} \geq \mathbf{0}\}$ (resp., $\min\{\mathbf{b}^T \mathbf{y} \mid \mathbf{A}^T \mathbf{y} \geq \mathbf{c} \wedge \mathbf{y} \geq \mathbf{0}\}$). The following properties of LP duality make it a powerful tool. The *weak duality* theorem states that $\mathbf{c}^T \mathbf{x} \geq \mathbf{b}^T \mathbf{y}$ (resp., $\mathbf{c}^T \mathbf{x} \leq \mathbf{b}^T \mathbf{y}$) for every two feasible solutions \mathbf{x} and \mathbf{y} to the primal and dual programs, respectively. The *strong duality* theorem states that $\mathbf{c}^T \mathbf{x} = \mathbf{b}^T \mathbf{y}$ if and only if \mathbf{x} and \mathbf{y} are optimal primal and dual solutions, respectively. The *relaxed complementary slackness* conditions are stated as follows, for given parameters $\beta, \gamma \geq 1$.

- Primal relaxed complementary slackness:

For every primal variable x_j , if $x_j > 0$, then $c_j/\beta \leq \sum_{i=1}^r a_{ij}y_i \leq c_j$ (resp., $c_j \leq \sum_{i=1}^r a_{ij}y_i \leq \beta \cdot c_j$).

- Dual relaxed complementary slackness:

For every dual variable y_i , if $y_i > 0$, then $b_i \leq \sum_{j=1}^s a_{ij}x_j \leq \gamma \cdot b_i$ (resp., $b_i/\gamma \leq \sum_{j=1}^s a_{ij}x_j \leq b_i$).

If the (primal and dual) relaxed complementary slackness conditions hold, then it is guaranteed that $\mathbf{c}^T \mathbf{x} \leq \beta \cdot \gamma \cdot \mathbf{b}^T \mathbf{y}$ (resp., $\mathbf{c}^T \mathbf{x} \geq \frac{1}{\beta \cdot \gamma} \cdot \mathbf{b}^T \mathbf{y}$). Combined with the weak duality theorem, this means that \mathbf{x} approximates an optimal primal solution by a multiplicative factor of $\beta \cdot \gamma$.

3 Our Technique

In this section, we present a high-level description of our technique for designing self-stabilizing approximation algorithms for a large family of distributed graph optimization problems (henceforth, OptDGPs). We say that an OptDGP Ψ is a *covering* (resp., *packing*) problem if it can be formulated as a minimization (resp., maximization) LP P with a dual LP D such that the variables and constraints of P and D are associated with the nodes and/or edges of the graph.¹ We focus on covering/packing problems that are *locally-constrained* in the sense that a primal/dual constraint associated with a node $v \in V$ or an edge $e \in E$, only involves variables associated with incident nodes and/or edges.

Consider a locally-constrained covering/packing problem Ψ . The technique augments a fault-free distributed α -approximation algorithm \mathbf{Alg} for Ψ with a *local-checking* procedure, resulting in a self-stabilizing distributed α -approximation algorithm $\mathbf{Alg}^{\text{stab}}$. We typically consider a fault-free algorithm \mathbf{Alg} that admits the following structure: (1) \mathbf{Alg} maintains a feasible dual solution \mathbf{y} throughout its execution; (2) \mathbf{Alg} constructs a primal solution \mathbf{x} such that no primal constraint is violated; and (3) throughout its execution, \mathbf{Alg} maintains the property that \mathbf{x} and \mathbf{y} are not "too far" from each other (e.g., by maintaining relaxed complementary slackness conditions).

We now describe the key ideas behind the transformation of \mathbf{Alg} into a self-stabilizing algorithm $\mathbf{Alg}^{\text{stab}}$. At the heart of this transformation, we have the aforementioned local-checking procedure that is invoked repeatedly at the beginning of each round. The local-checking procedure starts from a *detection* step whose goal is to verify the primal and dual feasibility as well as the approximation guarantees of \mathbf{Alg} . To that end, during $\mathbf{Alg}^{\text{stab}}$ each node $v \in V$ keeps track of all the primal and dual variables that appear in the constraints associated with v and its incident edges $E(v)$. We emphasize that this allows each node to perform the detection step locally without communication.

Following the detection step, $\mathbf{Alg}^{\text{stab}}$ branches into one of two possibilities: if the current primal and dual assignments satisfy the detection conditions for a node v , then v proceeds to perform local computation and send messages according to \mathbf{Alg} ; otherwise, v performs a *correction* step. While the details of the correction step are often problem-specific, its common idea is to change the primal and dual variables so that they meet the detection conditions. To preserve consistency between two neighbors v and u regarding their mutual

¹ For simplicity, we assume that each node $v \in V$ knows all of the coefficients of P (and D) that are associated with its neighbors and incident edges (this includes, e.g., node/edge weights, capacities, etc.). We note that this assumption is w.l.o.g. since it can be implemented by means of sending this information through messages at the cost of at most 1 round of communication.

primal and dual variables (i.e., the variables maintained by both v and u), v and u inform each other of the current values assigned to those variables at each round. Following that, upon receiving each other's messages, u and v consistently update their mutual variables (the details of this update are also problem-specific).

Notice that once Alg^{stab} reaches primal and dual assignments that satisfy the detection conditions for every node $v \in V$, it proceeds to construct the primal and dual solution strictly according to Alg . By the correctness of Alg , reaching such assignments guarantees that Alg^{stab} converges to an α -approximation for the OptDGP Ψ . Therefore, the main challenge of Alg^{stab} is to recover from arbitrary primal and dual assignments to primal and dual assignments that satisfy the detection conditions. As we show in Section 4, for some classical covering/packing problems this recovery process can be obtained using only $O(1)$ rounds. Thus, for those problems, Alg^{stab} achieves the same (asymptotic) runtime guarantee as Alg . Moreover, if we stick to the common assumption that all the primal and dual coefficients of the problems mentioned in Section 4 can be represented using $O(\log n)$ bits, then we get that $s_{\text{stab}} = s_{\text{Alg}} + O(\log n)$, where s_{stab} and s_{Alg} denote the message size of Alg^{stab} and Alg , respectively.

4 Results

4.1 Minimum Weight Vertex Cover

Consider a graph $G = (V, E)$ associated with a node-weight function $w : V \rightarrow \mathbb{R}_{\geq 0}$. A *vertex cover* is a set $U \subseteq V$ of nodes such that each edge $e \in E$ has at least one endpoint in U . A *minimum weight vertex cover (MWVC)* is a vertex cover U that minimizes $w(U) = \sum_{u \in U} w(u)$. In a natural LP formulation of MWVC, each node $v \in V$ is associated with a variable x_v and each edge $(u, v) \in E$ is associated with a covering constraint $x_u + x_v \geq 1$. In the dual LP, each edge $e \in E$ is associated with a variable y_e and each node $v \in V$ is associated with a packing constraint $\sum_{e \in E(v)} y_e \leq w(v)$.

In this section, we devise a self-stabilizing $2(1 + \varepsilon)$ -approximation algorithm for MWVC. More concretely, we constructively prove the following theorem.

► **Theorem 4.1.** *There exists a self-stabilizing algorithm that converges to a $2(1 + \varepsilon)$ -approximation for MWVC in $O(\log \Delta / \varepsilon \log \log \Delta)$ rounds.*

Our algorithm involves adapting the (fault-free) algorithm by Bar-Yehuda et al. [4] to an algorithm that works in a primal-dual framework, i.e., an algorithm that constructs primal and dual solutions. We then exploit the properties of valid primal and dual solutions to construct a self-stabilizing algorithm, i.e., an algorithm that is guaranteed to converge to a $2(1 + \varepsilon)$ -approximation for MWVC from an arbitrary global state (and in particular, an arbitrary assignment to the primal and dual variables). Refer to Pseudocode 1 for the full description of the algorithm. We now give a high-level overview of the algorithm.

Overview of the algorithm. Throughout the execution of Algorithm 1, each node $v \in V$ maintains a primal variable $v.x_v \in \{0, 1, \perp\}$ associated with v , where $v.x_v = \perp$ reflects that v is undecided; $v.x_v = 0$ reflects that v is not in the cover; and $v.x_v = 1$ reflects that v is in the cover. Additionally, for each neighbor $u \in N(v)$, v maintains a primal variable $v.x_u \in \{0, 1, \perp\}$ associated with u and a dual variable $v.y_{u,v} \in \mathbb{R}_{\geq 0}$ associated with the edge (u, v) . Each node $v \in V$ also maintains the set $N^{\text{und}}(v)$ consisting of v 's currently undecided neighbors (according to the $v.x_u$ values) and the value $d(v) = |N^{\text{und}}(v)|$. For each neighbor $u \in N(v)$, v also maintains a value $v.d(u)$.

Let us now describe how v operates during a round of Algorithm 1. First, v splits the weight $w(v)$ to $\text{threshold}(v) = w(v)/(1 + \varepsilon)$ and $\text{slack}(v) = w(v) - \text{threshold}(v)$. Then, v performs detection, i.e., it checks whether the current assignment to its variables is faulty, and performs correction if necessary. Specifically, v checks the following conditions in order: (1) primal feasibility, i.e., if $v.x_v = 0$, then v checks that $v.x_u = 1$ for all $u \in N(v)$; (2) dual feasibility, i.e., v checks that $\sum_{e \in E(v)} v.y_e \leq w(v)$; and (3) primal relaxed complementary slackness, i.e., if $v.x_v = 1$, then v checks that $\sum_{e \in E(v)} v.y_e \geq \text{threshold}(v)$. If conditions (1) or (3) fail, then v sets $v.x_v = \perp$; if condition (2) fails, then v sets $v.y_e = 0$ for each $e \in E(v)$.

After detection, v computes the message it sends each neighbor $u \in N(v)$. Every message from v to $u \in N(v)$ first indicates whether v is decided or undecided. To preserve consistency of shared values, each message from v to $u \in N(v)$ contains the current values of $v.x_v, v.y_{u,v}$, and $d(v)$. Upon receiving values $u.x_u, u.y_{u,v}$, and $d(u)$, node v updates its own values by setting $v.x_u = u.x_u$, $v.y_{u,v} = \min\{v.y_{u,v}, u.y_{u,v}\}$, and $v.d(u) = d(u)$.

If v is undecided, then for each undecided neighbor $u \in N^{\text{und}}(v)$, in addition to the values $v.x_v, v.y_{u,v}$, and $d(v)$ node v sends $v.d(u)$ and a real value $\text{budget}(v, u)$. The value $\text{budget}(v, u)$ is determined based on an ordering $u_1, \dots, u_{d(v)}$ of $N^{\text{und}}(v)$ as follows. For each $i \in [d(v)]$, v sets $\text{budget}(v, u_i) = \min\{\text{slack}(u_i)/v.d(u_i), \text{bank}(v) - \sum_{j=1}^{i-1} \text{budget}(v, u_j)\}$, where $\text{bank}(v) = \text{threshold}(v) - \sum_{e \in E(v)} v.y_e$. If v receives a message $\langle \text{budget}(u, v), d(u), u.d(v), u.y_{u,v} \rangle$ from a neighbor $u \in N^{\text{und}}(v)$ that satisfies $d(u) \leq v.d(u)$ and $d(v) \leq u.d(v)$, then v increments the variable $v.y_{u,v}$ by $\text{budget}(u, v) + \text{budget}(v, u)$.

Finally, if v is undecided, then it becomes decided at the beginning of the following round in one of the following cases: if $\sum_{e \in E(v)} v.y_e \geq \text{threshold}(v)$, then v sets $v.x_v = 1$; otherwise, if $v.x_u = 1$ for every neighbor $u \in N(v)$, then v sets $v.x_v = 0$.

Analysis. We now analyze Algorithm 1. Recall that our goal is to establish that Algorithm 1 converges to a $2(1 + \varepsilon)$ -approximation for MWVC in $O(\log \Delta/\varepsilon \log \log \Delta)$ rounds starting from any global state. To that end, let us first state the following straightforward observation that holds trivially by the construction of Algorithm 1.

► **Observation 4.2.** *At the end of each round of Algorithm 1, it holds that $v.x_v = u.x_v$ and $v.y_{u,v} = u.y_{u,v}$ for every $(u, v) \in E$.*

The goal of the following three claims is to show that Algorithm 1 recovers quickly from any global state. As such, these claims play a major role in proving Theorem 4.1.

▷ **Claim 4.3.** *At the end of each round of Algorithm 1, it holds that $\sum_{e \in E(v)} v.y_e \leq w(v)$ for each node $v \in V$.*

Proof. Consider some node $v \in V$ and fix some round $i \geq 1$. Notice that the check in line 8 of Pseudocode 1 guarantees that $\sum_{e \in E(v)} v.y_e \leq w(v)$ right before v receives messages. If $v.x_v \neq \perp$ at the time v receives messages, then v will not increase its $v.y_e$ variables, and thus the claim holds. Now, suppose that $v.x_v = \perp$. Let Y^s and Y^f denote the sum of dual variables $v.y_e$ before and after v updates its dual variables in round i , respectively. Let $N'(v) \subseteq N^{\text{und}}(v)$ be the set of neighbors $u \in N^{\text{und}}(v)$ that send v a message $\langle \text{budget}(u, v), d(u), u.d(v), u.y_{u,v} \rangle$ in the current round such that $d(u) \leq v.d(u)$ and $d(v) \leq u.d(v)$. Notice that $Y^f \leq Y^s + \sum_{u \in N'(v)} \text{budget}(u, v) + \sum_{u \in N'(v)} \text{budget}(v, u)$. By definition, it holds that $\sum_{u \in N'(v)} \text{budget}(v, u) \leq \text{bank}(v) = \text{threshold}(v) - Y^s$. In addition, by the way $\text{budget}(u, v)$ is assigned, and since $u.d(v) \geq d(v)$ for all $u \in N'(v)$, and $|N'(v)| \leq d(v)$, it follows that

$$\sum_{u \in N'(v)} \text{budget}(u, v) \leq \sum_{u \in N'(v)} \frac{\text{slack}(v)}{u.d(v)} \leq \sum_{u \in N'(v)} \frac{\text{slack}(v)}{d(v)} \leq \text{slack}(v).$$

Overall, we have $Y^f \leq \text{threshold}(v) + \text{slack}(v) = w(v)$. \triangleleft

\triangleright **Claim 4.4.** Let $i \geq 2$. At the end of the i -th round of Algorithm 1 it holds that if $v.x_v = 1$, then $\sum_{e \in E(v)} v.y_e \geq \text{threshold}(v)$ for each node $v \in V$.

Proof. Consider some node $v \in V$ during the i -th round for some $i \geq 2$. Notice that the check in line 10 of Pseudocode 1 guarantees that v satisfies $v.x_v = 1 \Rightarrow \sum_{e \in E(v)} v.y_e \geq \text{threshold}(v)$ right before v receives messages. If $v.x_v \neq 1$ at that time, then $v.x_v \neq 1$ at the end of round i and the claim is trivial; so, suppose that $v.x_v = 1$ right before receiving messages. This means that by the end of round i , node v sets $v.y_{u,v} = \min\{v.y_{u,v}, u.y_{u,v}\}$ for each $u \in N(v)$. Notice that by Claim 4.3, every neighbor $u \in N(v)$ does not satisfy the condition in line 8 of the i -th round. This means that the value $u.y_{u,v}$ sent to v from neighbor u during round i does not change from the end of round $i - 1$. From Observation 4.2, it follows that $v.y_{u,v}$ does not change by the end of round i . Therefore, the inequality $\sum_{e \in E(v)} v.y_e \geq \text{threshold}(v)$ is still satisfied by the end round i . \triangleleft

\triangleright **Claim 4.5.** Let $i \geq 3$. At the end of the i -th round of Algorithm 1 it holds that for each node $v \in V$, if $v.x_v = 0$, then $v.x_u = 1$ for every neighbor $u \in N(v)$.

Proof. Consider some node $v \in V$ during the i -th round for some $i \geq 3$. Notice that the check in line 6 of Pseudocode 1 guarantees that if $v.x_v = 0$, then $v.x_u = 1$ for every neighbor $u \in N(v)$ before v receives messages. By Observation 4.2, at the end of round $i - 1$ it holds that $u.x_u = v.x_u = 1$ for all $u \in N(v)$. By Claim 4.4, the value of $u.x_u$ remains 1 for every node $u \in N(v)$ during round $i \geq 3$ (since the condition in line 10 is not satisfied). Therefore, if $v.x_v = 0$, then v receives the message $\langle \text{"DECIDED"}, d(u), u.x_u = 1, u.y_{u,v} \rangle$ from each neighbor $u \in N(v)$ in the i -th round, thus the claim holds at the end of round i . \triangleleft

We are now prepared to prove Theorem 4.1.

Proof of Theorem 4.1. We start with the runtime analysis of Algorithm 1. We note that the runtime analysis uses similar arguments as the analysis presented in [4]. Claims 4.4 and 4.5 imply that if node v is decided (i.e., $v.x_v \neq \perp$) in round $i > 3$, then it will not change its decision at any round $i' \geq i$. We now bound the number of rounds until a node $v \in V$ becomes decided.

Fix some $i > 4$ and node $v \in V$, and suppose that $v.x_v = \perp$ in the i -th round. Let $d_i(v)$ be the value of $d(v)$ in round i (after the update in line 2) and let $Y_i(v)$ be the sum of dual variables $\sum_{e \in E(v)} v.y_e$ at the end of round i . We denote by $u.d_i(v)$ the value of $u.d(v)$ at the beginning of round i for each neighbor $u \in N(v)$. Observe that u updates $u.d(v)$ at the end of round $i - 1$ according to a message from v , and thus $u.d_i(v) = d_{i-1}(v)$. We also note that nodes that are decided at the beginning of round $i - 1$ do not become undecided at any time afterwards. Therefore, it follows that $d_i(v) \leq d_{i-1}(v) = u.d_i(v)$.

We now show that for every parameter $z > 0$, it holds that either (1) $d_{i+2}(v) \leq d_{i-1}(v)/z$; or (2) $Y_i(v) \geq Y_{i-1}(v) + \text{slack}(v)/z$. First, observe that if during the i -th round it holds that $\text{budget}(u, u') < \text{slack}(u')/u.d_i(u')$ for some undecided node $u \in V$ and neighbor $u' \in N^{und}(u)$, then u sets $u.x_u = 1$ during round $i + 1$ and inform its neighbors. Hence, if $d_{i+2}(v) > d_{i-1}(v)/z$, then v updates $Y_i(v)$ according to more than $d_{i-1}(v)/z$ messages

with $\text{budget}(u, v) = \text{slack}(v)/u \cdot d_i(v) = \text{slack}(v)/d_{i-1}(v)$ for every undecided neighbor $u \in N^{\text{und}}(v)$. It follows that $Y_i(v) \geq Y_{i-1}(v) + (d_{i-1}(v)/z) \cdot (\text{slack}(v)/d_{i-1}(v)) = Y_{i-1}(v) + \text{slack}(v)/z$. Recall that v becomes decided in round i' if either $d_{i'}(v) = 0$; or $Y_{i'}(v) \geq \text{threshold}(v)$. By the above, case (2) can occur in at most $z \cdot w(v)/\text{slack}(v)$ rounds until $\sum_{e \in E(v)} y_e \geq \text{threshold}(v)$. As case (1) can occur in at most $\log(\deg(v))/\log z$ rounds, it follows that after

$$\begin{aligned} \frac{z \cdot w(v)}{\text{slack}(v)} + O\left(\frac{\log(\deg(v))}{\log z}\right) &= \frac{z}{1 - 1/(1 + \varepsilon)} + O\left(\frac{\log(\deg(v))}{\log z}\right) \\ &= \frac{z(1 + \varepsilon)}{\varepsilon} + O\left(\frac{\log(\deg(v))}{\log z}\right) \end{aligned}$$

rounds v must be decided. Taking $z = \log(\deg(v))/\log \log(\deg(v))$, we get the desired bound of $O(\log(\deg(v))/\varepsilon \log \log(\deg(v)))$ rounds.

As for the correctness, first notice that by Observation 4.2, it holds at convergence that $v \cdot x_v = u \cdot x_v$ and $v \cdot y_{u,v} = u \cdot y_{u,v}$ for each $(u, v) \in E$. Additionally, by the design of Algorithm 1 and by Claims 4.3, 4.4, and 4.5, the variables' values do not change afterwards. Let $\mathbf{x} = \langle x_v \mid v \in V \rangle \in \{0, 1\}^n$ and $\mathbf{y} = \langle y_e \mid e \in E \rangle \in \mathbb{R}_{\geq 0}^m$ be the primal and dual solutions derived from the variables $v \cdot x_v$ and $v \cdot y_e$, respectively. By relaxed complementary slackness, it is sufficient to show that the following conditions are satisfied: (1) \mathbf{x} is a feasible primal solution; (2) \mathbf{y} is a feasible dual solution; (3) $x_v > 0 \Rightarrow \sum_{e \in E(v)} y_e \geq w(v)/(1 + \varepsilon)$; and (4) $y_{u,v} > 0 \Rightarrow x_u + x_v \leq 2$. Conditions (1), (2), and (3) follow directly from Claims 4.5, 4.3, and 4.4, respectively. Condition (4) holds trivially, since $x_u + x_v \leq 1 + 1 = 2$ for all $(u, v) \in E$. ◀

Message size. Note that the size of messages sent during Algorithm 1 depends on the values of $\text{budget}(v, u)$ computed during its execution. Observe that this dependency is manifested in the $\text{budget}(v, u)$ values themselves as well as the dual values $v \cdot y_{u,v}$. As remarked in [4], each $\text{budget}(v, u)$ value can be modified to be represented using $O(\log n)$ bits without affecting the correctness or the (asymptotic) runtime of the algorithm. The idea is to round each $\text{budget}(v, u)$ value and reduce the $\text{slack}(v)$ values accordingly. We note that applying a similar modification to Algorithm 1 is straightforward. Using this modification, we get messages of size $O(\log n)$.

4.2 Maximum Weight Independent Set

Consider a graph $G = (V, E)$ associated with a node-weight function $w : V \rightarrow \mathbb{R}_{\geq 0}$. An *independent set* is a set $X \subseteq V$ of nodes such that each edge $e \in E$ has at most one endpoint in X . A *maximum weight independent set (MWIS)* is an independent set $X \subseteq V$ that maximizes $w(X) = \sum_{v \in X} w(v)$. In a natural LP formulation of MWIS, each node $v \in V$ is associated with a variable x_v and each edge $(u, v) \in E$ is associated with a packing constraint $x_u + x_v \leq 1$. In the dual LP, each edge $e \in E$ is associated with a variable y_e and each node $v \in V$ is associated with a covering constraint $\sum_{e \in E(v)} y_e \geq w(v)$.

In this section, we present a self-stabilizing algorithm that given a proper $(\Delta + 1)$ -coloring, obtains a Δ -approximation for MWIS. More concretely, we constructively prove the following lemma.

► **Lemma 4.6.** *Given a proper $(\Delta + 1)$ -coloring $c : V \rightarrow \{1, \dots, \Delta + 1\}$, there exists a self-stabilizing algorithm that converges to a Δ -approximation for MWIS in $O(\Delta)$ rounds.*

27:10 Design of Self-Stabilizing Approximation Algorithms via a Primal-Dual Approach

■ **Algorithm 1** A self-stabilizing $2(1 + \varepsilon)$ -approximation algorithm for MWVC. Code for node $v \in V$ in a single round.

```

1: threshold( $v$ ) =  $w(v)/(1 + \varepsilon)$ ; slack( $v$ ) =  $w(v) - \mathbf{threshold}(v)$ 
2:  $N^{und}(v) = \{u \in N(v) \mid v.x_u = \perp\}$ ;  $d(v) = |N^{und}(v)|$  ▷  $v$ 's undecided neighbors
3: if  $v.x_v == \perp$  then
4:   if  $\sum_{e \in E(v)} v.y_e \geq \mathbf{threshold}(v)$  then  $v.x_v = 1$ 
5:   else if  $v.x_u == 1$  for every neighbor  $u \in N(v)$  then  $v.x_v = 0$ 
6: if  $(v.x_v == 0) \wedge (\exists u \in N(v) : v.x_u \neq 1)$  then ▷ checking primal feasibility
7:    $v.x_v = \perp$ 
8: if  $\sum_{e \in E(v)} v.y_e > w(v)$  then ▷ checking dual feasibility
9:    $v.y_e = 0$  for all  $e \in E(v)$ 
10: if  $(v.x_v == 1) \wedge (\sum_{e \in E(v)} v.y_e < \mathbf{threshold}(v))$  then ▷ checking comp. slackness
11:    $v.x_v = \perp$ 
12: if  $v.x_v \in \{0, 1\}$  then
13:   send  $\langle \text{"DECIDED"}, d(v), v.x_v, v.y_{u,v} \rangle$  to each neighbor  $u \in N(v)$ 
14: else
15:   send  $\langle \text{"UNDECIDED"}, d(v), v.y_{u,v} \rangle$  to each neighbor  $u \in N(v) - N^{und}(v)$ 
16:    $\mathbf{bank}(v) = \mathbf{threshold}(v) - \sum_{e \in E(v)} v.y_e$ 
17:   let  $u_1, \dots, u_{d(v)}$  be an ordering of  $N^{und}(v)$  ▷ e.g., by port numbers
18:   for  $i = 1, \dots, d(v)$  do
19:      $\mathbf{slack}(u_i) = (1 - 1/(1 + \varepsilon)) \cdot w(u_i)$ 
20:      $\mathbf{budget}(v, u_i) = \min\{\mathbf{slack}(u_i)/v.d(u_i), \mathbf{bank}(v) - \sum_{j=1}^{i-1} \mathbf{budget}(v, u_j)\}$ 
21:     send  $\langle \mathbf{budget}(v, u_i), d(v), v.d(u), v.y_{u_i,v} \rangle$  to  $u_i$ 
22: for each message  $\mu_u$  received from neighbor  $u \in N(v)$  do
23:   if  $\mu_u == \langle \text{"DECIDED"}, d(u), u.x_u, u.y_{u,v} \rangle$  then
24:      $v.d(u) = d(u)$ ;  $v.x_u = u.x_u$ ;  $v.y_{u,v} = \min\{v.y_{u,v}, u.y_{u,v}\}$ 
25:   if  $\mu_u == \langle \text{"UNDECIDED"}, d(u), u.y_{u,v} \rangle$  then
26:      $v.d(u) = d(u)$ ;  $v.x_u = \perp$ ;  $v.y_{u,v} = \min\{v.y_{u,v}, u.y_{u,v}\}$ 
27:   if  $\mu_u == \langle \mathbf{budget}(u, v), d(u), u.d(v), u.y_{u,v} \rangle$  then
28:      $v.x_u = \perp$ ;  $v.y_{u,v} = \min\{v.y_{u,v}, u.y_{u,v}\}$ 
29:     if  $(u \in N^{und}(v)) \wedge (v.x_v == \perp) \wedge (d(u) \leq v.d(u)) \wedge (d(v) \leq u.d(v))$  then
30:        $v.y_{u,v} = v.y_{u,v} + \mathbf{budget}(u, v) + \mathbf{budget}(v, u)$ 
31:        $v.d(u) = d(u)$ 

```

Our algorithm involves adapting the (fault-free) algorithm by Bar-Yehuda et al. [3] to a primal-dual algorithm, and then applying our technique to obtain a self-stabilizing algorithm. Refer to Pseudocode 2 for a full description of the algorithm. For simplicity of presentation, we assume that each node knows the colors of all its neighbors.

Combining Lemma 4.6 with the self-stabilizing $(\Delta + 1)$ -coloring algorithm by Barenboim et al. [5] (henceforth referred to as the BEG algorithm), we establish the following theorem.²

² We note that the BEG algorithm requires that the nodes' labels include the values of Δ and n , as well as a unique ID.

► **Theorem 4.7.** *There exists a self-stabilizing algorithm that converges to a Δ -approximation for MWIS in $O(\Delta + \log^* n)$ rounds.*

We remark that incorporating the BEG algorithm into Algorithm 2 can be done in a straightforward manner. This requires the nodes to repeatedly check that the current coloring is proper, and correct it according to the BEG algorithm if necessary. As established in [5], the BEG algorithm converges to a proper $(\Delta + 1)$ -coloring in $O(\Delta + \log^* n)$ rounds. The execution of the BEG algorithm is performed in parallel to Algorithm 2 so that after $O(\Delta + \log^* n)$ rounds, the incorporated algorithm performs its updates strictly based on Algorithm 2.

Overview of Algorithm 2. Throughout the execution of Algorithm 2, each neighbor $v \in V$ maintains a primal variable $v.x_v \in \{0, 1\}$ and a dual variable $v.y_{u,v} \in \mathbb{R}_{\geq 0}$ for each node $u \in N(v)$.³ Additionally, v maintains a primal variable $v.x_u$ for each neighbor $u \in N(v)$.

Consider a node $v \in V$ and let $S(v) = \{u \in N(v) \mid c(u) < c(v)\}$ and $L(v) = N(v) - S(v)$. At each round, v updates its primal and dual variables as follows. If $\sum_{u \in S(v)} v.y_{u,v} \geq w(v)$, then v sets $v.x_v = 0$ and $v.y_{u,v} = 0$ for each $u \in L(v)$. Otherwise, v sets $v.y_{u,v} = w(v) - \sum_{u' \in S(v)} v.y_{u',v}$ for each $u \in L(v)$. In the case that $\sum_{u \in S(v)} v.y_{u,v} < w(v)$, node v sets $v.x_v = 0$ if there exists a neighbor $u \in L(v)$ such that $v.x_u = 1$. Otherwise, node v sets $v.x_v = 1$.

At the end of each round, v sends the dual variable $v.y_{u,v}$ to each neighbor $u \in L(v)$, and the primal variable $v.x_v$ to each neighbor $u \in S(v)$. Upon receiving a message $u.y_{u,v}$ (resp., $u.x_u$) from a neighbor $u \in S(v)$ (resp., $u \in L(v)$), node v sets $v.y_{u,v} = u.y_{u,v}$ (resp., $v.x_u = u.x_u$).

Analysis. We now turn to analyze Algorithm 2. To that end, let us first state the following straightforward observation that holds trivially by the construction of Algorithm 2.

► **Observation 4.8.** *Consider a node $v \in V$. At the end of each round of Algorithm 2, it holds that $v.x_u = u.x_u$ for each neighbor $u \in L(v)$; and $v.y_{u,v} = u.y_{u,v}$ for each neighbor $u \in S(v)$.*

The following two claims are used to establish the convergence time of the primal and dual solutions.

► **Claim 4.9.** Fix some node $v \in V$. During the execution of Algorithm 2, the dual variable $v.y_{u,v}$ does not change at any time from the end of round $c(v)$ for every $u \in N(v)$.

Proof. We prove the claim by induction on $c(v) = 1, \dots, \Delta + 1$. For the base of the induction, consider the case where $c(v) = 1$. This means that $S(v) = \emptyset$ and thus v sets $v.y_{u,v} = w(v) - \sum_{u' \in S(v)} v.y_{u',v} = w(v)$ at round 1 for each $u \in N(v)$. By the construction of Algorithm 2, these values do not change afterwards.

Now, suppose that $i = c(v) > 1$. Notice that by Observation 4.8, it holds that $v.y_{u,v} = u.y_{u,v}$ for each $u \in S(v)$ at the beginning of round i . By the induction hypothesis, these variables do not change throughout the execution from round i onward. Notice that for each neighbor $u \in L(v)$, node v sets $v.y_{u,v} = 0$ in the case that $\sum_{u \in S(v)} v.y_{u,v} \geq w(v)$; and $v.y_{u,v} = w(v) - \sum_{u' \in S(v)} v.y_{u',v}$ otherwise. Since the value $v.y_{u',v}$ does not change for each

³ We note that unlike the other algorithms presented in this paper, the dual solution obtained by the dual variables in Algorithm 2 is not necessarily feasible. We elaborate on that in the proof of Lemma 4.6.

27:12 Design of Self-Stabilizing Approximation Algorithms via a Primal-Dual Approach

node $u' \in S(v)$, it follows that the value $v.y_{u,v}$ does not change for every $u \in L(v)$. Overall, we conclude that the value $v.y_{u,v}$ does not change from the end of round $c(v)$ onward for every $u \in S(v) \cup L(v) = N(v)$. \triangleleft

\triangleright **Claim 4.10.** Fix some node $v \in V$. During the execution of Algorithm 2, the primal variable $v.x_v$ does not change at any time from the end of round $2\Delta + 3 - c(v)$.

Proof. We prove the claim by induction on $c(v) = \Delta + 1, \dots, 1$. For the base of the induction, suppose that $c(v) = \Delta + 1$ and consider round $\Delta + 2 = 2\Delta + 3 - c(v)$. Since $c(v) = \Delta + 1$, it follows that $L(v) = \emptyset$ and v sets $v.x_v = 0$ if $\sum_{u \in S(v)} v.y_{u,v} \geq w(v)$; and $v.x_v = 1$ otherwise. By Claim 4.9, the value $v.y_{u,v}$ for each neighbor $u \in S(v)$ does not change throughout the execution from the end of round $\Delta + 1$. Therefore, it follows that the value $v.x_v$ does not change from the end of round $\Delta + 2$ onward.

Let $v \in V$ such that $c(v) < \Delta + 1$, and consider round $i = 2\Delta + 3 - c(v)$. If at the beginning of round i it holds that $\sum_{u \in S(v)} v.y_{u,v} \geq w(v)$, then v sets $v.x_v = 0$. By Claim 4.9, the value of $\sum_{u \in S(v)} v.y_{u,v}$ does not change after round i and thus it follows that $v.x_v = 0$ at all times from round i onward. Now, suppose that $\sum_{u \in S(v)} v.y_{u,v} < w(v)$. Notice that v sets $v.x_v = 0$ if there exists a neighbor $u \in L(v)$ such that $v.x_u = 1$; and $v.x_v = 1$ otherwise. By Observation 4.8, it holds that $v.x_u = u.x_u$ for each $u \in L(v)$ at the beginning of round i . By the induction hypothesis, these variables do not change throughout the execution from round i onward. Hence, the value $v.x_v$ does not change either. \triangleleft

We are now prepared to prove Lemma 4.6.

Proof of Lemma 4.6. From Claims 4.9 and 4.10, we can deduce that Algorithm 2 converges to a primal solution $\mathbf{x} = \langle x_v \mid v \in V \rangle \in \{0, 1\}^n$ derived from the variables $v.x_v$ and a dual solution $\mathbf{y} = \langle y_{(u,v)} \mid (u,v) \in E \rangle \in \mathbb{R}_{\geq 0}^m$ derived from the variables $v.y_{u,v}$ after at most $2\Delta + 2$ rounds. Let $\lambda(v) = \max\{0, w(v) - \sum_{u \in S(v)} y_{u,v}\}$ for each node $v \in V$. Notice that \mathbf{y} is constructed such that $y_{u,v} = \lambda(v)$ for each $u \in L(v)$.

Recall the dual constraint $\sum_{u \in N(v)} y_{u,v} \leq w(v)$ associated with each node v . Notice that \mathbf{y} is constructed such that if $\sum_{u \in S(v)} y_{u,v} < w(v)$ for node $v \in V$, then $y_{u,v} = \lambda(v) = w(v) - \sum_{u' \in S(v)} y_{u',v}$ for each $u \in L(v)$. Thus, the dual constraint is violated only for nodes $v \in V$ such that $L(v) = \emptyset$ and $\sum_{u \in S(v)} y_{u,v} < w(v)$. For the sake of the analysis, we fix the dual feasibility by defining the dual solution \mathbf{y}' as follows. If a node v satisfies $L(v) = \emptyset$ and $\sum_{u \in S(v)} y_{u,v} < w(v)$, then we set the dual value $y'_{z,v} = y_{z,v} + \lambda(v)$ for a single neighbor $z \in S(v)$, and set $y'_{u,v} = y_{u,v}$ for every other neighbor $u \in S(v) - \{z\}$. Otherwise (if $L(v) \neq \emptyset$ or $\sum_{u \in S(v)} y_{u,v} \geq w(v)$), we set the dual value $y'_{u,v} = y_{u,v}$ for every neighbor $u \in S(v)$. It is not hard to see that \mathbf{y}' is a feasible dual solution. In addition, notice that \mathbf{x} is a feasible primal solution since for each node $v \in V$, if $x_v = 1$, then $x_u = 0$ for each neighbor $u \in L(v)$.

Let $X = \{v \mid x_v = 1\}$ be the independent set obtained by Algorithm 2 and consider a node $v \in X$. Let $\mu(v) = \{(u, u') \in E \mid u \in S(v) \wedge u' \in L(u) \wedge x_{u'} = 0\}$ and notice that $y_e = y'_e$ for every edge $e \in \mu_v$. We argue that $\Delta \cdot w(v) \geq \sum_{u \in N(v)} y'_{u,v} + \sum_{e \in \mu(v)} y'_e$ for each $v \in X$. To establish that, first suppose that $L(v) \neq \emptyset$. It holds that

$$\begin{aligned} \sum_{u \in N(v)} y'_{u,v} + \sum_{e \in \mu(v)} y'_e &= \sum_{u \in N(v)} y_{u,v} + \sum_{e \in \mu(v)} y_e = \sum_{u \in L(v)} y_{u,v} + \sum_{u \in S(v)} y_{u,v} + \sum_{e \in \mu(v)} y_e \\ &\leq \sum_{u \in L(v)} y_{u,v} + \sum_{u \in S(v)} \sum_{u' \in L(u)} y_{u,u'} \\ &= |L(v)| \cdot \lambda(v) + \sum_{u \in S(v)} |L(u)| \cdot \lambda(u) \end{aligned}$$

$$\begin{aligned}
&\leq \Delta \left(\lambda(v) + \sum_{u \in S(v)} \lambda(u) \right) \\
&= \Delta \left(w(v) - \sum_{u \in S(v)} y_{u,v} + \sum_{u \in S(v)} \lambda(u) \right) \\
&= \Delta \left(w(v) - \sum_{u \in S(v)} \lambda(u) + \sum_{u \in S(v)} \lambda(u) \right) = \Delta \cdot w(v).
\end{aligned}$$

Now, suppose that $L(v) = \emptyset$. Notice that since $v \in X$, it must hold that $\sum_{u \in S(v)} y_{u,v} < w(v)$. By the definition of \mathbf{y}' , it holds that $y'_{z,v} = y_{z,v} + \lambda(v)$ for a single neighbor $z \in S(v) = N(v)$, and $y'_{u,v} = y_{u,v}$ for every other neighbor $u \in S(v) - \{z\}$. It follows that

$$\begin{aligned}
\sum_{u \in N(v)} y'_{u,v} + \sum_{e \in \mu(v)} y'_e &= \lambda(v) + \sum_{u \in S(v)} y_{u,v} + \sum_{e \in \mu(v)} y_e = w(v) + \sum_{e \in \mu(v)} y_e \\
&\leq w(v) + \sum_{u \in S(v)} \sum_{u' \in L(u) - \{v\}} y_{u,u'} \\
&\leq w(v) + (\Delta - 1) \sum_{u \in S(v)} \lambda(u) \\
&< \Delta \cdot w(v),
\end{aligned}$$

where the last transition holds because $x_v = 1$ implies that $\sum_{u \in S(v)} \lambda(u) < w(v)$.

Observe that for every node $v \notin X$, if $L(v) \cap X = \emptyset$, then it holds that $\sum_{u \in S(v)} y'_{u,v} = \sum_{u \in S(v)} y_{u,v} \geq w(v)$ and thus $y'_{u',v} = y_{u',v} = 0$ for each $u' \in L(v)$. Therefore, it follows that

$$\sum_{e \in E} y'_e = \sum_{v \in V} \sum_{u \in L(v)} y'_{u,v} = \sum_{v \in X} \left(\sum_{u \in N(v)} y'_{u,v} + \sum_{e \in \mu(v)} y'_e \right) \leq \sum_{v \in X} \Delta \cdot w(v) = \Delta \cdot w(X)$$

From the weak duality theorem, we conclude that X is a Δ -approximation for MWIS. \blacktriangleleft

Algorithm 2 A self-stabilizing Δ -approximation algorithm for MWIS given a proper $(\Delta + 1)$ -coloring $c : V \rightarrow [\Delta + 1]$. Code for node $v \in V$ in a single round.

```

1:  $S(v) = \{u \in N(v) \mid c(u) < c(v)\}$   $\triangleright v$ 's neighbors with a smaller color
2:  $L(v) = N(v) - S(v)$   $\triangleright v$ 's neighbors with a larger color
3: if  $\sum_{u \in S(v)} v.y_{u,v} \geq w(v)$  then
4:    $v.x_v = 0$ 
5:    $v.y_{u,v} = 0, \forall u \in L(v)$ 
6: else
7:    $v.y_{u,v} = w(v) - \sum_{u' \in S(v)} v.y_{u',v}, \forall u \in L(v)$   $\triangleright w(v) > \sum_{u' \in S(v)} v.y_{u',v}$ 
8:   if  $\exists u \in L(v) : v.x_u == 1$  then
9:      $v.x_v = 0$ 
10:  else  $v.x_v = 1$ 
11: send  $v.y_{u,v}$  to each neighbor  $u \in L(v)$ 
12: send  $v.x_v$  to each neighbor  $u \in S(v)$ 
13: for each  $u.y_{u,v}$  received from neighbor  $u \in S(v)$  do  $v.y_{u,v} = u.y_{u,v}$ 
14: for each  $u.x_u$  received from neighbor  $u \in L(v)$  do  $v.x_u = u.x_u$ 

```

4.3 Minimum Weight Dominating Set in Bounded Arboricity Graphs

Consider a graph $G = (V, E)$ associated with a node-weight function $w : V \rightarrow \mathbb{R}_{\geq 0}$. Let us denote $N^+(v) = N(v) \cup \{v\}$ for each node v . We naturally extend this notation to node sets and denote $N^+(X) = \bigcup_{v \in X} N^+(v)$ for a node set $X \subseteq V$. A set $X \subseteq V$ of nodes is said to be a dominating set if $N^+(X) = V$. A *minimum weight dominating set (MWDS)* is a dominating set X that minimizes $w(X)$. In a natural LP formulation for MWDS, each node $v \in V$ is associated with a variable x_v and a covering constraint $\sum_{u \in N^+(v)} x_u \geq 1$. In the dual LP, each node $v \in V$ is associated with a variable y_v and a packing constraint $\sum_{u \in N^+(v)} y_u \leq w(v)$.

In this section, we focus on graphs with bounded *arboricity*. The arboricity of graph G is the minimal number ρ for which there exists a partition $E = E_1 \dot{\cup} \dots \dot{\cup} E_\rho$ such that E_i induces a forest for each $i \in [\rho]$. We obtain the following results for MWDS on graphs with arboricity at most ρ .

► **Theorem 4.11.** *There exists a self-stabilizing algorithm that converges to a $((2\rho+1)(1+\varepsilon))$ -approximation for MWDS in graphs with arboricity at most ρ in $O(\log \Delta/\varepsilon)$ rounds.*

Our algorithm is based on the primal-dual algorithm by Dory et al. [11]. We assume w.l.o.g. that every node $v \in V$ knows the value $w_{\min}(u) = \min_{u' \in N^+(u)} \{w(u')\}$ of each of its neighbors $u \in N(v)$. We further assume that for each $u \in V$, the neighbor $\arg \min_{u' \in N^+(u)} \{w(u')\}$ is unique (breaking ties, e.g., by port numbers) and that each node v knows if it is the node that realizes $\arg \min_{u' \in N^+(u)} \{w(u')\}$ for each neighbor $u \in N(v)$. Finally, we assume that the arboricity ρ and the maximum degree Δ are encoded in the label of each node $v \in V$. As remarked in [11], the latter assumption can be lifted by replacing Δ with $\max_{u \in N^+(v)} \{\deg(u)\}$ without affecting the correctness and (asymptotic) runtime of the algorithm. Refer to Pseudocode 3 for a full description of the algorithm.

■ **Algorithm 3** A self-stabilizing $((2\rho+1)(1+\varepsilon))$ -approximation algorithm for MWDS in graphs with arboricity at most ρ . Code for node $v \in V$ in a single round.

```

1:  $\lambda = 1/((2\rho+1)(1+\varepsilon))$ ; reset = FALSE
2: MWDS_update_variables ▷ may change the value of reset
3: MWDS_update_status
4: if reset == TRUE then
5:   send ⟨“RESET”, status(v), v.x_v, v.y_v⟩ to each neighbor  $u \in N(v)$ 
6: else
7:   if status(v) == active then
8:      $v.y_v = (1 + \varepsilon)v.y_v$ 
9:   else if status(v) == waiting then
10:    if  $\forall u \in N(v) : v.\text{status}(u) \neq \text{active}$  then
11:       $v.y_v = (1 + \varepsilon)v.y_v$ 
12:      status(v) = done_waiting
13:    send ⟨status(v), v.x_v, v.y_v⟩ to each  $u \in N(v)$ 
14:    MWDS_receive_messages

```

Overview of the algorithm. Let us first briefly describe the high-level idea of the (fault-free) algorithm presented in [11]. Throughout the execution, the algorithm maintains a feasible dual solution \mathbf{y} and uses it to construct a dominating set X such that at termination, $w(X)$ is within a multiplicative $((2\rho+1)(1+\varepsilon))$ factor from the objective value of \mathbf{y} . This is done in two stages. In the first stage, the algorithm constructs a set X_1 which consists of nodes

■ **Algorithm 4** Procedure `MWDS_update_variables`. Node v updates its variables.

```

1:  $v.y_v = \max\{v.y_v, w_{\min}(v)/(\Delta + 1)\}$   $\triangleright$  setting a lower bound for dual variables
2: if  $\exists u \in N^+(v) : v.\text{status}(u) == \text{done\_waiting}$  then
3:   if  $v = \arg \min_{z \in N^+(u)} \{w(z)\}$  then  $\triangleright$  breaking ties by port numbers
4:      $v.x_v = 1$ 
5: else
6:   if  $\sum_{u \in N^+(v)} v.y_u > w(v)$  then  $\triangleright$  checking dual feasibility
7:      $v.y_u = w_{\min}(u)/(\Delta + 1), v.x_u = 0$  for all  $u \in N^+(v)$ 
8:      $\text{reset} = \text{TRUE}$ 
9:   if  $v.y_v \leq \lambda \cdot w_{\min}(v)$  then
10:    if  $\sum_{u \in N^+(v)} v.y_u < w(v)/(1 + \varepsilon)$  then  $\triangleright$  maintaining primal comp. slackness
11:       $v.x_v = 0$ 
12:    else
13:       $v.x_v = 1$ 
14:    else if  $\exists u \in N(v) : v.\text{status}(u) == \text{active}$  then
15:       $v.y_v = \lambda \cdot w_{\min}(v)$ 

```

■ **Algorithm 5** Procedure `MWDS_update_status`. Node v updates its status.

```

1: if  $v.y_v \leq w_{\min}(v) \cdot \lambda/(1 + \varepsilon)$  then
2:   if  $\sum_{u \in N^+(v)} v.x_u == 0$  then  $\triangleright$  primal constraint is not satisfied
3:      $\text{status}(v) = \text{active}$ 
4:   else
5:      $\text{status}(v) = \text{over}$ 
6: else if  $v.y_v \leq w_{\min}(v) \cdot \lambda$  then
7:   if  $\sum_{u \in N^+(v)} v.x_u == 0$  then
8:      $\text{status}(v) = \text{waiting}$ 
9:   else
10:     $\text{status}(v) = \text{over}$ 
11: else  $\text{status}(v) = \text{done\_waiting}$ 

```

■ **Algorithm 6** Procedure `MWDS_receive_messages`. Node v receives messages from its neighbors.

```

1: for each message  $\mu_u$  received from neighbor  $u \in N(v)$  do
2:   if  $\mu_u == \langle \text{"RESET"}, \text{status}(u), u.x_u, u.y_u \rangle$  then
3:      $v.y_v = w_{\min}(v)/(\Delta + 1); v.y_u = u.y_u; v.x_u = u.x_u; v.\text{status}(u) = \text{status}(u)$ 
4:   else  $\triangleright \mu_u = \langle \text{status}(u), u.x_u, u.y_u, u.y_v \rangle$ 
5:      $v.y_u = u.y_u; v.x_u = u.x_u; v.\text{status}(u) = \text{status}(u)$ 

```

$v \in V$ that satisfy the following two conditions by the end of the stage: (1) $y_u \leq \lambda w_{\min}(u)$ for every $u \in N^+(v)$, where $\lambda = 1/((2\rho + 1)(1 + \varepsilon))$; and (2) $\sum_{u \in N^+(v)} y_u \geq w(v)/(1 + \varepsilon)$. In the second stage, a set X_2 is constructed greedily by having each node $u \in V - N^+(X_1)$ which is not dominated by X_1 , add to X_2 a node $v \in N^+(u)$ that satisfies $w(v) = w_{\min}(u)$. As shown in [11], the set $X = X_1 \cup X_2$ is a $((2\rho + 1)(1 + \varepsilon))$ -approximation of MWDS.

In Algorithm 3, we modify the algorithm of [11] to produce a self-stabilizing algorithm. The challenge of such algorithm is to recover from an arbitrary primal and dual assignment. To that end, each node $v \in V$ maintains a primal variable $v.x_v \in \{0, 1\}$, a dual variable $v.y_v \in \mathbb{R}_{\geq 0}$, and the primal and dual variables $v.x_u$ and $v.y_u$ of each neighbor $u \in N(v)$. In addition, v maintains a variable $\mathbf{status}(v) \in \{active, over, waiting, done_waiting\}$ and the status $v.\mathbf{status}(u)$ of each neighbor $u \in N(v)$.

The role of $\mathbf{status}(v)$ is to reflect the current stage of node v with respect to the variables of its neighbors. For each node $v \in V$, $\mathbf{status}(v) = active$ reflects that $y_v \leq \lambda w_{\min}(v)/(1 + \varepsilon)$ and v is currently not dominated; $\mathbf{status}(v) = waiting$ reflects that $\lambda w_{\min}(v)/(1 + \varepsilon) < y_v \leq \lambda w_{\min}(v)$ and v is currently not dominated; $\mathbf{status}(v) = over$ reflects that $y_v \leq \lambda w_{\min}(v)$ and v is dominated; and $\mathbf{status}(v) = done_waiting$ reflects that $y_v > \lambda w_{\min}(v)$.

At the beginning of each round, each node updates its variables using Procedure 4. This procedure makes sure that dual feasibility is maintained, and also updates the primal variables to achieve similar guarantees to those of [11]. In addition, to enable quick convergence of Algorithm 3, the procedure bounds the dual variables from below by setting $v.y_v = \max\{v.y_v, w_{\min}(v)/(\Delta + 1)\}$.

Following the update of the variables, each node updates its status according to Procedure 5. Then, if the dual constraint of node v was violated in Procedure 4, then v resets the dual variables of its neighbors and informs them by sending a “RESET” message. Nodes that receive a “RESET” message set $v.y_v$ to $w_{\min}(v)/(\Delta + 1)$.

If the dual constraint was not violated, v increases its dual variable by setting $v.y_v = (1 + \varepsilon)v.y_v$ if one of the following cases holds: (1) $\mathbf{status}(v) = active$; or (2) $\mathbf{status}(v) = waiting$ and $v.\mathbf{status}(u) \neq active$ for each $u \in N(v)$. In the latter case, v also sets $\mathbf{status}(v) = done_waiting$. Finally, v informs its neighbors about its status and current values of variables, and uses the messages received to update the values of variables $v.x_u$, $v.y_u$, and $v.\mathbf{status}(u)$ of each neighbor $u \in N(v)$ according to Procedure 6.

Analysis. We now turn to analyze Algorithm 3. To that end, let us first state the following straightforward observation that holds trivially by the construction of Algorithm 3.

► **Observation 4.12.** *At the end of each round of Algorithm 3, it holds that $v.y_v = u.y_v$, $v.x_v = u.x_v$, and $\mathbf{status}(v) = u.\mathbf{status}(v)$ for every $(u, v) \in E$.*

We now establish an important property regarding the dual solution maintained by Algorithm 3.

▷ **Claim 4.13.** Let $i \geq 2$. At the end of the i -th round of Algorithm 3, it holds that $\sum_{u \in N^+(v)} v.y_u \leq w(v)$ for every node $v \in V$.

Proof. Fix some node $v \in V$. First, suppose that $\sum_{u \in N^+(v)} v.y_u \leq w(v)$ at the beginning of round i . By Observation 4.12, at the beginning of round i it holds that $v.y_u = u.y_u$ for every $u \in N(v)$. Notice that v updates the dual variable $v.y_u$ at the end of the round according to the message from neighbor u . Each update increases the dual variable $v.y_u$ by a multiplicative factor of at most $(1 + \varepsilon)$. We argue that this update does not violate the inequality in the statement. Notice that if $v.x_v = 0$, then by the construction of Procedure 4

it follows that $\sum_{u \in N^+(v)} v.y_u < w(v)/(1 + \varepsilon)$ at the beginning of the round. Therefore, an increase of this sum by a multiplicative $(1 + \varepsilon)$ does not exceed $w(v)$. If $v.x_v = 1$, then each neighbor $u \in N(v)$ sets its status to *over* and does not increase $u.y_u$.

Now, suppose that $\sum_{u \in N^+(v)} v.y_u > w(v)$ at the beginning of round i . This means that v sets $v.y_u = w_{\min}(u)/(\Delta + 1)$ for each $u \in N^+(v)$. Therefore, at the end of round i it holds that

$$\sum_{u \in N^+(v)} v.y_u = \sum_{u \in N^+(v)} \frac{w_{\min}(u)}{\Delta + 1} \leq \frac{(\deg(v) + 1) \cdot w(v)}{\deg(v) + 1} = w(v),$$

thus establishing the assertion. \triangleleft

From Claim 4.13, we deduce the following corollary.

► **Corollary 4.14.** *Consider a node $v \in V$. During the execution of Algorithm 3, the value $v.y_u$ does not decrease from round 3 onward for each $u \in N^+(v)$.*

We can now show the following claim.

▷ **Claim 4.15.** *Let $i \geq 3$ and consider a node $v \in V$. If $v.x_v = 1$ at the end of round i , then $v.x_v = 1$ at each round $i' \geq i$.*

Proof. Observe that $v.x_v = 1$ at the end of round i in one of the following cases: (1) there exists a node $u \in N^+(v)$ such that $\mathbf{status}(u) = \mathit{done_waiting}$ and $w_{\min}(u) = w(v)$; or (2) node v satisfies the inequality $\sum_{u \in N^+(v)} v.y_u \geq w(v)/(1 + \varepsilon)$ at the beginning of the i -th round. In case (1), we note that by Corollary 4.14 node u will not decrease its dual variable throughout the execution of Algorithm 3. This means that $\mathbf{status}(u) = \mathit{done_waiting}$ throughout the execution and thus it follows that $v.x_v = 1$. For case (2), by Corollary 4.14 the value of $v.y_u$ does not decrease for all $u \in N^+(v)$ throughout the execution of Algorithm 3. Therefore, the inequality $\sum_{u \in N^+(v)} v.y_u \geq w(v)/(1 + \varepsilon)$ remains satisfied. \triangleleft

We conclude the analysis by proving Theorem 4.11.

Proof of Theorem 4.11. First, observe that by Claim 4.15 and the construction of Procedure 4, if $v.x_v = 1$ for node $v \in V$ at some round $i \geq 3$, then $v.x_v = 1$ from round i onward. To see that Algorithm 3 converges to a dominating set, observe that for each node $v \in V$, if $\mathbf{status}(v) \in \{\mathit{over}, \mathit{done_waiting}\}$ at some round $i \geq 1$, then it follows that there exists a node $u \in N^+(v)$ such that $u.x_u = v.x_u = 1$ by the end of round i . Notice that $v.y_v \geq w_{\min}(u)/(\Delta + 1)$ for each $v \in V$ throughout the execution. It now follows from Claim 4.14 and the design of Algorithm 3 that after $O(\log \Delta/\varepsilon)$ rounds, each node $v \in V$ satisfies $\mathbf{status}(v) \in \{\mathit{over}, \mathit{done_waiting}\}$.

We are now ready to establish the correctness of Algorithm 3. Observe that if $\mathbf{status}(v) \in \{\mathit{over}, \mathit{done_waiting}\}$ for each node $v \in V$, then all the primal and dual variables do not change. Let $\mathbf{x} = \langle x_v \mid v \in V \rangle \in \{0, 1\}^n$ and $\mathbf{y} = \langle y_v \mid v \in V \rangle \in \mathbb{R}_{\geq 0}^n$ be the primal and dual solutions derived from the variables $v.x_v$ and $v.y_v$ after convergence, respectively. Let $X = \{v \mid x_v = 1\}$ be the dominating set obtained by Algorithm 3. We divide the set X into two subsets $X_1 = \{v \in X \mid \forall u \in N^+(v) : y_u \leq \lambda w_{\min}(u)\}$, and $X_2 = X - X_1$.

By the construction of Procedure 4, it follows that $\sum_{u \in N^+(v)} y_u \leq w(v)/(1 + \varepsilon)$ for each node $v \in X_1$. In addition, each node $v \in X_2$ satisfies $v = \arg \min_{z \in N^+(u)} \{w(z)\}$ for some node $u \in V - N^+(X_1)$. As established in [11], these properties imply that $w(X_1) \leq (2\rho + 1)(1 + \varepsilon) \sum_{v \in N^+(X_1)} y_v$ and that $w(X_2) \leq (2\rho + 1)(1 + \varepsilon) \sum_{v \in V - N^+(X_1)} y_v$, thus, $w(X) \leq (2\rho + 1)(1 + \varepsilon) \sum_{v \in V} y_v$. It follows from Claim 4.13 that \mathbf{y} is feasible. The set X is a $(2\rho + 1)(1 + \varepsilon)$ -approximation for MWDS as a consequence of weak duality. \blacktriangleleft

5 Discussion

In this paper, we presented a new approach for designing self-stabilizing approximation algorithms that is based on the properties of primal and dual LPs. Our approach leaves various open questions for future research. A particularly interesting subject in this context is LP-based algorithms that rely on rounding a fractional solution. Due to their usefulness in the fault-free setting (see, e.g., [13, 25]), we advocate for the study of rounding-based approximation algorithms in the self-stabilizing setting.

References

- 1 Ozkan Arapoglu and Orhan Dagdeviren. An asynchronous self-stabilizing maximal independent set algorithm in wireless sensor networks using two-hop information. In *2019 International Symposium on Networks, Computers and Communications (ISNCC)*, pages 1–5. IEEE, 2019.
- 2 Baruch Awerbuch and George Varghese. Distributed program checking: a paradigm for building self-stabilizing distributed protocols. In *FOCS*, volume 91, pages 258–267, 1991.
- 3 Reuven Bar-Yehuda, Keren Censor-Hillel, Mohsen Ghaffari, and Gregory Schwartzman. Distributed approximation of maximum independent set and maximum matching. 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 165–174. ACM, 2017.
- 4 Reuven Bar-Yehuda, Keren Censor-Hillel, and Gregory Schwartzman. A distributed $(2 + \epsilon)$ -approximation for vertex cover in $o(\log \Delta / \epsilon \log \log \Delta)$ rounds. *J. ACM*, 64(3):23:1–23:11, 2017.
- 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.
- 6 Jean RS Blair and Fredrik Manne. Efficient self-stabilizing algorithms for tree networks. In *23rd International Conference on Distributed Computing Systems, 2003. Proceedings.*, pages 20–26. IEEE, 2003.
- 7 Subhendu Chattopadhyay, Lisa Higham, and Karen Seyffarth. Dynamic and self-stabilizing distributed matching. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 290–297, 2002.
- 8 Well Y Chiu, Chiuyuan Chen, and Shih-Yu Tsai. A $4n$ -move self-stabilizing algorithm for the minimal dominating set problem using an unfair distributed daemon. *Information Processing Letters*, 114(10):515–518, 2014.
- 9 Johanne Cohen, Jonas Lefevre, Khaled Maâmra, Laurence Pilard, and Devan Sohier. A self-stabilizing algorithm for maximal matching in anonymous networks. *Parallel Processing Letters*, 26(04):1650016, 2016.
- 10 Edsger W Dijkstra. Self-stabilization in spite of distributed control. In *Selected writings on computing: a personal perspective*, pages 41–46. Springer, 1982.
- 11 Michal Dory, Mohsen Ghaffari, and Saeed Ilchi. Near-optimal distributed dominating set in bounded arboricity graphs. In Alessia Milani and Philipp Woelfel, editors, *PODC '22: ACM Symposium on Principles of Distributed Computing, Salerno, Italy, July 25–29, 2022*, pages 292–300. ACM, 2022.
- 12 Swan Dubois and Sébastien Tixeuil. A taxonomy of daemons in self-stabilization. *arXiv preprint*, 2011. [arXiv:1110.0334](https://arxiv.org/abs/1110.0334).
- 13 Manuela Fischer. Improved deterministic distributed matching via rounding. *Distributed Computing*, 33, June 2020.
- 14 Mohsen Ghaffari and Goran Zuzic. Universally-optimal distributed exact min-cut. In Alessia Milani and Philipp Woelfel, editors, *PODC '22: ACM Symposium on Principles of Distributed Computing, Salerno, Italy, July 25–29, 2022*, pages 281–291. ACM, 2022.
- 15 Wayne Goddard, Stephen T Hedetniemi, David P Jacobs, Pradip K Srimani, and Zhenyu Xu. Self-stabilizing graph protocols. *Parallel Processing Letters*, 18(01):189–199, 2008.

- 16 Wayne Goddard, Stephen T Hedetniemi, David Pokrass Jacobs, and Pradip K Srimani. Self-stabilizing protocols for maximal matching and maximal independent sets for ad hoc networks. In *Proceedings International Parallel and Distributed Processing Symposium*, pages 14–pp. IEEE, 2003.
- 17 Maria Gradinariu and Sébastien Tixeuil. Conflict managers for self-stabilization without fairness assumption. In *27th International Conference on Distributed Computing Systems (ICDCS'07)*, pages 46–46. IEEE, 2007.
- 18 Nabil Guellati and Hamamache Kheddouci. A survey on self-stabilizing algorithms for independence, domination, coloring, and matching in graphs. *Journal of Parallel and Distributed Computing*, 70(4):406–415, 2010.
- 19 Sandra M Hedetniemi, Stephen T Hedetniemi, David P Jacobs, and Pradip K Srimani. Self-stabilizing algorithms for minimal dominating sets and maximal independent sets. *Computers & Mathematics with Applications*, 46(5-6):805–811, 2003.
- 20 Stephen T Hedetniemi, David P Jacobs, and Pradip K Srimani. Maximal matching stabilizes in time $o(m)$. *Information Processing Letters*, 80(5):221–223, 2001.
- 21 Su-Chu Hsu and Shing-Tsaan Huang. A self-stabilizing algorithm for maximal matching. *Information processing letters*, 43(2):77–81, 1992.
- 22 Michiyo Ikeda, Sayaka Kamei, and Hirotsugu Kakugawa. A space-optimal self-stabilizing algorithm for the maximal independent set problem. In *the Third International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 70–74, 2002.
- 23 Ken-ichi Kawarabayashi, Seri Khoury, Aaron Schild, and Gregory Schwartzman. Improved distributed approximations for maximum independent set. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, volume 179 of *LIPICs*, pages 35:1–35:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020.
- 24 Jun Kiniwa. Approximation of self-stabilizing vertex cover less than 2. In *Symposium on Self-Stabilizing Systems*, pages 171–182. Springer, 2005.
- 25 Fabian Kuhn and Rogert Wattenhofer. Constant-time distributed dominating set approximation. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, PODC '03, pages 25–32. Association for Computing Machinery, 2003.
- 26 Christoph Lenzen, Jukka Suomela, and Roger Wattenhofer. Local algorithms: Self-stabilization on speed. In *Symposium on Self-Stabilizing Systems*, pages 17–34. Springer, 2009.
- 27 Fredrik Manne, Morten Mjelde, Laurence Pilard, and Sébastien Tixeuil. A new self-stabilizing maximal matching algorithm. *Theoretical Computer Science*, 410(14):1336–1345, 2009.
- 28 Sandeep K Shukla, Daniel J Rosenkrantz, S Sekharipuram Ravi, et al. Observations on self-stabilizing graph algorithms for anonymous networks. In *Proceedings of the second workshop on self-stabilizing systems*, volume 7, page 15. University of Nevada LasVegas, 1995.
- 29 Gerard Tel. Maximal matching stabilizes in quadratic time. *Information Processing Letters*, 49(6):271–272, 1994.
- 30 Volker Turau. Linear self-stabilizing algorithms for the independent and dominating set problems using an unfair distributed scheduler. *Information Processing Letters*, 103(3):88–93, 2007.
- 31 Volker Turau and Bernd Hauck. A fault-containing self-stabilizing $(3 - 2\delta + 1)$ -approximation algorithm for vertex cover in anonymous networks. *Theoretical computer science*, 412(33):4361–4371, 2011.
- 32 Vijay V. Vazirani. *Approximation algorithms*. Springer, 2001.
- 33 Guangyuan Wang, Hua Wang, Xiaohui Tao, and Ji Zhang. A self-stabilizing protocol for minimal weighted dominating sets in arbitrary networks. In *Proceedings of the 2013 IEEE 17th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, pages 496–501. IEEE, 2013.
- 34 Zhenyu Xu, Stephen T Hedetniemi, Wayne Goddard, and Pradip K Srimani. A synchronous self-stabilizing minimal domination protocol in an arbitrary network graph. In *International Workshop on Distributed Computing*, pages 26–32. Springer, 2003.

Self-Stabilizing Clock Synchronization in Dynamic 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 periodic clock synchronization in a synchronous multi-agent system. Each agent holds a clock with an arbitrary initial value, and clocks must eventually be congruent, modulo some positive integer P . Previous algorithms worked in static networks with drastic connectivity properties and assumed that global informations are available at each node. In this paper, we propose a finite-state algorithm for time-varying topologies that does not require any global knowledge on the network. The only assumption is the existence of some integer D such that any two nodes can communicate in each sequence of D consecutive rounds, which extends the notion of strong connectivity in static network to dynamic communication patterns. The smallest such D is called the *dynamic diameter* of the network. If an upper bound on the diameter is provided, then our algorithm achieves synchronization within $3D$ rounds, whatever the value of the upper bound. Otherwise, using an adaptive mechanism, synchronization is achieved with little performance overhead. Our algorithm is parameterized by a function g , which can be tuned to favor either time or space complexity. Then, we explore a further relaxation of the connectivity requirement: our algorithm still works if there exists a positive integer R such that the network is rooted over each sequence of R consecutive rounds, and if eventually the set of roots is stable. In particular, it works in any rooted static network.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Theory of computation → Dynamic graph algorithms

Keywords and phrases Self-stabilization, Clock synchronization, Dynamic networks

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2022.28

Supplementary Material *Software (Source Code)*: https://gitlab.com/bossuet/sap_execution

Acknowledgements We would like to thank Patrick Lambein-Monette, Stephan Merz, and Guillaume Prémel for very useful discussions. We are also indebted to Paolo Boldi and Sebastiano Vigna for their deep and inspiring work on self-stabilization.

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 informations, and robust against unexpected changes in topology.

A canonical problem in distributed control is the *mod P -synchronization problem*: In a system where each agent is equipped with a local discrete clock, the objective is that all clocks are eventually congruent modulo some integer P , despite arbitrary initializations. This synchronization problem arises in a number of applications, both in engineering and natural systems. It is a basic block in many engineering systems, e.g., in the universal self-stabilizing algorithm developed by Boldi and Vigna [8], or for deploying distributed algorithms structured



© Bernadette Charron-Bost and Louis Penet de Monterno;
licensed under Creative Commons License CC-BY 4.0

26th International Conference on Principles of Distributed Systems (OPODIS 2022).

Editors: Eshcar Hillel, Roberto Palmieri, and Etienne Rivière; Article No. 28; pp. 28:1–28:17

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

into synchronized *phases* (e.g., the *Two-Phase* and *Three-Phase Commit* algorithms [6], or many consensus algorithms [5, 15, 22, 11]). Periodic clock synchronization also corresponds to an 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 [23], synchronous flashing fireflies, collective synchronization of pancreatic beta cells [20].

Our goal is the design of distributed algorithms achieving mod P -synchronization in a networked system of n agents that operate in synchronous rounds and communicate by broadcast. The network is supposed to be uniform and anonymous, i.e., agents are identical and have no identifiers. We consider the self-stabilization model where the initial state of each agent is arbitrary. In particular, agents do not have a consistent numbering of the rounds. Moreover, agents may use only local informations.

The communication pattern at each round is modeled by a directed graph that may change continually from one round to the next. In other words, we allow for time-varying communication graphs, which is important if we want to take into account link failure and link creation, reconfigurable networks, or for dealing with probabilistic communication models like the rumor spreading models. We impose weak assumptions on the communication topology; in particular, we allow for non-bidirectional links and do not assume full connectivity. Even the assumption of strong connectivity may be too restrictive in various settings: for instance, asynchrony and benign failures in a fully connected network may be handled in the model of this paper (i.e., synchronous and non-faulty networks) by dynamic graphs that are permanently rooted, but not strongly connected [11].

Contribution. Our contribution in this paper is a finite state algorithm, called *SAP* (for self-adaptive period), that synchronizes periodic clocks in a large class of dynamic networks. As opposed to most of previous solutions, our algorithm does not assume any global knowledge on the network, and tolerates time-varying topologies.

First, we show that the *SAP* algorithm solves the mod P -synchronization problem in any dynamic network with a finite *dynamic diameter*, i.e., 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 .¹ If a bound on the diameter is given, its stabilization time is bounded above by three times the diameter, whatever the value of the bound. However, the *SAP* algorithm fundamentally works when no bound is available, with a limited increase of stabilization time.

Interestingly, the *SAP* algorithm unifies several seemingly different algorithms for the synchronization of periodic clocks in static networks, including the algorithms in [3, 19, 9] and the one deployed in the finite-state universal self-stabilizing protocol in [8], with useful insights for improving their solvability powers. In particular, we show that the pioneer algorithm proposed by Arora et al. [3] works for a period $P \geq 6n$ while the authors proved its correctness only when $P \geq n^2$.

Then we study how to relax the assumption of a finite diameter and introduce the class of *strongly centered networks*: Roughly speaking, a strongly centered network corresponds to a dynamic graph containing at least one *central node*, that is, a node that can reach any nodes through a temporal path of bounded length. Moreover, in such a network, non-central nodes are not allowed to infinitely often communicate with central nodes. This class strictly contains all the dynamic networks with a finite dynamic diameter and all the static rooted networks. Thus the property of a strongly centered network allows for non-strong connectivity while authorizing dynamic links. We prove that the *SAP* algorithm still works in this class

¹ Observe that the diameter of a static strongly connected network is less than the number of agents, while it may be arbitrarily large for a dynamic network. This is why the assumption of a bound on the diameter available at each agent may be quite problematic in the dynamic setting.

of dynamic networks. Once again, no global knowledge is assumed. In particular, neither the bound on the length of the paths (for the central nodes to communicate with all nodes) nor the set of central nodes are supposed to be known. Finally, we provide upper bounds on the stabilization time and the space complexity of each execution of the *SAP* algorithm.

Related work. Self-stabilizing clocks have been extensively studied in different communication models, under different assumptions, and with various problem specifications. 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 [16, 18].

The point of periodic clocks is the use of finite memory, as opposed to unbounded clocks which inherently require infinite memory. This is why the use of a synchronization algorithm for unbounded clocks with a modulo operation at each round is not appropriate for synchronizing periodic clocks. In addition to strong connectivity and static networks, the pioneer papers on periodic clock synchronization [3, 19, 9, 1] all assume that a bound on the diameter is available. To the best of our knowledge, only the synchronization algorithm in [8] for a *static* communication graph dispenses with the latter assumption.

More recently, periodic clock synchronization has been studied in the *Beeping model* [12] in which agents have severely limited communication capabilities: given a 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. [17], which is optimal both in time and space, but which, unfortunately, requires that a bound on the network size is available for each agent.²

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 [13, 14]). They typically require a bidirectional connected (most of the time fully connected) network.

Clock synchronization has also been studied in the model of *population protocols* [2], consisting of a set of agents, interacting in randomly chosen pairs. In this model, the underlying network is assumed to be fully connected, and the pairwise interactions are modeled by bidirectional links. Moreover, only stabilization with probability one or with high probability is required. The same weakening of problem specification is considered for another popular probabilistic communication model, namely the *PULL* model [21], in which, at each round each agent interacts with one random incoming neighbor in a fixed directed graph G . Unfortunately, in addition to a probabilistic weakening of the problem, the self-stabilizing clock synchronization algorithms developed in this model [7, 4] highly rely on the assumption that G is the fully connected graph.

2 Preliminaries

2.1 The computing model

We consider a networked system with a fixed and finite set V of agents. We assume a round-based computational model in the spirit of the Heard-Of model [11]. Point-to-point communications are organized into *synchronized rounds*: each node sends messages to all

² In [17], Feldmann et al. also proposed an algorithm that does not use any bound on the network size, but that only tolerates asynchronous starts.

nodes and receives messages sent by *some* of the nodes. Rounds are communication closed in the sense that no node receives messages in round t that are sent in a round different from t . Communication at each round t is thus modeled by a directed graph (digraph) $\mathbb{G}(t) = (V, E_t)$: $(i, j) \in E_t$ iff communication from i to j is enabled at round t . There is a self-loop at each node i in all the digraphs $\mathbb{G}(t)$ as i communicates with itself instantaneously. The sequence of digraphs $\mathbb{G} = (\mathbb{G}(t))_{t \geq 1}$ is called a *dynamic graph*.

An *algorithm* \mathcal{A} is given by a set \mathcal{Q} of states, a set of messages \mathcal{M} , a sending function $\sigma : \mathcal{Q} \rightarrow \mathcal{M}$, and a transition function $\delta : \mathcal{Q} \times \mathcal{M}^\oplus \rightarrow \mathcal{Q}$, where \mathcal{M}^\oplus is the set of finite multisets over \mathcal{M} .

We consider the *self-stabilization* model where all the nodes start to run the algorithm at round one but their initial states are arbitrary in the set \mathcal{Q} . An *execution* of \mathcal{A} with the dynamic graph \mathbb{G} proceeds as follows: In round t ($t = 1, 2, \dots$), every 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 $\mathbb{G}(t)$, and finally applies the transition function δ to its current state and the list of messages it has just received to go to a next state. Given an execution of \mathcal{A} , the value of any 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

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 $t' \geq t \geq 1$, we let $\mathbb{G}(t : t') \stackrel{\text{def}}{=} \mathbb{G}(t) \circ \dots \circ \mathbb{G}(t')$. 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 $\text{In}_i(t : t')$, and simply by $\text{In}_i(t)$ 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 $i = k_0, k_1, \dots, k_{t'-t+1} = j$ such that (k_r, k_{r+1}) is an edge of $\mathbb{G}(t + r)$ for each $r = 0, \dots, t' - t$.

The *eccentricity* of a node i in a dynamic graph \mathbb{G} , denoted $e_{\mathbb{G}}(i)$, is defined as

$$e_{\mathbb{G}}(i) \stackrel{\text{def}}{=} \inf\{d \in \mathbb{N}^+ \mid \forall t \in \mathbb{N}^+, \forall j \in V : (i, j) \text{ is an edge in } \mathbb{G}(t : t + d - 1)\}.$$

The *dynamic diameter* of \mathbb{G} is then defined as:

$$\text{diam}(\mathbb{G}) \stackrel{\text{def}}{=} \sup_{i \in V} e_{\mathbb{G}}(i).$$

The notion of dynamic diameter generalizes the classical one of diameter of a digraph in the sense that $\text{diam}(\mathbb{G}) = \text{diam}(G)$ if for each positive integer t , $\mathbb{G}(t) = G$. As no confusion can arise, $\text{diam}(\mathbb{G})$ will simply be called the *diameter* of \mathbb{G} .

3 The SAP_g algorithm

3.1 Informal description and pseudo-code

Let \mathcal{A} be an algorithm where each node i maintains an *integer* variable C_i , called the clock of i . Given a positive integer P , an execution of \mathcal{A} is said to *achieve mod P -synchronization* if

$$\exists t_0, c, \forall t \geq t_0, \forall i \in V, C_i(t) \equiv_P t + c,$$

where $C_i(t)$ denotes the value of C_i at the end of round t in the execution. Even in the case of a static strongly connected network, the naive algorithm in which, at each round, each node sends its own C_i and applies the following update rule:

$$C_i \leftarrow \left[\min_{j \in S} C_j + 1 \right]_P,$$

(where S is the set of i 's incoming neighbors, and $[c]_P$ denotes the remainder of the Euclidean division of c by P) does not work when the network diameter is too large compared to the period P . Theorem 10 provides an execution in which such a system never achieves mod P -synchronization. To overcome this problem, we present an algorithm, called *SAP* (for self-adaptive period), inspired by the ideas developed by Boldi and Vigna for their finite-state universal self-stabilizing algorithm [8]. The key point of the *SAP* algorithm lies in the fact that for any positive integer M , we have

$$\left[[c]_{PM} \right]_P = [c]_P.$$

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 there is “enough space” between the largest clock value and the shortest period PM_i in the network. The update rule for M_i is parametrized by a non-decreasing function $g : \mathbb{N} \rightarrow \mathbb{N}$. 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, $C_i(0) < PM_i(0)$.

Let $g : \mathbb{N} \rightarrow \mathbb{N}$ be a non-decreasing function. If q is a positive integer, g^q denotes the q -th iterate of g , and g^0 is the identity function. For every non-negative integer m , we let

$$g^*(m) \stackrel{\text{def}}{=} \inf\{q \in \mathbb{N} \mid g^q(0) \geq m\}.$$

■ **Algorithm 1** The *SAP_g* algorithm, pseudo-code of node i .

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 $\langle C_{j_1}, M_{j_1} \rangle, \langle C_{j_2}, M_{j_2} \rangle, \dots$ from the set S of incoming neighbours
 - 5: $C_i \leftarrow \left[\min_{j \in S} C_j + 1 \right]_{PM_i}$
 - 6: $M_i \leftarrow \max_{j \in S} M_j$
 - 7: **if** $C_j \not\equiv_P C_{j'}$ for some $j, j' \in S$ **then**
 - 8: $M_i \leftarrow g(M_i)$
 - 9: **end if**
-

3.2 Notation and basic invariants

We fix an execution of *SAP_g* with the dynamic graph \mathbb{G} , and for any $t \in \mathbb{N}$, we let

$$\tilde{M}(t) \stackrel{\text{def}}{=} \min_{i \in V} M_i(t).$$

28:6 Self-Stabilizing Clock Synchronization in Dynamic Networks

For each round t in this execution, let i_t^+ denote any one of i 's incoming neighbours in $\mathbb{G}(t)$ that satisfies

$$C_{i_t^+}(t-1) = \min_{j \in \text{In}_i(t)} C_j(t-1).$$

Given an integer $P > 0$, the system is said to be *synchronized* (for *mod P-synchronized*) in round t if

$$\forall i, j \in V, C_i(t) \equiv_P C_j(t).$$

We start with two preliminary lemmas.

► **Lemma 1.**

1. If the system is synchronized in round s , then it is synchronized in any round $t \geq s$.
2. If (i, j) is an edge in $\mathbb{G}(s : t)$, then $C_j(t) \leq C_i(s-1) + t - s + 1$.
3. Each variable M_i is non-decreasing.

The second claim of Lemma 1 simply follows from the fact that $C_i(t+1) \leq C_j(t) + 1$ for every round t and every pair of nodes $i \in V$ and $j \in \text{In}_i(t)$. The first and third claims directly follow from the transition function.

► **Lemma 2.** For each round $t \geq 1$ and each $i \in V$, one of the following statements is true:

1. $C_i(t)$ is positive and $C_i(t) = 1 + C_{i_t^+}(t-1)$
2. $C_i(t) = 0$, $C_i(t-1) = PM_i(t-1) - 1$, and $i_t^+ = i$.

Proof. The lemma just relies on the following series of inequalities:

$$C_{i_t^+}(t-1) \leq C_i(t-1) \leq PM_i(t-1) - 1.$$

The last inequality is clear for $t = 1$, and for $t \geq 2$, it is a consequence of $C_i(t-1) \leq PM_i(t-2) - 1$ and of the fact that M_i is non-decreasing. If one of those inequalities is strict, then assertion 1 in the lemma holds. Otherwise, assertion 2 holds. ◀

Given any node i and two rounds t and t' , we introduce the set

$$S_i^t(t') \stackrel{\text{def}}{=} \{j \in V \mid C_j(t') \equiv_P C_i(t) + t' - t\},$$

and the integer

$$\tilde{M}_i^t(t') \stackrel{\text{def}}{=} \inf_{j \notin S_i^t(t')} M_j(t').$$

Intuitively, $S_i^t(t')$ is the set of nodes whose clocks in round t' are “in accordance” with i 's clock at round t . In each round t' , V may be partitioned into subsets of nodes whose clocks are all congruent mod P , and each $S_i^t(t')$ is either empty, or is equal to one part of this partition. Once the system is synchronized, this partition contains only one part. Clearly, i belongs to $S_i^t(t)$, but i may not belong to $S_i^t(t')$ when $t' \neq t$.

► **Lemma 3.**

1. If $j \in S_i^t(t'+1)$, then $j_{t'+1}^+ \in S_i^t(t')$.
2. If $t \geq t'$, then $S_i^t(t') \neq \emptyset$.
3. $\tilde{M}_i^t(t'+1) \geq \tilde{M}_i^t(t')$.

Proof. The first claim (1) is a direct consequence of the definition of $j_{t'+1}^+$. Then, using (1), we demonstrate the second claim by induction on $t - t' \geq 0$. Finally, the pseudo-code of SAP_g implies $M_i(t'+1) \geq M_{i_{t'+1}^+}(t')$, and hence, $\tilde{M}_i^t(t'+1) \geq \tilde{M}_i^t(t')$. ◀

3.3 Correctness proof with a finite diameter

We fix a dynamic graph \mathbb{G} whose diameter is finite and an execution of SAP_g with \mathbb{G} , and we let $\text{diam}(\mathbb{G}) = D$.

► **Lemma 4.** *Let ℓ be any round. Let i_0 be a node whose clock value is minimum in some round t , and let δ be any integer that is greater or equal to $e_{\mathbb{G}}(i_0)$. One of the following statements is true:*

1. *there exist a round $d \in \{1, \dots, \delta - 1\}$ and a node $i \notin S_{i_0}^{\ell}(t + d)$ such that $C_i(t + d) = 0$;*
2. *the system is synchronized in round $t + \delta$.*

Proof. Let us assume that the first proposition does not hold. First, we prove by induction on $d \in \{1, \dots, \delta - 1\}$, that

$$\forall i \notin S_{i_0}^{\ell}(t + d), \quad C_i(t + d) = d + \min_{j \in \text{In}_i(t+1:t+d)} C_j(t). \quad (1)$$

The base case $d = 1$ is an immediate consequence of Lemma 2. For the inductive step, assume that Eq. (1) holds for some $d \in \{1, \dots, \delta - 1\}$. For every node $i \notin S_{i_0}^{\ell}(t + d + 1)$, we have

$$\begin{aligned} C_i(t + d + 1) &= 1 + \min_{j \in \text{In}_i(t+d+1)} C_j(t + d) \\ &= 1 + d + \min_{j \in \text{In}_i(t+d+1)} \left(\min_{k \in \text{In}_j(t+1:t+d)} C_k(t) \right) \\ &= 1 + d + \min_{k \in \text{In}_i(t+1:t+d+1)} C_k(t). \end{aligned}$$

The first equality is a direct consequence of Lemma 2. The second one is by the inductive hypothesis applied to i_{t+d+1}^+ . Notice that $i_{t+d+1}^+ \notin S_{i_0}^{\ell}(t + d)$ since $i \notin S_{i_0}^{\ell}(t + d + 1)$. The third one is because $\mathbb{G}(t + 1 : t + d + 1) = \mathbb{G}(t + 1 : t + d) \circ \mathbb{G}(t + d + 1)$. This completes the proof of Eq. (1) for every integer $d \in \{1, \dots, \delta - 1\}$.

Then for each node $i \notin S_{i_0}^{\ell}(t + \delta)$, we get

$$\begin{aligned} C_i(t + \delta) &= \left[1 + \min_{j \in \text{In}_i(t+\delta)} C_j(t + \delta - 1) \right]_{PM_i(t+\delta-1)} \\ &= \left[\delta + \min_{j \in \text{In}_i(t+\delta)} \left(\min_{k \in \text{In}_j(t+1:t+\delta-1)} C_k(t) \right) \right]_{PM_i(t+\delta-1)} \\ &= \left[\delta + \min_{k \in \text{In}_i(t+1:t+\delta)} C_k(t) \right]_{PM_i(t+\delta-1)} \\ &= [\delta + C_{i_0}(t)]_{PM_i(t+\delta-1)} \end{aligned}$$

The first equality is by line 5. The second equality is due to Eq. (1) at round $t + \delta - 1$ and the fact that if $i \notin S_{i_0}^{\ell}(t + \delta)$ implies that $i_{t+\delta}^+ \notin S_{i_0}^{\ell}(t + \delta - 1)$. The fourth one is a consequence of the definition of i_0 and $e_{i_0}(\mathbb{G}) \leq \delta$. It follows that all the clocks $C_i(t + \delta)$ are equal modulo P , i.e., the system is synchronized in round $t + \delta$. ◀

Using the assumption of a finite diameter D , we then derive the following lemma.

► **Lemma 5.** *Let t be a round in which $C_i(t) + D \leq PM_i(t)$ holds for each node i . Then the system is synchronized in round $t + D$.*

Proof. Let i be any node, and let $d \in \{1, \dots, D-1\}$. We have

$$\begin{aligned} C_i(t+d-1) &\leq d-1 + C_i(t) \\ &< D-1 + C_i(t) \\ &\leq PM_i(t) - 1 \\ &\leq PM_i(t+d-1) - 1. \end{aligned}$$

The first and fourth inequalities are direct consequences of Lemma 1, and the third inequality is the assumption of the lemma. By Lemma 2, it follows that $C_i(t+d) \neq 0$. Since D is greater or equal to each $e_{\mathbb{G}}(i)$, Lemma 4 shows that the system is synchronized in round $t+D$. ◀

The next lemma focuses on the self-adaptive period mechanism in the SAP_g algorithm. It intuitively states that, as long as all nodes hear of at least one node whose clock is “in accordance” with i ’s clock at round t , every node j not in accordance with $C_i(t)$ increases its M_j variable.

► **Lemma 6.** *Let t, q and δ be three positive integers, and let $i \in V$. If it holds that*

$$\forall j \in V, \forall \ell \leq (q-1)\delta, \quad \text{In}_j(\ell+1 : \ell+\delta) \cap S_i^t(\ell) \neq \emptyset,$$

then $\tilde{M}_i^t(q\delta) \geq g^q(0)$.

Proof. We proceed by induction on q . The base case is obvious since each $M_j(0)$ is non-negative. For the inductive step, assume that the lemma holds in round $q\delta$ and that some node j does not belong to $S_i^t((q+1)\delta)$. By assumption, the node j has an in-neighbor $k \in S_i^t(q\delta)$ in the digraph $\mathbb{G}(q\delta+1 : (q+1)\delta)$, i.e., there exist a node $k \in S_i^t(q\delta)$ and a path $k = j_0, j_1, \dots, j_\delta = j$ in the round interval $[q\delta+1, (q+1)\delta]$. We have

$$k \in S_i^t(q\delta) \text{ and } j \notin S_i^t((q+1)\delta).$$

Let $d \in \{1, \dots, \delta\}$ be the first index such that

$$j_{d-1} \in S_i^t(q\delta+d-1) \text{ and } j_d \notin S_i^t(q\delta+d).$$

By Lemma 3, $(j_d)_{q\delta+d}^+ \notin S_i^t(q\delta+d-1)$ since $j_d \notin S_i^t(q\delta+d)$. Then j_{d-1} and $(j_d)_{q\delta+d}^+$ are two in-neighbors of j_d whose clocks are not congruent modulo P in round $q\delta+d-1$. It follows that:

$$M_j((q+1)\delta) \geq M_{j_d}(q\delta+d) \geq g(M_{(j_d)_{q\delta+d}^+}(q\delta+d-1)) \geq g(\tilde{M}_i^t(q\delta+d-1)) \geq g^{q+1}(0).$$

The first two inequalities are due to the update rules for M_j and M_{j_d} . The third one is by definition of \tilde{M}_i^t and the fact that g is non-decreasing. The last one is a consequence of the inductive assumption and the third claim of Lemma 3. ◀

Using the assumption of a finite diameter D , we then derive the following lemma.

► **Lemma 7.** *For all non-negative integer $q \in \mathbb{N}$, one of the following statements is true:*

1. *the system is synchronized in round qD ;*
2. $\tilde{M}(qD) \geq g^q(0)$.

Proof. Assume that two nodes i_0 and i_1 hold non-congruent clocks in round qD . For each positive integer $\ell \leq qD$, the second claim of Lemma 3 gives that $S_{i_0}^{qD}(\ell)$ and $S_{i_1}^{qD}(\ell)$ are both non-empty. Since D is the diameter of \mathbb{G} , for each node $j \in V$, we have

$$\text{In}_j(\ell + 1 : \ell + D) \cap S_{i_0}^{qD}(\ell) \neq \emptyset \quad \text{and} \quad \text{In}_j(\ell + 1 : \ell + D) \cap S_{i_1}^{qD}(\ell) \neq \emptyset.$$

By Lemma 6, this implies

$$\tilde{M}_{i_0}^{qD}(qD) \geq g^q(0) \quad \text{and} \quad \tilde{M}_{i_1}^{qD}(qD) \geq g^q(0),$$

and hence, $\tilde{M}(qD) \geq g^q(0)$. ◀

► **Theorem 8.** *In any execution with a dynamic graph whose diameter D is finite, the SAP_g algorithm achieves mod P -synchronization for any non-decreasing function $g : \mathbb{N} \rightarrow \mathbb{N}$ such that $g^* (\lceil \frac{2D}{P} \rceil)$ is finite. Moreover, the stabilization time is bounded by $(g^* (\lceil \frac{2D}{P} \rceil) + 2) D$.*

Proof. We let $q_0 = g^* (\lceil \frac{2D}{P} \rceil)$; hence $q_0 \geq 1$. Applying Lemma 4 with $\delta = D$ and $t = (q_0 - 1)D$, we obtain that either the system is synchronized in round $q_0 D$, or there exist a node j and an integer $d \in \{1, \dots, D - 1\}$ such that $C_j(q_0 D + d - D) = 0$. The digraph $\mathbb{G}(q_0 D + d - D + 1 : q_0 D + d)$ is complete since D is the diameter of \mathbb{G} , and the second claim in Lemma 1 implies that $C_i(q_0 D + d) \leq D$ for any node $i \in V$. Hence,

$$\begin{aligned} PM_i(q_0 D + d) &\geq PM_i(q_0 D) \\ &\geq P\tilde{M}(q_0 D) \\ &\geq P g^{q_0}(0) \\ &\geq 2D \\ &\geq C_i(q_0 D + d) + D \end{aligned}$$

The third inequality is by Lemma 7 and the fourth one is due to the definition of q_0 . Finally, Lemma 5 shows that the system is synchronized in round $(q_0 + 1)D + d$. ◀

3.4 Specializations of the SAP_g Algorithm

We consider the following two strategies:

1. The function g is constant and equal to M , in which case $g^*(m) = 1$ if $m \leq M$, and $g^*(m) = \infty$ otherwise.
2. The function g^* takes only finite values. This is equivalent to the fact that g has no fixed point, or g is strictly inflationary, i.e., $m < g(m)$ for every non-negative integer m .

Theorem 8 leads to two corollaries corresponding to each of the strategies on g . Firstly, when some bound B on the diameter of the dynamic graph is given, we may choose g to be the constant function $g = \lambda x.M$ with $M = \lceil \frac{2B}{P} \rceil$. Then we get $g^* (\frac{2D}{P}) = 1$ and the pseudo-code SAP_g reduces to Algorithm 2.

► **Corollary 9.** *The $SAP_{\lambda x.M}$ algorithm solves the mod P -synchronization problem in any dynamic graph with a diameter less than or equal to $PM/2$.*

Let us observe that Theorem 8 provides an upper bound of three times the diameter D on $SAP_{\lambda x.M}$'s stabilization time, which is independent on the bound B . The limit of $PM/2$ in Corollary 9 is tight, as proved by the following result.

■ **Algorithm 2** The $SAP_{\lambda x.M}$ algorithm.

Variables:

1: $C_i \in \mathbb{N}$;

In each round do:

2: send $\langle C_i \rangle$ to all

3: receive $\langle C_{j_1} \rangle, \langle C_{j_2} \rangle, \dots$ from the set S of in-neighbors

4: $C_i \leftarrow \left[\min_{j \in S} C_j + 1 \right]_{PM}$

► **Theorem 10** (Theorem 4.13 in [1]). *For any even integers P and D satisfying $P < 2D$, there exists an execution of $SAP_{\lambda x.1}$ with a dynamic graph \mathbb{G} whose diameter is D in which mod P -synchronization is never achieved.*

Interestingly, the self-stabilizing algorithm in [9], called *SS-MinSU* and developed for clock synchronization in a static and strongly connected network when a bound B on the diameter³ is available, is actually an optimization of the $SAP_{\lambda x.M}$ algorithm.

As for the algorithm proposed in [3] for a static strongly connected digraph G , it corresponds to the $SAP_{\lambda x.1}$ algorithm, combined with a round-robin strategy which consists, for each node, to send one message per round according to this fixed cyclic order amongst the outgoing neighbors in G . This strategy thus translates the fixed digraph G into a dynamic graph \mathbb{G} . Using Proposition 24 in [10], \mathbb{G} 's diameter can be upper bounded by $3|V|$. Via Corollary 9, the interpretation of the algorithm in [3] for a fixed digraph G in terms of a run of $SAP_{\lambda x.1}$ over the corresponding dynamic graph \mathbb{G} shows that this algorithm works when $P \geq 6|V|$, and its stabilization time is less than $9|V|$ (instead of the correctness condition $P \geq n^2$ and the stabilization bound of $\frac{3}{2}n^2$, given both in [3]).

When the diameter of the dynamic graph is finite but no bound is available, we may use the following corollary of Theorem 8:

► **Corollary 11.** *For any non-decreasing and inflationary function g , SAP_g solves the mod P -synchronization in any dynamic graph whose diameter is finite.*

The idea of a self-adaptive period is borrowed from the seminal paper by Boldi and Vigna [8], and the SAP_g algorithm is a variant of the algorithm they presented for static strongly connected communication graphs. From the viewpoint of design, the main discrepancy lies in the period lengths equal to PM_i in SAP_g , instead of PM_i^2 in Boldi and Vigna's algorithm. As a result, the two algorithms differ in space complexity: while the variables C_i in SAP_g are of the order of $PM(q_0 D)$, the algorithm in [8] uses variables of the order of $PM(q_0 D)^2$, where $q_0 = g^* \left(\lceil \frac{2D}{P} \rceil \right)$; see Section 5.

4 The SAP_g algorithm with infinite diameter

The aim of this section is to study how the assumption of a finite diameter can be relaxed so that the SAP_g algorithm still achieves mod P -synchronization.

³ The bound B is denoted α in the *SS-MinSU* algorithm.

4.1 Extending the class of static rooted networks

A node i is said to be *central* in a dynamic graph \mathbb{G} if its eccentricity is finite, and the *center* of \mathbb{G} , denoted by $Z(\mathbb{G})$, is defined as the set of \mathbb{G} 's central nodes.

$$Z(\mathbb{G}) \stackrel{\text{def}}{=} \{i \in V \mid e_{\mathbb{G}}(i) < \infty\}.$$

If $Z(\mathbb{G})$ is non-empty, then the following integer is well-defined and finite.

$$R \stackrel{\text{def}}{=} \max_{i \in Z(\mathbb{G})} e_{\mathbb{G}}(i). \quad (2)$$

The *kernel* of \mathbb{G} , denoted $K(\mathbb{G})$, is defined as

$$K(\mathbb{G}) \stackrel{\text{def}}{=} \{i \in V \mid \forall t \geq 1, \forall j \in V, \exists t' \geq t : (i, j) \text{ is an edge in } \mathbb{G}(t : t')\}.$$

Intuitively, a node belongs to $K(\mathbb{G})$ if it can infinitely often reach all nodes in finite time. Clearly, it holds that $Z(\mathbb{G}) \subseteq K(\mathbb{G})$. The inclusion is strict in general, as illustrated in Section 4.2. Indeed, the construction guarantees that $Z(\mathbb{G}) = \{i\}$ and $K(\mathbb{G}) = V$. A dynamic graph \mathbb{G} is said to be *strongly centered* if $Z(\mathbb{G}) \neq \emptyset$ and $K(\mathbb{G}) = Z(\mathbb{G})$.

► **Lemma 12.** *The center of any strongly centered dynamic graph \mathbb{G} has no incoming edge from some index t_0 .*

Proof. We denote $\mathbb{G}(\infty)$ a digraph whose set of nodes is V that contains every edge that appears infinitely often in \mathbb{G} . By definition of $K(\mathbb{G})$, each node $i \in K(\mathbb{G})$ can infinitely often reach each node $j \in V$ in the dynamic graph \mathbb{G} , whereas there are finitely many paths between any two nodes. By the pigeonhole principle, each node in $K(\mathbb{G})$ is the root of a spanning tree in $\mathbb{G}(\infty)$. Using the definitions of $K(\mathbb{G})$ and $\mathbb{G}(\infty)$, the converse can also be proved. Then $K(\mathbb{G})$, and hence $Z(\mathbb{G})$ have no incoming edge in $\mathbb{G}(\infty)$, since if i is the root of some spanning tree in $\mathbb{G}(\infty)$, then all i 's incoming neighbours are also roots of a spanning tree. Then, from a certain round, $Z(\mathbb{G})$ has no incoming edge in \mathbb{G} . ◀

In the self-stabilizing paradigm, any predicate that holds from a certain round can be assumed to hold from the beginning. We may then assume $t_0 = 0$ in the rest of the paper.

4.2 The SAP_g algorithm with a central node

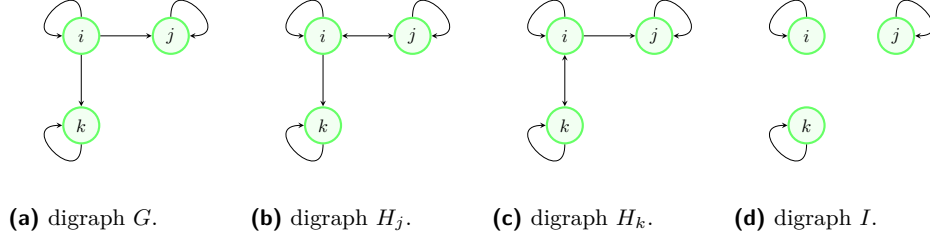
We now study whether SAP_g can achieve mod P -synchronization in networks with an infinite diameter. For that, we first demonstrate that the sole assumption of a non-empty center is not sufficient for SAP_g to achieve mod P -synchronization. We construct an execution of SAP_g with a central node i . The underlying idea of our scenario is that sporadic incoming neighbors disrupt the value of i 's clock and hence preclude any alignment of the other clocks on C_i .⁴

Let G, H_j, H_k, I be the four digraphs defined in Figure 1 with three nodes i, j, k , and let Φ_k be the following predicate on the rounds of a SAP_g execution:

$$(M_i = M_j) \wedge (M_i \geq M_k) \wedge (C_i = C_j) \wedge (C_i \not\equiv_P 0) \wedge (C_i \leq PM_i - 2) \wedge (C_k = 0).$$

The predicate Φ_j is obtained by exchanging the roles of j and k . The proof of the following lemma, which is omitted, follows from a step by step execution of the SAP_g algorithm between rounds t and $t + PM - c$.

⁴ We provide a Python script that may be helpful to verify the correctness of our construction: https://gitlab.com/bossuet/sap_execution.git.



■ **Figure 1** Four digraphs with three nodes.

► **Lemma 13.** *Let t be a round of a SAP_g execution with a dynamic graph \mathbb{G} , and let M and c denote $M_i(t)$ and $C_j(t)$, respectively. Let \mathbb{G}' be any dynamic graph that coincides with \mathbb{G} up to t and such that:*

$$\mathbb{G}'(t+1) = \dots = \mathbb{G}'(t+PM-c-2) = G, \quad \mathbb{G}'(t+PM-c-1) = H_k, \quad \mathbb{G}'(t+PM-c) = I.$$

If Φ_k holds at round t of the SAP_g execution with \mathbb{G}' , then Φ_j holds at round $t+PM-c$ of this execution.

We now fix two positive integers M^0 and c^0 such that $c^0 \in \{1, \dots, PM^0 - 2\}$ and $c^0 \not\equiv_P 0$, and we consider the two sequences $(M^r)_{r \geq 0}$ and $(c^r)_{r \geq 0}$ defined by:

$$\begin{cases} M^{r+1} = g^{PM^r - c^r - 1}(M^r) \\ c^{r+1} = PM^r - c^r. \end{cases}$$

We let $M^{-1} = 0$. The dynamic graph \mathbb{G} defined as:

$$\begin{aligned} \mathbb{G}(PM^{r-1} + 1) &= \dots = \mathbb{G}(PM^r - c^r - 2) = G, \\ \mathbb{G}(PM^r - c^r - 1) &= H_k \text{ or } H_j, \\ \mathbb{G}(PM^r - c^r - 1) &= I, \end{aligned}$$

is rooted with delay two and i is its unique center. Lemma 13 shows that Φ_k holds infinitely often in the SAP_g execution with the dynamic graph \mathbb{G} and starting with:

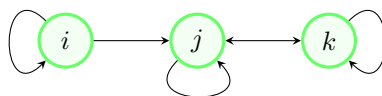
$$M_i(0) = M_j(0) = M_k(0) = M^0, \quad C_i(0) = C_j(0) = c^0, \quad \text{and} \quad C_k(0) = 0.$$

Hence, the nodes are never synchronized.

4.3 The SAP_g algorithm in strongly centered network

That leads us to consider the stronger assumption that the network is strongly centered, without requiring any global knowledge on $Z(\mathbb{G})$. However, the simple but typical scenario below shows that the simplified version of SAP_g with a fixed period, namely the $SAP_{\lambda x.M}$ algorithm, does not achieve mod P -synchronization in the execution with the initial values $C_i(0) = C_j(0) = 1$ and $C_k(0) = 0$ and the static graph H defined in Figure 2, even for large value of M . Indeed, at each round t , it holds that $C_i(t) = [t+1]_{PM}$, $C_k(t) = [t]_{PM}$, and

$$C_j(t) = \begin{cases} 1 & \text{if } [t]_{PM} = 0 \\ [t]_{PM} & \text{otherwise.} \end{cases}$$



■ **Figure 2** The digraph H with three nodes.

The striking point of increasing periods is precisely to overcome the above-mentioned limitation: we are going to prove that the SAP_g algorithm achieves mod P -synchronization in the case of a strongly centered network under the sole condition of a non-decreasing and strictly inflationary function g . In other words, while Corollary 9 has no counterpart for strongly centered dynamic graphs, we will show that Corollary 11 extends to this latter class of dynamic graphs.

We fix a strongly centered dynamic graph \mathbb{G} , and an execution σ of SAP_g with \mathbb{G} . Lemma 12 shows that, from a certain round t_0 , the nodes in $Z(\mathbb{G})$ (denoted Z , for short) receive no message from the nodes in $V \setminus Z$. From round t_0 , from the viewpoint of every node in Z , the execution σ is thus indistinguishable from an execution with the set of nodes equal to Z and a dynamic graph whose diameter is finite. Theorem 8 shows that mod P -synchronization is eventually achieved in Z . A closer look at the SAP_g algorithm yields the following more precise result: there exist two non-negative integers s and M such that

$$\forall t \geq s, \forall k, \ell \in Z, C_k(t) \equiv_P C_\ell(t) \text{ and } M_k(t) = M. \quad (3)$$

The minimum integer s satisfying Eq. (3) is denoted by t_1 . This integer corresponds to the round in which the subsystem composed of central nodes achieves mod P -synchronization. Recalling that R is the integer defined in Eq.(2), we also define the following two constants.

$$q_1 \stackrel{\text{def}}{=} g^* \left(\left\lceil M + \frac{R+1}{P} \right\rceil \right) \text{ and } t_2 \stackrel{\text{def}}{=} \max(t_1, q_1 R). \quad (4)$$

► **Lemma 14.** *Let i_0 be any central node. For any $t > t_2$, and any nodes $j \notin S_{i_0}^{t_2}(t)$, it holds that $C_j(t) \neq 0$.*

Proof. Let j be any node that does not belong to $S_{i_0}^{t_2}(t)$. By definition of R , there exists an edge (i_0, j_t^+) in each digraph $\mathbb{G}(t-R : t-1)$. The third claim of Lemma 1 and Eq. (3) imply that $C_{i_0}(t-R-1) < PM$. Then the second claim in Lemma 1 implies that $C_{j_t^+}(t-1) < PM + R$. Moreover, using the second claim of Lemma 3, each integer $t' \leq t_2$ satisfies

$$\text{In}_j(t'+1 : t'+R) \cap S_{i_0}^{t_2}(t') \supseteq Z \cap S_{i_0}^{t_2}(t') \neq \emptyset.$$

Applying the third claim of Lemma 3, Lemma 6 and the definition of q_1 ,

$$P\tilde{M}_{i_0}^{t_2}(t_2) \geq P\tilde{M}_{i_0}^{t_2}(q_1 R) \geq Pg^{q_1}(0) \geq PM + R + 1. \quad (5)$$

From $j \notin S_{i_0}^{t_2}(t)$, we have $j_t^+ \notin S_{i_0}^{t_2}(t-1)$ by Lemma 3, and then

$$C_{j_t^+}(t-1) < PM + R \leq P\tilde{M}_{i_0}^{t_2}(t_2) - 1 \leq P\tilde{M}_{i_0}^{t_2}(t-1) - 1 \leq PM_{j_t^+}(t-1) - 1.$$

The second inequality comes from Eq. (5), the third from Lemma 3, and the fourth is a consequence of $j_t^+ \notin S_{i_0}^{t_2}(t-1)$. We obtain $C_j(t) \neq 0$ by Lemma 2. ◀

► **Theorem 15.** *In any execution with a strongly centered dynamic graph, the SAP_g algorithm achieves mod P -synchronization for any non-decreasing and inflationary function g . Moreover, the stabilization time is bounded by $t_2 + PM + R$, where R , M and t_2 are defined by Eq.(2), (3) and (4).*

Proof. Each node $i \in Z$ satisfies $C_i(t) < PM$ in each round. Then we obtain that some node $i_0 \in Z$ reaches $C_{i_0}(t_3) = 0$ in some round $t_3 \in \{t_2, \dots, t_2 + PM - 1\}$ (otherwise, an inductive reasoning using Lemma 2 would contradict the above-mentioned fact). By Lemma 14, each node $j \notin S_{i_0}^{t_2}(t_3 + d)$ satisfies $C_j(t_3 + d) \neq 0$ for each $d > 0$. By Lemma 4, the system is synchronized in round $t_3 + R$. ◀

5 Complexity analysis

In this section, we provide a complexity analysis of SAP_g in the case of a network with finite diameter, and then in the case of a strongly centered network. We discuss the choice of the g function and its impact on both stabilization time and space complexity. We first state a theorem that will be used to bound the memory usage, measured in bits, of each node in any execution of SAP_g that achieves mod P -synchronization.

► **Theorem 16.** *In any execution of SAP_g that achieves mod P -synchronization, if q is the round in which the system synchronizes, then the memory usage of each node is less than $\log_2 P + 2 \log_2 \left(g^q \left(\max_{i \in V} M_i(0) \right) \right)$ bits.*

Proof. We define, for each round t ,

$$\overline{M}(t) \stackrel{\text{def}}{=} \max_{i \in V} M_i(t).$$

From the pseudo-code of SAP_g , we directly obtain, for each positive integer t ,

$$\overline{M}(t) \leq g(\overline{M}(t-1)),$$

and thus,

$$\overline{M}(t) \leq g^t(\overline{M}(0)). \tag{6}$$

As $\overline{M}(t)$ is non-decreasing as long as $t \leq q$ and is stable afterwards, each M_i belongs to the interval $\{1, \dots, \overline{M}(q)\}$, and each C_i belongs to $\{0, \dots, P\overline{M}(q) - 1\}$. The number of reachable states by any single node is at most equal to the cardinality of the product of these two sets, that is, $Pg^q(\overline{M}(0))^2$. Then at most $\log_2 P + 2 \log_2 (g^q(\overline{M}(0)))$ bits are needed to store the state of one node. ◀

5.1 Networks with finite diameter

Theorem 16 implies the following corollary in the case of a network with finite diameter.

► **Corollary 17.** *In any execution of SAP_g , if the diameter D of the network is finite, then the memory usage of each node is bounded by $\log_2 P + 2 \log_2 \left(g^{(g^*(2^{D/P+1})+2)^D} \left(\max_{i \in V} M_i(0) \right) \right)$.*

Theorem 8 and Corollary 17 demonstrate some trade-off between stabilization time and space complexity. The faster g grows, the lower the synchronization time is, and the higher its space complexity is. To further illustrate this trade-off, Table 1 provides the time and

space complexity in three cases. First, when a bound B on the diameter is given, choosing $g = \lambda x \cdot \lceil \frac{2B}{P} \rceil$ provides the best stabilization time, namely $3D$, which interestingly does not depend on the bound B . When no bound on the diameter is available, the overhead of $SAP_{\lambda x \cdot 2x}$ over $SAP_{\lambda x \cdot \lceil 2B/P \rceil}$ is only logarithmic while $SAP_{\lambda x \cdot x+1}$ results in an additional delay of $O(D^2)$ rounds for stabilization.

Regarding space complexity, $SAP_{\lambda x \cdot \lceil 2B/P \rceil}$ and $SAP_{\lambda x \cdot x+1}$ uses $O(\log B)$ and $O(\log D)$ bits, respectively. This illustrates how SAP_g may be more memory-efficient using its adaptive mechanism and a judicious choice of g . By contrast, the space complexity of $SAP_{\lambda x \cdot 2x}$ is only linear in D , which might be problematic for memory-constrained devices.

■ **Table 1** Complexity bounds of SAP_g in networks with finite diameter D and $B \geq D$.

g	synchronization time	space complexity
$g = \lambda x \cdot \lceil \frac{2B}{P} \rceil$	$3D$	$\log_2 P + 2 \log_2 \lceil \frac{2B}{P} \rceil$
$g = \lambda x \cdot x + 1$	$(\frac{2D}{P} + 3) D$	$\log_2 P + 2 \log_2 \left(\max_{i \in V} M_i(0) + \frac{2D^2}{P} + 3D \right)$
$g = \lambda x \cdot 2x$	$(\log_2 (1 + \frac{2D}{P}) + 2) D$	$\log_2 P + 2 \log_2 \left(\max_{i \in V} M_i(0) \right) + 2D \log_2 (1 + \frac{2D}{P}) + 4D$

5.2 Strongly centered networks

In the case of a strongly centered network, Theorem 15 bounds the stabilization time by $t_2 + PM + R$. Eq. (6) then provides the following upper bound for M , where t_1 is the minimum integer satisfying Eq.(3).

$$M \leq g^{t_1} (\max_{i \in V} M_i(0)),$$

and thus, we obtain:

$$\begin{aligned} t_2 &\leq \max \left(t_1, Rg^* \left(\left\lceil g^{t_1} (\max_{i \in V} M_i(0)) + \frac{R+1}{P} \right\rceil \right) \right) \\ &\leq Rg^* \left(g^{t_1} (\max_{i \in V} M_i(0)) + \frac{R+1}{P} + 1 \right). \end{aligned}$$

Theorem 8 also provides the following upper bound for t_1 :

$$t_1 \leq \left(g^* \left(\left\lceil \frac{2D}{P} \right\rceil \right) + 2 \right) D,$$

where D is the diameter of the (dynamic) subgraph of \mathbb{G} induced by Z . Theorem 16 then implies the following corollary in the case of a strongly centered network.

► **Corollary 18.** *In any execution of SAP_g in which the network is strongly centered, the memory usage of each node is bounded by*

$$\log_2 P + 2 \log_2 \left(g^{Rg^*(M' + \frac{R+1}{P} + 1) + PM' + R} \left(\max_{i \in V} M_i(0) \right) \right),$$

where $M' = g^{(g^*(2D/P+1)+2)D} (\max_{i \in V} M_i(0))$.

■ **Table 2** Complexity bounds of SAP_g in strongly centered networks. Here, R is the quantity defined by Eq.(2), and D is the diameter of the dynamic subgraph induced by $Z(\mathbb{G})$.

g	synchronization time	space complexity
$g = \lambda x.x + 1$	$\left(t_1 + \max_{i \in V} M_i(0) \right) (P + R)$ $+ R \left(2 + \frac{R+1}{P} \right)$ where t_1 satisfies $t_1 \leq \left(\frac{2D}{P} + 3 \right) D$	$\log_2 P + 2 \log_2 \left(\left(t_1 + \max_{i \in V} M_i(0) \right) (P + R) \right)$ $+ R \left(2 + \frac{R+1}{P} \right) + \max_{i \in V} M_i(0)$
$g = \lambda x.2x$	$P \left(\max_{i \in V} M_i(0) \right) 2^{t_1} + R(1 + t_1)$ $+ R \log_2 \left(\left(\max_{i \in V} M_i(0) \right) \left(1 + \frac{R+1}{P} \right) \right)$ where $t_1 \leq \log_2 \left(1 + \frac{2D}{P} \right) D + 2D$	$\log_2 P + 2 \log_2 \left(\max_{i \in V} M_i(0) \right) + 2P \left(\max_{i \in V} M_i(0) \right) 2^{t_1}$ $+ 2R \left(1 + t_1 + \log_2 \left(\left(\max_{i \in V} M_i(0) \right) \left(1 + \frac{R+1}{P} \right) \right) \right)$

Table 2 provides a bound on synchronization time and space complexity in the cases $g = \lambda x.x + 1$ and $g = \lambda x.2x$. It shows that the trade-off presented in the previous section no longer applies. In the case $g = \lambda x.2x$, both time and space complexity contain exponential terms. A real-world device would quickly run out of memory. The SAP_g algorithm remains practical only if g is a slowly growing function. Comparing with Table 1, we observe that SAP_g achieves better performance in networks with finite diameter than in strongly centered networks. Choosing $g = \lambda x.x + 1$, the time complexity is in $O \left(R \left(D^2 + R + \max_{i \in V} M_i(0) \right) \right)$ in the later case, compared to $O(D^2)$ in the earlier case. A similar overhead is added to space complexity. Overall, choosing $g = \lambda x.x + 1$ seems to provide the best performances, as it is the “least inflationary” function.

6 Concluding remarks

We presented the SAP_g algorithm that solves the mod P -synchronization problem in any dynamic network that either has a finite diameter or is strongly centered. Both assumptions correspond to connectivity properties that have to hold in bounded periods of time. These results highlight the critical importance of timing bounds for the network to be connected enough and demonstrate how time may act as a healer.

The correctness proofs also provide bounds on stabilization time and space complexity of the SAP_g algorithm. The time bound and the space bound depend respectively on the functions g^* and g , leading thus to a time-space trade-off for choosing g : the more inflationary g is, the lower the time complexity is, and the higher its space complexity is. Moreover, these results show how the initial knowledge on a bound on the diameter allows for more efficient solutions in terms of both time and space.

The scenario in Section 4.2 shows that the SAP_g algorithm does not work anymore when relaxing the assumption of a strongly centered network into the one of a non-empty center. A natural question then arises about the possibility of designing a finite-state self-stabilizing algorithm that provides nodes with clocks modulo P which eventually synchronize in a dynamic network with at least one central node.

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 D. Angluin, J. Aspnes, D. Eisenstat, and E. Ruppert. The computational power of population protocols. *Distributed Computing*, 20(4):279–304, 2007.
- 3 A. Arora, S. Dolev, and M. Gouda. Maintaining digital clocks in step. *Parallel Processing Letters*, 1:11–18, 1991.
- 4 P. Bastide, G. Giakkoupis, and H. 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. SIAM, 2021.
- 5 M. 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 P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
- 7 L. Boczkowski, A. Korman, and E. Natale. Minimizing message size in stochastic communication patterns: fast self-stabilizing protocols with 3 bits. *Distributed Comput.*, 32(3):173–191, 2019.
- 8 P. Boldi and S. Vigna. Universal dynamic synchronous self-stabilization. *Distributed Computing*, 15(3):137–153, 2002.
- 9 C. Boulinier, F. Petit, and V. Villain. Synchronous vs. asynchronous unison. *Algorithmica*, 51(1):61–80, 2008.
- 10 B. Charron-Bost. Geometric bounds for convergence rates of averaging algorithms. *Information and Computation*, 2022. To appear, available at [arXiv:2007.04837](https://arxiv.org/abs/2007.04837).
- 11 B. Charron-Bost and A. Schiper. The Heard-Of model: computing in distributed systems with benign faults. *Distributed Computing*, 22(1):49–71, 2009.
- 12 A. Cornejo and F. Kuhn. Deploying wireless networks with beeps. In *Distributed Computing, 24th International Symposium, DISC 2010, Cambridge, MA, USA, September 13-15, 2010. Proceedings*, volume 6343 of *Lecture Notes in Computer Science*, pages 148–162. Springer, 2010.
- 13 S. Dolev. Possible and impossible self-stabilizing digital clock synchronization in general graphs. *Real Time Syst.*, 12(1):95–107, 1997.
- 14 S. Dolev and J. L. Welch. Self-stabilizing clock synchronization in the presence of byzantine faults. *J. ACM*, 51(5):780–799, 2004.
- 15 C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.
- 16 S. Even and S. Rajsbaum. Unison, canon, and sluggish clocks in networks controlled by a synchronizer. *Math. Syst. Theory*, 28(5):421–435, 1995.
- 17 M. Feldmann, A. Khazraei, and C. Scheideler. Time- and space-optimal discrete clock synchronization in the beeping model. In *SPAA '20: 32nd ACM Symposium on Parallelism in Algorithms and Architectures, USA, 2020*, pages 223–233. ACM, 2020.
- 18 M. Gouda and T. Herman. Stabilizing unison. *Inf. Process. Lett.*, 35(4):171–175, 1990.
- 19 T. Herman and S. Ghosh. Stabilizing phase-clocks. *Inf. Process. Lett.*, 54(5):259–265, 1995.
- 20 A. Jadbabaie. Natural algorithms in a networked world: technical perspective. *Commun. ACM*, 55(12):100, 2012.
- 21 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.
- 22 L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.
- 23 S. H. Strogatz. From kuramoto to crawford: exploring the onset of synchronization in populations of coupled oscillators. *Physica D*, 143(1-4):1–20, 2000.

