31 International Symposium on Distributed Computing

DISC 2017, October 16–20, Vienna, Austria

Edited by

Andréa W. Richa

Editor

Andréa W. Richa
Computer Science and Engineering
School of Computing, Informatics and Decision Systems Engineering (CIDSE)
Arizona State University
Tempe, AZ, USA
aricha@asu.edu

ACM Classification 1998
C.2 Computer-Communication Networks, C.2.4 Distributed Systems, D.1.3 Concurrent Programming, E.1 Data Structures, F Theory of Computation, F.1.1 Models of Computation, F.1.2 Modes of Computation

ISBN 978-3-95977-053-8

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at http://www.dagstuhl.de/dagpub/978-3-95977-053-8.

Publication date
October, 2017

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 http://dnb.d-nb.de.

License
This work is licensed under a Creative Commons Attribution 3.0 Unported license (CC-BY 3.0): http://creativecommons.org/licenses/by/3.0/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.DISC.2017.0

LIPIcs – Leibniz International Proceedings in Informatics

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

Editorial Board

- Luca Aceto (Reykjavik University)
- Susanne Albers (TU München)
- Chris Hankin (Imperial College London)
- Deepak Kapur (University of New Mexico)
- Michael Mitzenmacher (Harvard University)
- Madhavan Mukund (Chennai Mathematical Institute)
- Anca Muscholl (University Bordeaux)
- Catuscia Palamidessi (INRIA)
- Raimund Seidel (Saarland University and Schloss Dagstuhl – Leibniz-Zentrum für Informatik)
- Thomas Schwentick (TU Dortmund)
- Reinhard Wilhelm (Saarland University)

ISSN 1868-8969

http://www.dagstuhl.de/lipics

Contents

Keynote Talks

Regular Papers

31st International Symposium on Distributed Computing (DISC 2017).
Editor: Andréa W. Richa
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Brief Announcements

Preface

DISC, the International Symposium on DIStributed Computing, is an international forum on the theory, design, analysis, implementation and application of distributed systems and networks. DISC is organized in cooperation with the European Association for Theoretical Computer Science (EATCS).

This volume contains the papers presented at DISC 2017, the 31st International Symposium on Distributed Computing, held on October 16–20, 2017 in Vienna, Austria. The volume includes the citation for the 2017 Edsger W. Dijkstra Prize in Distributed Computing, jointly sponsored by DISC and the ACM Symposium on Principles of Distributed Computing (PODC), that was presented at DISC 2017 to Elizabeth Borowsky and Eli Gafni for their work *Generalized FLP Impossibility Result for t-resilient Asynchronous Computations*. The volume also includes the citation for the 2017 Doctoral Dissertation Award, also jointly sponsored by DISC and PODC, that was presented at PODC 2017 in Washington, DC, USA to Mohsen Ghaffari for his PhD thesis titled *Improved Distributed Algorithms for Fundamental Graph Problems*, supervised by Nancy Lynch at the Massachusetts Institute of Technology.

DISC 2017 received a very high number of submissions — 160 regular paper and 11 brief announcement submissions — which were all peer reviewed. The quality of the submissions was also very high this year, posing a challenge to the Program Committee (PC). Every submission was read and evaluated by at least three members of the PC, with the assistance of 134 external reviewers. Following a 7-day discussion period, the PC held a virtual meeting on June 28–29, 2017, which was attended by all but a few of its members. The PC selected 39 contributions out of the 160 regular paper submissions, for 37 regular presentations at the symposium: Three of the papers had highly overlapping results and were therefore asked to combine their published and oral presentations (the combined paper appears as *Three Notes on Distributed Property Testing*).

For each regular presentation, the authors were invited to submit a paper of up to 15 pages for this volume (the final number of pages per paper may vary slightly due to the final typesetting of this volume); the only exception was the paper resulting from the 3-way merge, which was allowed a longer proceedings version. Nineteen brief announcements were accepted in total, for a short presentation accompanied by a 3-page publication in the proceedings each: Four of those were originally submitted as a brief announcements; the other 15 were regular submissions that were rejected, but generated substantial interest among the members of the PC and were invited to be published as brief announcements. Each brief announcement summarizes ongoing work or recent results, and it can be expected that these results will appear as full papers in later conferences or journals.

This was the first year that DISC had its proceedings published by LIPIcs (Leibniz International Proceedings in Informatics): Jukka Suomela, the DISC 2017 proceedings chair, embraced the challenge and successfully led the transition to LIPIcs. Revised and expanded versions of several selected proceedings papers will be considered for publication in a special issue of the journal Distributed Computing.

The Best Paper Award for DISC 2017 was presented to Mohsen Ghaffari, Juho Hirvonen, Fabian Kuhn, Yannic Maus, Jukka Suomela and Jara Uitto for their paper *Improved Distributed Degree Splitting and Edge Coloring*. The Best Student Paper Award for DISC 2017 was presented to Manuela Fischer for her solo-authored paper *Improved Deterministic Distributed Matching via Rounding*.

31st International Symposium on Distributed Computing (DISC 2017).
Editor: Andréa W. Richa
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The program featured three keynote lectures, presented by Anne-Marie Kermarrec (INRIA, Rennes, France), Christian Cachin (IBM Research Zurich, Switzerland), and Dana Randall (Georgia Tech, USA). An abstract of each keynote lecture is included in the proceedings. The program also included a celebration for Yoram Moses' 60th birthday, organized by Nir Shavit, which included a set of invited talks by Shafi Goldwasser (MIT, USA), Joe Halpern (Cornell University, USA), Sergio Rajsbaum (UNAM, Mexico), Moshe Tenenholtz (Technion, Israel), and Moshe Vardi (Rice University, USA).

Six workshops were co-located with the DISC symposium this year. The following workshops were held on the day preceding the main conference (October 16): the *4th Workshop on Formal Reasoning in Distributed Algorithms (FRIDA)*, organized by Swen Jacobs, Igor Konnov, Stephan Merz, and Josef Widder; the *1st Workshop on Blockchain Technology and Theory*, organized by Emmanuelle Anceaume, Christian Cachin, Maurice Herlihy, and Maria Potop-Butucaru; and the *1st Workshop on the Theory and Practice of Concurrency*, organized by Dan Alistarh. The following workshops were held following the main conference on October 20: the *6th Workshop on Advances in Distributed Graph Algorithms (ADGA)*, chaired by Fabian Kuhn; the *2nd Workshop on Computing in Dynamic Networks (CoDyn)*, organized by Arnaud Casteigts and Swan Dubois; and the *1st Workshop on Hardware Design and Theory (HDT)*, chaired by Cristoph Lenzen.

We wish to thank the many contributors to DISC 2017: the authors of the submitted papers, the PC members and the reviewers, the three keynote speakers, the conference general chairs and local organizers Ulrich Schmid and Josef Widder, the publicity chair Dan Alistarh, the proceedings chair Jukka Suomela, the web chair Kyrill Winkler, all the workshop organizers led by the workshop chair Josef Widder, the DISC Steering Committee, under the guidance of Shlomi Dolev, and the sponsors for their generous support of DISC 2017.

October 2017

Andréa W. Richa,
DISC 2017 Program Chair

Symposium Organization

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

Program Chair

Andréa W. Richa	Arizona State University, USA

Program Committee

Marcos K. Aguilera	VMware Research Group, USA
Dan Alistarh	IST, Austria
James Aspnes	Yale University, USA
Rida Bazzi	Arizona State University, USA
Borzoo Bonakdarpour	McMaster University, Canada
Christian Cachin	IBM Research Zurich, Switzerland
Andrea Clementi	Tor Vergata University, Rome, Italy
Andrzej Czygrinow	Arizona State University, USA
Oksana Denysyuk	Pure Storage, USA
David Doty	UC Davis, USA
Alon Efrat	University of Arizona, USA
Yuval Emek	Technion, Israel
Sándor Fekete	TU Braunschweig, Germany
Jeremy Fineman	Georgetown University, USA
Pierre Fraigniaud	CNRS & U. Paris-Diderot, France
Mohsen Ghaffari	ETH Zurich, Switzerland
George Giakkoupis	INRIA Rennes, France
Seth Gilbert	NUS, Singapore
Olga Goussevskaia	UFMG, Brazil
Rachid Guerraoui	EPFL, Switzerland
Fabian Kuhn	University of Freiburg, Germany
Dahlia Malkhi	VMware Research Group, USA
Rotem Oshman	Tel-Aviv University, Israel
Boaz Patt-Shamir	Tel-Aviv University, Israel
Maria Potop-Butucaru	Paris-6 University, France
Rajmohan Rajaraman	Northeastern University, USA
Nicola Santoro	Carleton University, Canada
Christian Scheideler	University of Paderborn, Germany

31st International Symposium on Distributed Computing (DISC 2017).
Editor: Andréa W. Richa

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

Stefan Schmid	University of Aalborg, Denmark
David Soloveichik	UT Austin, USA
Aravind Srinivasan	University of Maryland, USA
Jukka Suomela	Aalto University, Finland
Shang-Hua Teng	USC, USA
Nitin Vaidya	UIUC, USA
Jennifer Welch	Texas A&M University, USA
Yukiko Yamauchi	Kyushu University, Japan

Steering Committee

Roberto Baldoni	Sapienza Università di Roma, Italy
Keren Censor-Hillel	Technion, Israel
Shlomi Dolev (*chair*)	Ben-Gurion University of the Negev, Israel
Cyril Gavoille	Bordeaux University, France
Yoram Moses	Technion, Israel
Andréa W. Richa	Arizona State University, USA
Jukka Suomela	Aalto University, Finland

Local Organization

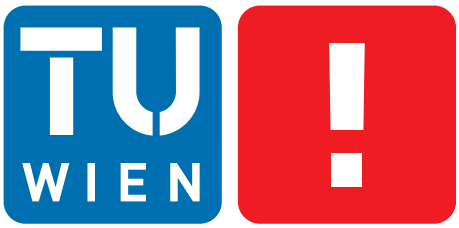
Ulrich Schmid (*general chair*)	TU Wien, Austria
Josef Widder (*general/workshop chair*)	TU Wien, Austria
Jukka Suomela (*proceedings chair*)	Aalto University, Finland
Dan Alistarh (*publicity chair*)	IST, Austria
Kyrill Winkler (*web chair*)	TU Wien, Austria

External Reviewers

David Adjiashvili	Mattia D'Emidio	Adam Goldbraich
Yehuda Afek	Shantanu Das	Luciano Gualà
Mohamad Ahmadi	Pedro de Carvalho Gomes	Manoj Gupta
Saeed Akhoondian Amiri	Giuseppe Di Luna	Mika Göös
Emmanuelle Anceaume	Gabriele Di Stefano	Bernhard Haeupler
Chen Avin	Carola Doerr	Michal Hanckowiak
Alkida Balliu	Aleksandar Dragojevic	Marc Heinrich
Soumya Basu	Swan Dubois	Maurice Herlihy
Luca Becchetti	El Mahdi El Mhamdi	Kristian Hinnenthal
Aditya Bhaskara	Michael Feldmann	Juho Hirvonen
Davide Bilò	Laurent Feuilloley	Qiang-Sheng Hua
Shimon Bitton	Manuela Fischer	Taisuke Izumi
Quentin Bramas	Orr Fischer	Leander Jehl
Trevor Brown	Paola Flocchini	Colette Johnen
Costas Busch	Klaus-Tycho Förster	Dominik Kaaser
Keerti Choudhary	Carlos Vinícius G. C. Lima	Sayaka Kamei
Colin Cooper	Vijay Garg	Erez Kantor
Jurek Czyzowicz	Rati Gelashvili	Yoshiaki Katayama
Gianlorenzo D'Angelo	Robert Gmyr	Lucianna Kiffer

Peter Kling
Christina Kolb
Janne H. Korhonen
Amos Korman
Adrian Kosowski
Evangelos Kranakis
Dominik Krupke
Saptaparni Kumar
Shay Kutten
Arnaud Labourel
Tuomo Lempiäinen
Johannes Lengler
Christoph Lenzen
Hongjin Liang
Thomas Locher
Romaric Ludinard
Russell Martin
Fabien Mathieu
Alex Matveev
Luzius Meisser
Pall Melsted
Hammurabi Mendes
Pedro Montealegre
Achour Mostéfaoui
Giorgi Nadiradze
Danupon Nanongkai
Emanuele Natale
Alfredo Navarra
Rajko Nenadov
Dennis Olivetti
Fukuhito Ooshita
Keishla Ortiz-Lopez
Francesco Pasquale
Sriram Pemmaraju
Sathya Peri
Matthieu Perrin
Mor Perry
Giuseppe Persiano
Erez Petrank
Giuseppe Prencipe
Christopher Purcell
Ivan Rapaport
Christoforos Raptopoulos
Nicolás Rivera
Will Rosenbaum
Matthieu Roy
Leonid Ryzhyk
Thomas Sauerwald
Saket Saurabh
Giacomo Scornavacca
Michele Scquizzato
Dragos-Adrian Seredinschi
Alexander Setzer
Gokarna Sharma
Masahiro Shibata
Julian Shun
Umair Siddique
Sebastian Siebertz
Janos Simon
Michael Spear
Kannan Srinathan
Grzegorz Stachowiak
Thim Strothmann
Edward Talmage
Sumedh Tirodkar
Ioan Todinca
Lewis Tseng
Jara Uitto
Giovanni Viglietta
Paolo Viotti
Marko Vukolić
Anil Kumar Vullikanti
Jingjing Wang
Wojciech Wawrzyniak
Marcin Witkowski
Philipp Woelfel
Igor Zablotchi

Sponsoring Organizations



Dissemination sponsored by

Der Wissenschaftsfonds.

DISC 2017 acknowledges the use of the EasyChair system for handling submissions and managing the review process, and LIPIcs for producing and publishing the proceedings.

2017 Edsger W. Dijkstra Prize in Distributed Computing

The Edsger W. Dijkstra Prize in Distributed Computing was created to acknowledge outstanding papers on the principles of distributed computing whose significance and impact on the theory or practice of distributed computing have been evident for at least a decade. The Prize is sponsored jointly by the ACM Symposium on Principles of Distributed Computing (PODC) and the EATCS Symposium on Distributed Computing (DISC). This award is presented annually, with the presentation taking place alternately at PODC and DISC.

The 2017 Award Committee, composed of Alexander Schwarzmann (Chair), Marcos K. Aguilera, Alessandro Panconesi, Andrzej Pelc, Andréa W. Richa, and Roger Wattenhofer, has selected

Elizabeth Borowsky and Eli Gafni

to receive the 2017 Edsger W. Dijkstra Prize in Distributed Computing for the outstanding paper:

Elizabeth Borowsky, Eli Gafni:
Generalized FLP impossibility result for t-resilient asynchronous computations.
Proceedings of the 25th Annual ACM Symposium on Theory of Computing (STOC 1993), pages 91–100, 1993.

This is a fundamental paper in the original sense. It contains two breakthrough contributions. First, it lays a new concept of read-write simulations in the very foundation of distributed computing. Second, it introduces the *immediate-snapshot* model. For the first contribution, the paper argues that, even though distributed systems exhibit multiple seemingly incomparable instantiations, they operate on the same fundamental principles. By deriving these principles, we could obtain computability and complexity results concerning a given specific distributed system via simulations and reductions.

The paper illustrates this approach by proposing an ingenious simulation tool, now commonly referred to as the *BG Simulation*. The tool allows a system of $k+1$ processes to consistently *simulate* algorithms designed for any *k-resilient* system. The BG Simulation proved to be instrumental in establishing impossibility results and building reductions between them. In particular, this paper uses the BG Simulation to derive the fundamental impossibility of k-resilient k-set consensus from the impossibility of *wait-free* set consensus.

The second key contribution, the immediate-snapshot model, leads to a simple and elegant combinatorial characterization of the set of runs of a protocol. This characterization then leads to the impossibility of wait-free set consensus through a simple application of Sperner's Lemma.

These two points—the use of a simpler model of computation to establish the wait-free set-consensus impossibility and the use of simulation to derive the generalized k-resilient impossibility from the wait-free one—distinguishes this paper from two concurrent papers appeared at STOC 1993, by Herlihy and Shavit and by Saks and Zaharoglou, journal versions of which were later awarded the prestigious Gödel prize.

31st International Symposium on Distributed Computing (DISC 2017).
Editor: Andréa W. Richa

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

Since 1993, both contributions of this paper were widely adopted by the distributed-computing community. The illuminating BG Simulation technique gave rise to a broad spectrum of results in various contexts: from adversarial shared-memory computing to mobile Byzantine robots. The BG simulation and abstractions around it establish now the very basis of the state-of-the-art research field of distributed computability theory. The (iterated) immediate-snapshot model is widely adopted nowadays in combinatorial representations of distributed computations. As was correctly conjectured by the authors in a concurrent paper, the *protocol complex* of this model is precisely captured by the *standard chromatic subdivision*, enabling straightforward reasoning about the model's computability. The two contributions also help us in *teaching* the foundations of resilience: it is much easier to deal with the wait-free model, and deduce computability of other models via simulation.

In summary, this paper turned out to be crucial for our understanding of fault-tolerant distributed computing. It proposed a powerful reduction technique, the BG simulation, it introduced the immediate-snapshot model, and it established the fundamental impossibility of k-resilient k-set consensus.

Yehuda Afek, Tel Aviv University, Israel
Rachid Guerraoui, EPFL, Switzerland
Taisuke Izumi, Nagoya Institute of Technology, Japan
Petr Kuznetsov, Télécom ParisTech, France

2017 Principles of Distributed Computing Doctoral Dissertation Award

The winner of the 2017 Principles of Distributed Computing Doctoral Dissertation Award is Dr. **Mohsen Ghaffari**, for his dissertation **Improved Distributed Algorithms for Fundamental Graph Problems**, written under the supervision of Prof. Nancy Lynch at the Massachusetts Institute of Technology.

Ghaffari's thesis represents an extraordinary study of network algorithms which is, at the same time, both deep and extensive. Many of the results included in this thesis (5, to be precise) have already been awarded "Best Student Paper" award or "Best Paper" award (and sometimes both) in top-notch conferences. The number of publications produced by Ghaffari while working on his thesis is also staggering (over 35 papers!): the thesis covers only a small part of his work. Most important, Ghaffari made a very significant contribution to the Theory of Network Algorithms, particularly to randomized network algorithms.

Specifically, the thesis contains three parts. In the first part, a new randomized algorithm for the Maximal Independent Set (MIS) problem is developed. The algorithm is simple and local in a strong sense: the termination time of a node depends only the coin-tosses within distance 2. This algorithm improves on all previous results and thus leads to improved time complexity in the many applications that use MIS as a building block. In the second part, Ghaffari presents results concerning vertex- and edge-connectivity in graphs, with applications to different problems such as Connected Dominated Set and Minimum Spanning Tree computation. And in the last part of the thesis, following classical packet routing results, scheduling multiple network tasks concurrently is considered. It is shown that in fact, there may be an unavoidable logarithmic gap between the case of packet routing and general tasks, but on the positive side, we never need to pay more than a single logarithmic factor (beyond the "congestion+dilation" lower bound) to schedule multiple tasks.

The award. The award is sponsored jointly by the ACM Symposium on Principles of Distributed Computing (PODC) and the EATCS Symposium on Distributed Computing (DISC). This award is presented annually, with the presentation taking place alternately at PODC and DISC. The 2017 award will be presented at PODC 2017, to be held in Washington DC, USA.

The 2017 Principles of Distributed Computing Doctoral Dissertation Award Committee:

Cyril Gavoille (LaBRI, U. Bordeaux)
Boaz Patt-Shamir (Chair, Tel Aviv U.)
Michel Raynal (IRISA, U. Rennes 1)
Gadi Taubenfeld (IDC)

31st International Symposium on Distributed Computing (DISC 2017).
Editor: Andréa W. Richa
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Blockchain Consensus Protocols in the Wild*

Christian Cachin[1] and Marko Vukolić[2]

1 **IBM Research - Zürich, Rüschlikon, Switzerland**
`cca@zurich.ibm.com`
2 **IBM Research - Zürich, Rüschlikon, Switzerland**
`mvu@zurich.ibm.com`

Abstract

A blockchain is a distributed ledger for recording transactions, maintained by many nodes without central authority through a distributed cryptographic protocol. All nodes validate the information to be appended to the blockchain, and a consensus protocol ensures that the nodes agree on a unique order in which entries are appended. Consensus protocols for tolerating Byzantine faults have received renewed attention because they also address blockchain systems. This work discusses the process of assessing and gaining confidence in the resilience of a consensus protocols exposed to faults and adversarial nodes. We advocate to follow the established practice in cryptography and computer security, relying on public reviews, detailed models, and formal proofs; the designers of several practical systems appear to be unaware of this. Moreover, we review the consensus protocols in some prominent permissioned blockchain platforms with respect to their fault models and resilience against attacks.

1998 ACM Subject Classification C.2.4 Distributed Systems, D.1.3 Concurrent Programming

Keywords and phrases Permissioned blockchains, consensus, Byzantine fault-tolerance, snake oil, protocol analysis

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.1

Category Keynote talk

1 Introduction

Blockchains or *distributed ledgers* are systems that provide a trustworthy service to a group of *nodes* or parties that do not fully trust each other. They stand in the tradition of distributed protocols for secure multiparty computation in cryptography and replicated services tolerating Byzantine faults in distributed systems. Blockchains also contain many elements from cryptocurrencies, although a blockchain system can be conceived without a currency or value tokens. Generally, the blockchain acts as a trusted and dependable third party, for maintaining shared state, mediating exchanges, and providing a secure computing engine. Many blockchains can execute arbitrary tasks, typically called *smart contracts*, written in a domain-specific or a general-purpose programming language.

In a *permissionless blockchain*, such as Bitcoin or Ethereum, anyone can be a user or run a node, anyone can "write" to the shared state through invoking transactions (provided transaction fees are paid for), and anyone can participate in the consensus process for

* This work has been supported in part by the European Commission through the Horizon 2020 Framework Programme (H2020-ICT-2014-1) under grant agreements number 643964-SUPERCLOUD and 644579 ESCUDO-CLOUD and in part by the Swiss State Secretariat for Education, Research and Innovation (SERI) under contracts number 15.0091 and 15.0087.

© Christian Cachin and Marko Vukolić;
licensed under Creative Commons License CC-BY

determining the "valid" state. A *permissioned blockchain* in contrast, is operated by known entities, such as in *consortium blockchains*, where members of a consortium or stakeholders in a given business context operate a permissioned blockchain network. Permissioned blockchains systems have means to identify the nodes that can control and update the shared state, and often also have ways to control who can issue transactions. A *private blockchain* is a special permissioned blockchain operated by one entity, i.e., within one single trust domain.

Permissioned blockchains address many of the problems that have been studied in the field of distributed computing over decades, most prominently for developing *Byzantine fault-tolerant (BFT)* systems. Such blockchains can benefit from many techniques developed for reaching consensus, replicating state, broadcasting transactions and more, in environments where network connectivity is uncertain, nodes may crash or become subverted by an adversary, and interactions among nodes are inherently asynchronous. The wide-spread interest in blockchain technologies has triggered new research on practical distributed consensus protocols. There is also a growing number of startups, programmers, and industry groups developing blockchain protocols based on their own ideas, not relying on established knowledge.

The purpose of this paper is to give an overview of consensus protocols actually being used in the context of permissioned blockchains, to review the underlying principles, and to compare the resilience and trustworthiness of some protocols. We leave out permissionless (or "public") blockchains that are coupled to a cryptocurrency and their consensus protocols, such as *proof-of-work* or *proof-of-stake*, although this is a very interesting subject by itself.

Due to lack of space this paper contains only a part of the text of its *long version*, available online [18]. There we point out that developing consensus protocols is difficult and should not be undertaken in an ad-hoc manner. A resilient consensus protocol is only useful when it continues to deliver the intended service under a wide range of adversarial influence on the nodes and the network. Detailed analysis and formal argumentation are necessary to gain confidence that a protocol achieves its goal. In that sense, distributed computing protocols resemble cryptosystems and other security mechanisms; they require broad agreement on the underlying assumptions, detailed security models, formal reasoning, and wide-spread public discussion. Any claim for a "superior" consensus protocol that does not come with the necessary formal justification should be dismissed, analogously to the approach of "security by obscurity," which is universally rejected by experts.

Section 2 reviews the role of consensus for blockchain platforms. In Section 3, we discuss the consensus protocols of a number of *permissioned* blockchain platforms, based on the available product descriptions or source code. Different consensus mechanisms not directly following the BFT approach are found in the blockchain platforms *Sawtooth Lake*, *Ripple*, *Stellar*, and *IOTA Tangle*; they are discussed in the long version. Table 1 displays a summary of the discussed protocols.

2 Consensus

This section presents background and models for consensus in permissioned blockchains, first introducing the underlying concept of state-machine replication in Section 2.1. Sections 2.2 and 2.3 briefly review the most prominent family of protocols for this task, which is based Paxos/Viewstamped Replication (VSR) and PBFT. The essential step of transaction validation is discussed in Section 2.4. In the long version [18], we furthermore demonstrate the pitfalls of consensus-protocol design, by analyzing a proposed BFT consensus protocol called *Tangaroa* and showing that it does not achieve its goals.

2.1 Blockchains and consensus

A *blockchain* is a distributed database holding a continuously growing list of records, controlled by multiple entities that may not trust each other. Records are appended to the blockchain in batches or *blocks* through a distributed protocol executed by the nodes powering the blockchain. Each block contains a cryptographic hash of the previous block, which fixes all existing blocks and embeds a secure representation of the complete chain history into every block. Additional integrity measures are often used in potentially malicious, Byzantine environments, such as the requirement that a block hash is smaller than a given target (e.g., in Nakamoto-style proof-of-work consensus), or a multi-signature (or a threshold signature) over a block, by the nodes powering the blockchain (for permissioned blockchains). The nodes communicate over a network and collaboratively construct of the blockchain without relying on a central authority.

However, individual nodes might crash, behave maliciously, act against the common goal, or the network communication may become interrupted. For delivering a continuous service, the nodes therefore run a fault-tolerant *consensus protocol* to ensure that they all agree on the order in which entries are appended to the blockchain.

Since the whole blockchain acts as a trusted system, it should be *dependable*, *resilient*, and *secure*, ensuring properties such as availability, reliability, safety, confidentiality, integrity and more [4]. A blockchain protocol ensures this by *replicating* the data and the operations over many nodes. Replication can have many roles [52, 22, 35], but blockchains replicate data only for resilience, not for scalability. All nodes validate, in principle, the information to be appended to the blockchain; this feature stimulates the trust of all nodes in that the blockchain as a whole operates correctly.

For assessing a blockchain protocol, it is important to be clear about the underlying *trust assumption* or *security model*. This specifies the environment for which the protocol is designed and in which it satisfies its guarantees. Such assumptions should cover all elements in the system, including the network, the availability of synchronized clocks, and the expected (mis-)behavior of the nodes. For instance, the typical generic trust assumption for a system with n independent nodes says that no more than $f < n/k$ nodes become *faulty* (crash, leak information, perform arbitrary actions, and so on), for some $k = 2, 3, \ldots$. The other $n - f$ nodes are *correct*. A trust assumption always represents an idealization of the real world; if some aspect not considered by the model can affect the actually deployed system, then the security must be reconsidered.

State-machine replication. The formal study and development of algorithms for exploiting replication to build resilient servers and distributed systems goes back to Lamport et al.'s pioneering work introducing Byzantine agreement [48, 38]. The topic has evolved through a long history since and is covered in many textbooks [2, 14, 49, 58]; a good summary can be found in a "30-year perspective on replication" [22].

As summarized concisely by Schneider [52], the task of reaching and maintaining consensus among distributed nodes can be described with two elements: (1) a (deterministic) *state machine* that implements the logic of the service to be replicated; and (2) a *consensus protocol* to disseminate requests among the nodes, such that each node executes the *same sequence of requests* on its instance of the service. In the literature, "consensus" means traditionally only the task of reaching agreement on one single request (i.e., the first one), whereas "atomic broadcast" [31] provides agreement on a sequence of requests, as needed for state-machine replication. But since there is a close connection between the two (a sequence of consensus instances provides atomic broadcast), the term "consensus" more often actually stands for

atomic broadcast, especially in the context of blockchains. We adopt this terminology here and also use "transaction" and "request" as synonyms for one of the messages to be delivered in atomic broadcast.

Asynchronous and eventually synchronous models. Throughout this text, we assume the *eventual-synchrony* network model, introduced by Dwork et al. [26]. It models an asynchronous network that may delay messages among correct nodes arbitrarily, but eventually behaves synchronously and delivers all messages within a fixed (but unknown) time bound. Protocols in this model never violate their consistency properties (safety) during asynchronous periods, as long as the assumptions on the kind and number of faulty nodes are met. When the network stabilizes and behaves synchronously, then the nodes are guaranteed to terminate the protocol (liveness). Note that a protocol may stall during asynchronous periods; this cannot be avoided due to a fundamental discovery by Fischer et al. [27] (the celebrated "FLP impossibility result"), which rules out that deterministic protocols reach consensus in (fully) asynchronous networks.

The model is widely accepted today as realistic for designing resilient distributed systems. Replication protocols have to cope with network interruptions, node failures, system crashes, planned downtime, malicious attacks by participating nodes, and many more unpredictable effects. Developing protocols for asynchronous networks therefore provides the best possible resilience and avoids any assumptions about synchronized clocks and timely network behavior; making such assumptions can quickly turn into a vulnerability of the system if any one is not satisfied during deployment.

Protocol designers today prefer the *eventual synchrony* assumption for its simplicity and practitioners observe that it has broader coverage of actual network behavior, especially when compared to so-called partially synchronous models that assume probabilistic network behavior over time.

Consensus in blockchain. Although Nakamoto's Bitcoin paper [45] does not explicitly mention the *state-machine replication* paradigm [52], Bitcoin establishes consensus on one shared ledger based on voting among the nodes: "(Nodes) vote with their CPU power, expressing their acceptance of valid blocks by working on extending them and rejecting invalid blocks by refusing to work on them. Any needed rules and incentives can be enforced with this consensus mechanism" [45].

With the work of Garay et al. [28], a formal equivalence between the task solved by the "Nakamoto protocol" inside Bitcoin and the consensus problem in distributed computing was shown for the first time. This result coincided with the insight, developed in the fintech industry, that a blockchain platform may use a generic consensus mechanism and implement it with any protocol matching its trust model [55]. In today's understanding a blockchain platform may use an arbitrary consensus mechanism and retain most of its further aspects like distribution, cryptographic immutability, and transparency.

Existing consensus and replication mechanisms have therefore received renewed attention, for applying them to blockchain systems. Several protocols relevant for blockchains are reviewed in the next sections. We discuss only protocols for static groups here; they require explicit group reconfiguration [53, 7] and do not change membership otherwise. This assumption contrasts with view-synchronous replication [23], where the group composition may change implicitly by removing nodes perceived as unavailable.

2.2 Crash-tolerant consensus

As mentioned earlier, the form of consensus relevant for blockchain is technically known as *atomic broadcast*. It is formally obtained as an extension of a *reliable broadcast* among the node, which also provides a global or *total order* on the messages delivered to all correct nodes. An atomic broadcast is characterized by two (asynchronous) events *broadcast* and *deliver* that may occur multiple times. Every node may broadcast some message (or transaction) m by invoking *broadcast*(m), and the broadcast protocol outputs m to the local application on the node through a *deliver*(m) event.

Atomic broadcast ensures that each correct node outputs or *delivers* the same sequence of messages through the *deliver* events. More precisely [31, 14], it ensures these properties:

Validity: If a correct node p broadcasts a message m, then p eventually delivers m.

Agreement: If a message m is delivered by some correct node, then m is eventually delivered by every correct node.

Integrity: No correct node delivers the same message more than once; moreover, if a correct node delivers a message m and the sender p of m is correct, then m was previously broadcast by p.

Total order: For messages m_1 and m_2, suppose p and q are two correct nodes that deliver m_1 and m_2. Then p delivers m_1 before m_2 if and only if q delivers m_1 before m_2.

The most important and most prominent way to implement atomic broadcast (i.e., consensus) in distributed systems prone to $t < n/2$ node crashes is the family of protocols known today as *Paxos* [36, 37] and *Viewstamped Replication (VSR)* [46, 41]. Discovered independently, their core mechanisms exploit the same ideas [39, 40]. They have been implemented in dozens of mission-critical systems and power the core infrastructure of major cloud providers today [21].

The *Zab* protocol inside *ZooKeeper* is a prominent member of the protocol family; originally from Yahoo!, it is available as open source [33, 34, 56] (`https://zookeeper.apache.org/`) and used by many systems. A more recent addition to the family is *Raft* [47], a specialized variant developed with the aim of simplifying the understanding and the implementation of Paxos. It is contained in dozens of open-source tools (e.g., *etcd* – `https://github.com/coreos/etcd`).

All protocols in this family progress in a sequence of *views* or "epochs," with a unique leader for each view that is responsible for progress. If the leader fails, or more precisely, if the other nodes suspect that the leader has failed, they can replace the current leader by moving to the next view with a fresh leader. This *view change* protocol must ensure agreement, such that message already delivered by a node in the abandoned view is retained and delivered by all correct nodes in this or another future view.

2.3 Byzantine consensus

More recently, consensus protocols for tolerating *Byzantine* nodes have been developed, where nodes may be *subverted* by an adversary and act maliciously against the common goal of reaching agreement. In the eventual-synchrony model considered here, the most prominent protocol is *PBFT (Practical Byzantine Fault-Tolerance)* [19]. It can be understood as an extension of the Paxos/VSR family [39, 12, 40] and also uses a progression of views and a unique leader within every view. In a system with n nodes PBFT tolerates $f < n/3$ Byzantine nodes, which is optimal. Many research works have analyzed and improved aspects of it and made it more robust in prototypes [24].

Actual systems that implement PBFT or one of its variants are much harder to find than systems implementing Paxos/VSR. In fact, *BFT-SMaRt* (`https://github.com/bft-smart/library`) is the only known project that was developed before the interest in permissioned blockchains surged around 2015 [55]. Actually, Bessani et al. [7, 6] from the University of Lisbon started work on it around 2010. There is widespread agreement today that BFT-SMaRt is the most advanced and most widely tested implementation of a BFT consensus protocol available. Experiments have demonstrated that it can reach a throughput of about 80'000 transactions per second in a LAN [7] and very low latency overhead in a WAN [54].

Like Paxos/VSR, Byzantine consensus implemented by PBFT and BFT-SMaRt expects an eventually synchronous network to make progress. Without this assumption, only *randomized* protocols for Byzantine consensus are possible, such as the practical variations relying on distributed cryptography [16] as prototyped by SINTRA [17] or, much more recently, HoneyBadger [44].

2.4 Validation

In an atomic broadcast protocol resilient to crashes, every message is usually considered to be an acceptable request to the service. For Byzantine consensus, especially in blockchain applications, it makes sense to ask that only "valid" transactions are output by the broadcast protocol. To formalize this, the protocol is parameterized with a deterministic, external predicate $V()$, such that the protocol delivers only messages satisfying $V()$. This notion has been introduced as *external validity* by Cachin et al. [15].

The predicate must be deterministically computable locally by every process. More precisely, $V()$ must guarantee that when two correct nodes p and q in an atomic broadcast protocol have both delivered the same sequence of messages up to some point, then p obtains $V(m) = \textsc{True}$ for any message m if and only if q also determines that $V(m) = \textsc{True}$.

This combination of transaction validation and establishing consensus is inherent in permissionless blockchains based on proof-of-work consensus, such as Bitcoin and Ethereum. For permissioned-blockchain protocols, one could in principle also separate this step from consensus and perform the (deterministic) validation of transactions on the ordered "raw" sequence output by atomic broadcast. This could make the protocol susceptible to denial-of-service attacks from clients broadcasting excessively many invalid transactions. Hence most consensus protocols reviewed in this text combine ordering with validation and use a form of external validity based on the current blockchain state.

3 Permissioned blockchains

This section discusses some notable consensus protocols that are part of (or have at least been proposed for) the following consortium blockchain systems. We assume there are n nodes responsible for consensus, but some systems contain further nodes with other roles.

3.1 Overview

Among the recent flurry of blockchain-consensus protocols, many have not progressed past the stage of a paper-based description. In this section, we review only protocols *implemented* in a platform; the platform must either be available as open source or have been described in sufficient detail in marketing material. So far all implemented protocols discussed here assume independence among the failures, selfish behavior, and subversion of nodes. This

Table 1 Summary of consensus resilience properties, some of which use statically configured nodes with a *special* role. Symbols and notes: '✓' means that the protocol is resilient against the fault and '–' that it is not; '.' states that no such *special node* exists in the protocol; '?' denotes that the properties cannot be assessed due to lack of information; (✓) denotes the crash of *other* nodes, different from the special node; + MultiChain has non-final decisions; ⊕ PoET assumes trusted hardware available from only one vendor; ⊗ Ripple tolerates *one* of the five default Ripple-operated validators (special nodes) to be subverted. The last four protocols are discussed in the long version [18].

Which faults are tolerated by a protocol?	**Special-node crash**	**Any $t < n/2$ nodes crash**	**Special-node subverted**	**Any $f < n/3$ nodes subverted**
Hyperledger Fabric/Kafka	.	✓	.	–
Hyperledger Fabric/PBFT	.	✓	.	✓
Tendermint	.	✓	.	✓
Symbiont/BFT-SMaRt	.	✓	.	✓
R3 Corda/Raft	.	✓	.	–
R3 Corda/BFT-SMaRt	.	✓	.	✓
Iroha/Sumeragi (BChain)	.	✓	.	✓
Kadena/ScalableBFT	?	?	?	?
Chain/Federated Consensus	–	(✓)	–	–
Quorum/QuorumChain	–	(✓)	–	–
Quorum/Raft	.	✓	.	–
MultiChain +	.	✓	.	–
Sawtooth Lake/PoET	⊕	✓	⊕	–
Ripple	⊗	(✓)	⊗	–
Stellar/SCP	?	?	?	?
IOTA Tangle	?	?	?	?

justifies the choice of a *numeric* trust assumption, expressed only by a fraction of potentially faulty nodes.

It would be readily possible to extend such protocols to more complex fault assumptions, as formulated by *generic Byzantine quorum systems* [42]. For example, this would allow to run stake-based consensus (as done in some permissionless blockchains) or to express an arbitrary power structure formulated in a legal agreement for the consortium [11]. No platform offers this yet, however.

3.2 Hyperledger Fabric – Apache Kafka and PBFT

Hyperledger Fabric (`https://github.com/hyperledger/fabric`) is a platform for distributed ledger solutions, written in Golang and with a modular architecture that allows multiple implementations for its components. It is one of multiple blockchain frameworks hosted with the Hyperledger Project (`https://www.hyperledger.org/`) and aims at high degrees of confidentiality, resilience, flexibility, and scalability.

Following "preview" releases (*v0.5* and *v0.6*) in 2016, whose architecture [13] directly conforms to state-machine replication, a different and more elaborate design was adopted later

and is currently available in release *v1.0.0-beta*. The new architecture [1], termed *Fabric V1* here, separates the execution of smart-contract transactions (in the sense of validating the inputs and outputs of a program) from ordering transactions for avoiding conflicts (in the sense of an atomic broadcast that ensures consistency). This has several advantages, including better scalability, a separation of trust assumptions for transaction validation and ordering, support for non-deterministic smart contracts, partitioning of smart-contract code and data across nodes, and using modular consensus implementations [59].

The consensus protocol up to release *v0.6-preview* was a native implementation of PBFT [19]. With V1 the *ordering service* responsible for conflict-avoidance can be provided by an *Apache Kafka* cluster (`https://kafka.apache.org/`). Kafka is a distributed streaming platform with a publish/subscribe interface, aimed at high throughput and low latency. It logically consists of *broker nodes* and *consistency nodes*, where a set of redundant brokers processes each message stream and a ZooKeeper instance (`https://zookeeper.apache.org/`) running on the consistency nodes coordinates the brokers in case of crashes or network problems. Fabric therefore inherits is basic resilience against crashes from ZooKeeper. A second implementation of the ordering service is under development, which uses again the PBFT protocol and achieves resilience against subverted nodes. Besides, BFT-SMaRt (Sec. 2.3) is currently being integrated in Fabric V1 as one of the ordering services. Since BFT-SMaRt follows the well-established literature on Byzantine consensus protocols as mentioned earlier, its properties do not need special discussion here.

3.3 Tendermint

Tendermint Core (`https://github.com/tendermint/tendermint`) is a BFT protocol that can be best described as a variant of PBFT [19], as its common-case messaging pattern is a variant of Bracha's Byzantine reliable broadcast [8]. In contrast to PBFT, where the client sends a new transaction directly to all nodes, the clients in Tendermint disseminate their transactions to the validating nodes (or, simply, validators) using a gossip protocol. The external validity condition, evaluated within the Bracha-broadcast pattern, requires that a validator receives the transactions by gossip before it can vote for inclusion of the transaction in a block, much like in PBFT.

Tendermint's most significant departure from PBFT is the continuous rotation of the leader. Namely, the leader is changed after every block, a technique first used in BFT consensus space by the *Spinning* protocol [51]. Much like Spinning, Tendermint embeds aspects of PBFT's view-change mechanism into the common-case pattern. This is reflected in the following: while a validator expects the first message in the Bracha broadcast pattern from the leader, it also waits for a timeout, which resembles the view-change timer in PBFT. However, if the timer expires, a validator continues participating in the Bracha-broadcast message pattern, but votes for a *nil* block.

Tendermint as originally described by Buchman [9] suffers from a livelock bug, pertaining to locking and unlocking votes by validators in the protocol. However, the protocol contains additional mechanisms not described in the cited report that prevent the livelock from occurring [10]. While it appears to be sound, the Tendermint protocol and its implementation are still subject to a thorough, peer-reviewed correctness analysis.

3.4 Symbiont – BFT-SMaRt

Symbiont Assembly (`https://symbiont.io/technology/assembly`) is a proprietary distributed ledger platform. The company that stands behind it, Symbiont, focuses on applications

of distributed ledgers in the financial industry, providing automation for modeling and executing complex instruments among institutional market participants.

Assembly implements resilient consensus in its platform based on the open-source BFT-SMaRt toolkit (Sec. 2.3). Symbiont uses its own reimplementation of BFT-SMaRt in a different programming language; it reports performance numbers of 80'000 transactions per second (tps) using a 4-node cluster on a LAN. This matches the throughput expected from BFT-SMaRt [7] and similar results in the research literature on BFT protocols [3].

Assembly uses the standard resilience assumptions for BFT consensus in the eventually-synchronous model considered here.

3.5 R3 Corda – Raft and BFT-SMaRt

Unlike most of the other permissioned blockchain platforms discussed here, *Corda* (`https://github.com/corda/corda`) does not order all transactions as one single virtual execution that forms the blockchain. Instead, it defines *states* and *transactions*, where every transaction consumes (multiple) states and produces a new state [32]. Only nodes affected by a transaction store it. Seen across all users, this transaction execution model produces a hashed directed acyclic graph or *Hash-DAG*. Transactions must be *valid*, i.e., endorsed by the issuer and other affected nodes and correct according to the underlying smart-contract logic governing the state. Each state points to a *notary* responsible for ensuring transaction *uniqueness*, i.e., that each state is consumed only once. The notary is a logical service that can be provided jointly by multiple nodes. The *type* of a state may designate an asset represented by the network, such as a token or an obligation, or anything else controlled by a smart contract.

A transaction in Corda consumes only states controlled by the same notary; hence, one notary by itself can atomically verify the transaction's validity and uniqueness to decide whether it is executed or not. To enable transactions that operate across states governed by different notaries, there is a specialized transaction that changes the notary, such that one notary will become responsible for validating the transaction.

Since a node stores only a part of the Hash-DAG, it only knows about transactions and states that concern the node. This contrasts with most other distributed ledgers and provides means for partitioning the data among the nodes. As is the case for other smart-contract platforms, transactions refer to contracts that can be programmed in a universal general-purpose language.

A notary service in Corda orders and timestamps transactions that include states pointing to it. "Notaries are expected to be composed of multiple mutually distrusting parties who use a standard consensus algorithm" (`https://docs.corda.net`). A notary service needs to cryptographically sign its statements of transaction uniqueness, such that other nodes in the network can rely on its assertions without directly talking to the notary. Currently there is support for running a notary service as a single node (centralized), for running a distributed crash-tolerant implementation using Raft (Sec. 2.2), and for distributing it using the open-source BFT-SMaRt toolkit (Sec. 2.3). When using Raft deployed on n nodes, a Corda notary tolerates crashes of any $t < n/2$ of these nodes (Sec. 2.2). With BFT-SMaRt running on n nodes, the notary is resilient to the subversion of $f < n/3$ nodes.

3.6 Hyperledger Iroha – Sumeragi

Hyperledger Iroha (`https://github.com/hyperledger/iroha`) is another open-source blockchain platform developed under the Hyperledger Project. Its architecture is inspired by the original (v0.6) design of Fabric (Sec. 3.2). All validating nodes collaboratively execute a

Byzantine consensus protocol. In that sense it is also similar to Tendermint and Symbiont Assembly.

The *Sumeragi* consensus library of Iroha is "heavily inspired" by BChain [25] a chain-style Byzantine replication protocol that propagates transactions among the nodes with a "chain" topology. Chain replication [57, 22] arranges the n nodes linearly and each node normally only receives messages from its predecessor and sends messages to its successor. Although there is a leader at the head of the chain, like in many other protocols, the leader does not become a bottleneck since it usually communicates only with the head and the tail of the chain, but not with all n nodes. This balances the load among the nodes and lets chain-replication protocols achieve the best possible throughput [30, 3], at the cost of higher normal-case latency and slightly increased time for reconfiguration after faults.

In Sumeragi, the order of the nodes is determined based on a reputation system, which takes the "age" of a node and its past performance into account.

As becomes apparent from the documentation (`https://github.com/hyperledger/iroha/wiki/Sumeragi`), though, the protocol departs from the "chain" pattern, because the leader "broadcasts" to all nodes and so does the node at the tail. Hence, it is neither BChain nor chain replication. Assuming that Sumeragi would correctly implement BChain, then it relies on the standard assumptions for BFT consensus in the eventually-synchronous model, just like Fabric v0.6, Tendermint, and Symbiont.

3.7 Kadena – Juno and ScalableBFT

Juno from *kadena* (`https://github.com/kadena-io/juno`) is a platform for running smart contracts that has been developed until about November 2016 according to its website. Juno claims to use a "Byzantine Fault Tolerant Raft" protocol for consensus and appears to address the standard BFT model with n nodes, $f < n/3$ Byzantine faults among them, and eventual synchrony [26] as timing assumption. Later Juno has been deprecated in favor of a "proprietary BFT-consensus protocol" called *ScalableBFT* [43], which is "inspired by the Tangaroa protocol" and optimizes performance compared to Juno and Tangaroa. The whitepaper cites over 7000 transactions per second (tps) throughput on a cluster with size 256 nodes.

The design and implementation of ScalableBFT are proprietary and not available for public review. Being based on Tangaroa, the design might suffer from its devastating problems mentioned in Section 2.3. Further statements about ScalableBFT made in a blog post [50] do not enhance the trust in its safety: "Every transaction is replicated to every node. When a majority of nodes have replicated the transaction, the transaction is committed." As is well-known from the literature [22, 14] in the model considered here, with public-key cryptography for message authentication and asynchrony, agreement in a consensus protocol can only be ensured with $n > 3f$ and Byzantine quorums [42] of size *strictly larger* than $\frac{n+f}{2}$, which reduces to $2f + 1$ with $n = 3f + 1$ nodes. Hence "replicating among a majority" does *not* suffice.

The claimed performance number of more than 7000 tps is in line with the throughput of 30'000–80'000 tps, as reported by a representative state-of-the-art BFT protocol evaluation in the literature [3]. However, since Juno is proprietary, it is not not clear how it actually works nor why one should trust it, as discussed before. One should rather build on established consensus approaches and publicly validated algorithms than on a proprietary protocol for resilience. As the resilience of Juno and ScalableBFT cannot be assessed, it remains unclear whether it actually provides consensus as intended.

3.8 Chain – Federated Consensus

The *Chain Core* platform (`https://chain.com`) is a generic infrastructure for an institutional consortium to issue and transfer financial assets on permissioned blockchain networks. It focuses on the financial services industry and supports multiple different assets within the same network.

The *Federated Consensus* [20] protocol of Chain Core is executed by the n nodes that make up the network. One of the nodes is statically configured as "block generator." It periodically takes a number of new, non-executed transactions, assembles them into blocks, and submits the block for approval to "block signers." Every signer validates the block proposed for a given block height, checking the signature of the generator, validating the transactions, and verifying some real-time constraints and then signs an endorsement for the block. Each signer endorses only one block at each height. Once a node receives q such endorsements for a block, the node appends the block to its chain.

The protocol is resilient to a number of malicious (Byzantine-faulty) signers but not to a malicious block generator. If the block generator violates the protocol (e.g., by signing two different blocks for the same block height) the ledger might fork (i.e., the consensus protocol violates safety). The documentation states that such misbehavior should be addressed by retaliation and measures for this remain outside the protocol.

More specifically, when assuming the block generator operates correctly and is live, this Federated Consensus reduces to an ordinary Byzantine quorum system that tolerates f faulty signer nodes when $q = 2f + 1$ and $n = 3f + 1$; its use for consensus is similar, say, to the well-understood "authenticated echo broadcast" [14, Sec. 3.10.3]. Up to f block signers may behave arbitrarily, such as by endorsing incorrect transactions or by refusing to participate, and the protocol will remain live and available (with the correct block generator).

Overall, however, Federated Consensus is a special case of a standard BFT-consensus protocol that appears to operate with a fixed "leader" (in the role of the block generator). The protocol *cannot* prevent forks if the generator is malicious. Even if the generator simply crashes, the protocol halts and requires manual intervention. Standard BFT protocols instead will tolerate leader corruption and automatically switch to a different leader if it becomes apparent that one leader malfunctions.

Since the block generator must be correct, the purpose of a signature issued by a block signer remains unclear, at least at the level of the consensus protocol. The only reason appears to be guaranteeing that the signer cannot later repudiate having observed a block.

3.9 Quorum – QuorumChain and Raft

Quorum (`https://github.com/jpmorganchase/quorum`), mainly from developers at JP-Morgan Chase, is an enterprise-focused version of Ethereum, executing smart contracts with the Ethereum virtual machine, but using an alternative to the default proof-of-work consensus protocol of the public Ethereum blockchain. The platform currently contains two consensus protocols, called *QuorumChain* and *Raft-based consensus*.

QuorumChain. This protocol uses a smart contract to validate blocks. The trust model specifies a set of n "voter" nodes and some number of "block-maker" nodes, whose identities are known to all nodes. The documentation remains unclear about the trust model, not clearly expressing in which ways one or more of these nodes might fail or behave adversarially. (One can draw some conclusions from the protocol though.)

The protocol uses the standard peer-to-peer gossip layer of Ethereum to propagate blocks and votes on blocks, but the logic itself is formulated as a smart contract deployed with the genesis block. Nodes digitally sign every message they send. Only block-maker nodes are permitted to propose block to be appended; nodes with voter role validate blocks and express their approval by a (yes) vote. A block-maker waits for a randomly chosen time and then creates, signs, and propagates a new block that extends its own chain. A voter will validate the block (by executing its transactions and checking its consistency), "vote" on it, and propagate this. A voter apparently votes for every received block that is valid and extends its own chain, and it may vote multiple times for a given block height. Voting continues for a period specified in real time. Each node accepts and extends its own chain with the block that obtains more votes than a given threshold, and if there are multiple ones, the one with most votes. There is one block-maker node by default.

To assess the resilience of the protocol, it is obvious that already one malicious block-maker node can easily create inconsistencies (chain forks) unless the network is perfect and already provides consensus. With one block-maker, if this node crashes, the protocol halts. Depending on how the operator sets the voting threshold and on the network connectivity, it may fork the chain with only two block-makers and without any Byzantine fault. With a Byzantine fault in a block-maker node or a voter node can disrupt the protocol and also create inconsistencies. Furthermore, the protocol relies on synchronized clocks for safety and liveness. Taken together, the protocol cannot ensure consensus in any realistic sense.

Raft-based consensus. The second and more recent consensus option available for Quorum is based on the Raft protocol [47], which is a popular variant of Paxos [36] available in many open-source toolkits. Quorum uses the implementation in *etcd* (`https://github.com/coreos/etcd`) and co-locates every Quorum-node with an etcd-node (itself running Raft). Raft will replicate the transactions to all participating nodes and ensure that each node locally outputs the same sequence of transactions, despite crashes of nodes. The deployment actually tolerates that any $t < n/2$ of the n etcd-nodes may crash. Raft relies on timeliness and synchrony only for liveness, not for safety.

This is a canonical design, directly interpreting the replication of Quorum smart contracts as a replicated state machine. It seems appropriate for a protected environment, which is not subject to adversarial nodes.

3.10 MultiChain

The *MultiChain* platform (`https://github.com/MultiChain/multichain`) is intended for permissioned blockchains in the financial industry and for multi-currency exchanges in a consortium, aiming at compatibility with the Bitcoin ecosystem as much as possible.

MultiChain uses a dynamic permissioned model [29]: There is a list of permitted nodes in the network at all times, identified by their public keys. The list can be changed through transactions executed on the blockchain, but at all times, only nodes on this list validate blocks and participate in the protocol.

As the MultiChain platform is derived from Bitcoin, its consensus mechanism is called "mining" [29]; however, in the permissioned model, the nodes do not solve computational puzzles. Instead, any permitted node may generate new blocks after waiting for a random timeout, subject to a *diversity* parameter $\rho \in [0, 1]$ that constrains the acceptable miners for a given block height. More precisely, if the permitted list has length L, then a block proposal from a node is only accepted if the blockchain held by the validating node does not already contain a block generated by the *same* node among the $\lceil \rho L \rceil$ most recent blocks.

Any participating node will extend its blockchain with the first valid block of this kind that it receives, and if it learns about different, conflicting chain extensions, it will select the longer one (as in Bitcoin). Furthermore, a well-behaved node will not generate a new block if its own chain already contains a block of his within the last $\lceil \rho L \rceil$ blocks.

It appears that the random timeouts and network uncertainty easily lead to forks in the ledger, even if all nodes are correct. If two different nodes may generate a valid block at roughly the same time, and any other node will append the one of which it hears first to its chain, then these two nodes will be forked. This is not different from consensus in Bitcoin and will eventually converge to a single chain if all nodes follow the protocol. However, if a single attacking node generates transactions and blocks as it wants, and assuming that the network behaves favorably for the attack, the node can take over the entire network and revert arbitrarily many past transactions (in the same way as a "51%-attack" in Bitcoin).

Hence, MultiChain exhibits non-final transactions similar to any proof-of-work consensus. But whereas lack of finality appears to be a consequence of the public nature of proof-of-work, and since MultiChain is permissioned, forks and non-final decisions could be avoided here completely. The traditional consensus protocols for this model, discussed in Sections 2.2 and 2.3, all reach consensus with finality. In the model of non-final consensus decisions, with the corresponding delays and throughput constraints, the MultiChain consensus protocol can only remain consistent and live with one single correct node.

3.11 Further platforms

Another recent extension of the Ethereum platform is *HydraChain* (`https://github.com/HydraChain/hydrachain/blob/develop/README.md`), which adds support for creating a permissioned distributed ledger using the Ethereum infrastructure. The repository describes a proprietary consensus protocol "initially inspired by Tendermint." Without clear explanation of the protocol and formal review of its properties, its correctness remains unclear.

The *Swirlds hashgraph algorithm* is built into a proprietary "distributed consensus platform" (`https://www.swirlds.com`); a white paper is available [5] and the protocol is also implemented in an open-source consensus platform for distributed applications, called *Babble* (`https://github.com/babbleio/babble`). It targets consensus for a permissioned blockchain with n nodes and $f < n/3$ Byzantine faults among them, i.e., the standard Byzantine consensus problem according to Section 2.3. In contrast to PBFT and other protocols discussed there, it operates in a "completely asynchronous" model. The white paper states arguments for the safety and liveness of the protocol and explains that hashgraph consensus is randomized to circumvent the FLP impossibility [27]. Since the algorithm is guaranteed to reach agreement on a binary decision (i.e., with only 0/1 outcomes) only with exponentially small probability in n [5, Thm. 5.16], it appears similar to Ben-Or-style randomized agreement [14, Sec. 5.5]. However, no independent validation or analysis of hashgraph consensus is available.

Acknowledgments. We thank Andreas Kind, Pedro Moreno-Sanchez, and Björn Tackmann for interesting discussions and valuable comments.

References

1 Elli Androulaki, Christian Cachin, Konstantinos Christidis, Chet Murthy, Binh Nguyen, and Marko Vukolić. Next consensus architecture proposal. Hyperledger Wiki, Fab-

ric Design Documents, available at https://github.com/hyperledger/fabric/blob/master/proposals/r1/Next-Consensus-Architecture-Proposal.md, 2016.

2 Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. Wiley, second edition, 2004.

3 Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knezevic, Vivien Quéma, and Marko Vukolic. The next 700 BFT protocols. *ACM Transactions on Computer Systems*, 32(4):12:1–12:45, 2015.

4 Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

5 Leemon Baird. The Swirlds hashgraph consensus algorithm: Fair, fast, Byzantine fault tolerance. Swirlds Tech Report SWIRLDS-TR-2016-01, available online, http://www.swirlds.com/developer-resources/whitepapers/, 2016.

6 Alysson Bessani and João Sousa. From Byzantine consensus to BFT state machine replication: A latency-optimal transformation. In *Proc. 9th European Dependable Computing Conference*, pages 37–48, 2012.

7 Alysson Neves Bessani, João Sousa, and Eduardo Adílio Pelinson Alchieri. State machine replication for the masses with BFT-SMaRt. In *Proc. 44th International Conference on Dependable Systems and Networks*, pages 355–362, 2014.

8 Gabriel Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75:130–143, 1987.

9 Ethan Buchman. Tendermint: Byzantine fault tolerance in the age of blockchains. M.Sc. Thesis, University of Guelph, Canada, June 2016.

10 Ethan Buchman and Jae Kwon. Private discussion, 2017.

11 Christian Cachin. Distributing trust on the Internet. In *Proc. International Conference on Dependable Systems and Networks (DSN-DCCS)*, pages 183–192, 2001.

12 Christian Cachin. Yet another visit to Paxos. Research Report RZ 3754, IBM Research, November 2009.

13 Christian Cachin. Architecture of the Hyperledger blockchain fabric. Workshop on Distributed Cryptocurrencies and Consensus Ledgers (DCCL 2016), 2016. URL: https://www.zurich.ibm.com/dccl/papers/cachin_dccl.pdf.

14 Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming (Second Edition)*. Springer, 2011.

15 Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols (extended abstract). In Joe Kilian, editor, *Advances in Cryptology: CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 524–541. Springer, 2001.

16 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.

17 Christian Cachin and Jonathan A. Poritz. Secure intrusion-tolerant replication on the Internet. In *Proc. International Conference on Dependable Systems and Networks (DSN-DCCS)*, pages 167–176, June 2002.

18 Christian Cachin and Marko Vukolić. Blockchain consensus protocols in the wild. e-print, arXiv:1707.01873 [cs.DC], 2017. URL: https://arxiv.org/abs/1707.01873.

19 Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, November 2002.

20 Chain protocol whitepaper. Available online, https://chain.com/docs/1.2/protocol/papers/whitepaper, 2017.

21 Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *Proc. 26th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 398–407, 2007.

22 Bernadette Charron-Bost, Fernando Pedone, and André Schiper, editors. *Replication: Theory and Practice*, volume 5959 of *Lecture Notes in Computer Science*. Springer, 2010.

23 Gregory V. Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 33(4):427–469, 2001.

24 Allen Clement, Edmund L. Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proc. 6th Symp. Networked Systems Design and Implementation (NSDI)*, pages 153–168, 2009.

25 Sisi Duan, Hein Meling, Sean Peisert, and Haibin Zhang. BChain: Byzantine replication with high throughput and embedded reconfiguration. In Marcos K. Aguilera, Leonardo Querzoni, and Marc Shapiro, editors, *Proc. 18th Conference on Principles of Distributed Systems (OPODIS)*, volume 8878 of *Lecture Notes in Computer Science*, pages 91–106. Springer, 2014.

26 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

27 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

28 Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Advances in Cryptology: Eurocrypt 2015*, volume 9057 of *Lecture Notes in Computer Science*, pages 281–310. Springer, 2015.

29 Gideon Greenspan. Multichain private blockchain — White paper. `http://www.multichain.com/download/MultiChain-White-Paper.pdf`, 2016.

30 Rachid Guerraoui, Ron R. Levy, Bastian Pochon, and Vivien Quéma. Throughput optimal total order broadcast for cluster environments. *ACM Transactions on Computer Systems*, 28(2):5:1–5:32, 2010.

31 Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcasts and related problems. In Sape J. Mullender, editor, *Distributed Systems (2nd Ed.)*. ACM Press & Addison-Wesley, New York, 1993.

32 Mike Hearn. Corda: A distributed ledger. Available online, `https://docs.corda.net/_static/corda-technical-whitepaper.pdf`, 2016.

33 Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for internet-scale systems. In *Proc. USENIX Annual Technical Conference*, 2010.

34 Flavio Junqueira, Benjamin Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proc. 41st International Conference on Dependable Systems and Networks*, 2011.

35 Martin Kleppmann. *Designing Data-Intensive Applications*. O'Reilly Media, 2017.

36 Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

37 Leslie Lamport. Paxos made simple. *SIGACT News*, 32(4):51–58, 2001.

38 Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

39 Butler Lampson. The ABCD's of Paxos. In *Proc. 20th ACM Symposium on Principles of Distributed Computing (PODC)*, 2001.

40 Barbara Liskov. From viewstamped replication to Byzantine fault tolerance. In Bernadette Charron-Bost, Fernando Pedone, and André Schiper, editors, *Replication: Theory and Practice*, volume 5959 of *Lecture Notes in Computer Science*, pages 121–149. Springer, 2010.

41 Barbara Liskov and James Cowling. Viewstamped replication revisited. MIT-CSAIL-TR-2012-021, July 2012.

42 Dahlia Malkhi and Michael K. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

43 Will Martino. Kadena — The first scalable, high performance private blockchain. Whitepaper, `http://kadena.io/docs/Kadena-ConsensusWhitePaper-Aug2016.pdf`, 2016.

44 Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of BFT protocols. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, 2016.

45 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Whitepaper, 2009. `http://bitcoin.org/bitcoin.pdf`.

46 Brian M. Oki and Barbara Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proc. 7th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 8–17, 1988.

47 Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In *Proc. USENIX Annual Technical Conference*, pages 305–319, 2014.

48 M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

49 Michel Raynal. *Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool, 2010.

50 George Samman. Kadena: The first real private blockchain. `http://sammantics.com/blog/2016/11/29/kadena-the-first-real-private-blockchain`, November 2016.

51 Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. Spin one's wheels? Byzantine fault tolerance with a spinning primary. In *Proc. 28th Symposium on Reliable Distributed Systems (SRDS)*, pages 135–144, 2009.

52 Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

53 Alexander Shraer, Benjamin Reed, Dahlia Malkhi, and Flavio Paiva Junqueira. Dynamic reconfiguration of primary/backup clusters. In *Proc. USENIX Annual Technical Conference*, pages 425–437, 2012.

54 João Sousa and Alysson Bessani. Separating the WHEAT from the chaff: An empirical design for geo-replicated state machines. In *Proc. 34th Symposium on Reliable Distributed Systems (SRDS)*, pages 146–155, 2015.

55 Tim Swanson. Consensus-as-a-service: A brief report on the emergence of permissioned, distributed ledger systems. Report, available online, April 2015. URL: `http://www.ofnumbers.com/wp-content/uploads/2015/04/Permissioned-distributed-ledgers.pdf`.

56 Robbert van Renesse, Nicolas Schiper, and Fred B. Schneider. Vive la différence: Paxos vs. viewstamped replication vs. zab. *IEEE Transactions on Dependable and Secure Computing*, 12(4):472–484, 2015.

57 Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Proc. 6th Symp. Operating Systems Design and Implementation (OSDI)*, 2004.

58 Marko Vukolić. *Quorum Systems: With Applications to Storage and Consensus*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool, 2012.

59 Marko Vukolić. Rethinking permissioned blockchains. In *Proc. ACM Workshop on Blockchain, Cryptocurrencies and Contracts (BCC'17)*, 2017.

Recommenders: from the Lab to the Wild

Anne-Marie Kermarrec

Mediego/Inria France, EPFL Swizerland
anne-marie.kermarrec@mediego.com

Abstract

Recommenders are ubiquitous on the Internet today: they tell you which book to read, which movie you should watch, predict your next holiday destination, give you advices on restaurants and hotels, they are even responsible for the posts that you see on your favorite social media and potentially greatly influence your friendship on social networks.

While many approaches exist, collaborative filtering is one of the most popular approaches to build online recommenders that provide users with content that matches their interest. Interestingly, the very notion of users can be general and span actual humans or software applications. Recommenders come with many challenges beyond the quality of the recommendations. One of the most prominent ones is their ability to scale to a large number of users and a growing volume of data to provide real-time recommendations introducing many system challenges. Another challenge is related to privacy awareness: while recommenders rely on the very fact that users give away information about themselves, this potentially raises some privacy concerns.

In this talk, I will focus on the challenges associated to building efficient, scalable and privacy-aware recommenders.

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases Recommenders, Collaborative filtering, Distributed systems

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.2

Category Keynote talk

© Anne-Marie Kermarrec;
licensed under Creative Commons License CC-BY
31st International Symposium on Distributed Computing (DISC 2017).
Editor: Andréa W. Richa; Article No. 2; pp. 2:1–2:1

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

Phase Transitions and Emergent Phenomena in Random Structures and Algorithms*

Dana Randall

School of Computer Science, Georgia Institute of Technology, Atlanta, GA, USA
randall@cc.gatech.edu

Abstract

Markov chain Monte Carlo methods have become ubiquitous across science and engineering to model dynamics and explore large sets of configurations. The idea is to perform a random walk among the configurations so that even though only a very small part of the space is visited, samples will be drawn from a desirable distribution. Over the last 20 years there have been tremendous advances in the design and analysis of efficient sampling algorithms for this purpose, building on insights from statistical physics. One of the striking discoveries has been the realization that many natural Markov chains undergo phase transitions, whereby they change from being efficient to inefficient as some parameter of the system is modified, also revealing interesting properties of the underlying random structures.

We will explore how phase transitions can provide valuable insights in three settings. First, they allow us to understand the limitations of certain classes of sampling algorithms, potentially leading to faster alternative approaches. Second, they reveal statistical properties of stationary distributions, giving insight into various interacting models. Example include colloids, or binary mixtures of molecules, segregation models, where individuals are more likely move when they are unhappy with their local demographics, and interacting particle systems from statistical physics. Last, they predict emergent phenomena that can be harnessed for the design of distributed algorithms for certain asynchronous models of programmable active matter. We will see how these three research threads are closely interrelated and inform one another.

The talk will take a random walk through some of the results included in the references.

1998 ACM Subject Classification F.2 Analysis of Algorithms and Problem Complexity, G.2.1. Combinatorics/Counting Problems, G.3. Probability and Statistics – probabilistic algorithms (including Monte Carlo)

Keywords and phrases Markov chains, phase transitions, sampling, emergent phenomena, programmable matter

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.3

Category Keynote talk

References

1 M. Andres Arroyo, S. Cannon, J.J. Daymude, D. Randall, and A.W. Richa. A stochastic approach to shortcut bridging in programmable matter. In *Proc. 23rd International Conference on DNA Computing and Molecular Programming* (DNA), 2017.

2 P. Bhakta, S. Miracle, and D. Randall. Clustering and mixing times for segregation models on $\mathbb{Z}^2$. In *Proc. 25th ACM-SIAM Symposium on Discrete Algorithms* (SODA), 2014.

* This work was supported in part by NSF CCF-1526900 and CCF-1637031.

© Dana Randall;
licensed under Creative Commons License CC-BY

3 A. Blanca, Y. Chen, D. Galvin, D. Randall, and P. Tetali. Phase coexistence for the hard-core model on $\mathbb{Z}^2$. *Submitted*, 2017.

4 S. Cannon, J.J. Daymude, D. Randall, and A.W. Richa. A markov chain algorithm for compression in self-organizing particle systems. In *Proc. 2016 ACM Symposium on Principles of Distributed Computing* (PODC), pages 279–288, 2016.

5 E. Lubetzky and A. Sly. Critical ising on the square lattice mixes in polynomial time. *Communications in Mathematical Physics*, 313:815–836, 2012.

6 F. Martinelli. Lectures on glauber dynamics for discrete spin models, lectures on probability theory and statistics. Lecture Notes in Math., page 93–191, 1999.

7 S. Miracle, D. Randall, and A. Streib. Clustering in interfering binary mixtures. In *Proceedings of the 14th International Workshop and 15th International Conference on Approximation, Randomization, and Combinatorial Optimization: Algorithms and Techniques* (RANDOM), pages 652–663, 2011.

8 D. Randall. Rapidly mixing markov chains with applications in computer science and physics. *Computing in Science and Engineering*, 8:30–41, 2006.

9 R. Restrepo, J. Shin, P. Tetali, E. Vigoda, and L. Yang. Improving mixing conditions on the grid for counting and sampling independent sets. *Probability Theory and Related Fields*, 156:75–99, 2013.

Reuse, Don't Recycle: Transforming Lock-Free Algorithms That Throw Away Descriptors*†

Maya Arbel-Raviv‡[1] **and Trevor Brown**[2]

1 **Technion, Computer Science Department, Haifa, Israel**
mayaarl@cs.technion.ac.il
2 **Technion, Computer Science Department, Haifa, Israel**
me@tbrown.pro

Abstract

In many lock-free algorithms, threads help one another, and each operation creates a *descriptor* that describes how other threads should help it. Allocating and reclaiming descriptors introduces significant space and time overhead. We introduce the first *descriptor* abstract data type (ADT), which captures the usage of descriptors by lock-free algorithms. We then develop a *weak descriptor* ADT which has weaker semantics, but can be implemented significantly more efficiently. We show how a large class of lock-free algorithms can be transformed to use weak descriptors, and demonstrate our technique by transforming several algorithms, including the leading k-compare-and-swap (k-CAS) algorithm. The original k-CAS algorithm allocates at least $k+1$ new descriptors *per k-CAS*. In contrast, our implementation allocates two descriptors *per process*, and each process simply reuses its two descriptors. Experiments on a variety of workloads show significant performance improvements over implementations that reclaim descriptors, and reductions of up to three orders of magnitude in peak memory usage.

1998 ACM Subject Classification D.1.3 Concurrent Programming

Keywords and phrases Concurrency, data structures, lock-free, synchronization, descriptors

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.4

1 Introduction

Many concurrent data structures use locks, but locks have downsides, such as susceptibility to convoying, deadlock and priority inversion. Lock-free data structures avoid these downsides, and can be quite efficient. They guarantee that some process will always makes progress, even if some processes halt unexpectedly. This guarantee is typically achieved with *helping*, which allows a process to harness any time that it would otherwise spend waiting for another operation to complete. Specifically, whenever a process p is prevented from making progress by another operation, it attempts to perform some (or all) of the work of the other operation, on behalf of the process that started it. This way, even if the other process has crashed, its operation can be completed, so that it no longer blocks p.

In simple lock-free data structures (e.g., [27, 13, 22, 25]), a process can determine how to help an operation that blocks it by inspecting a small part of the data structure. In more complex lock-free data structures [12, 16, 26, 10], processes publish *descriptors* for their

* This work was supported by the Israel Science Foundation (grant 1749/14), the Natural Sciences and Engineering Research Council of Canada, and Global Affairs Canada.
† A full version of the paper is available at http://www.cs.utoronto.ca/~tabrown/desc/fullpaper.pdf.
‡ Maya Arbel-Raviv is supported in part by the Technion Hasso Plattner Institute Research School.


© Maya Arbel-Raviv and Trevor Brown;
licensed under Creative Commons License CC-BY

operations, and helpers look at these descriptors to determine how to help. A descriptor typically encodes a sequence of steps that a process should follow in order to complete the operation that created it.

Since lock-free algorithms cannot use mutual exclusion, many helpers can simultaneously help an operation, potentially long after the operation has terminated. Thus, to avoid situations where helpers read inconsistent data in a descriptor and corrupt the data structure, each descriptor must remain consistent and accessible until no helper will ever access it again. This leads to *wasteful algorithms* which allocate a new descriptor for each operation.

In this work, we introduce two simple abstract data types (ADTs) that capture the way descriptors are used by wasteful algorithms (in Section 2). The *immutable descriptor* ADT provides two operations, *CreateNew* and *ReadField*, which respectively create and initialize a new descriptor, and read one of its fields. The *mutable descriptor* ADT extends the immutable descriptor ADT by adding two operations: *WriteField* and *CASField*. These allow a helper to modify fields of the descriptor (e.g., to indicate that the operation has been partially or fully completed).

The natural way to implement the immutable and mutable descriptor ADTs is to have *CreateNew* allocate memory and initialize it, and to have *ReadField*, *WriteField* and *CASField* perform a read, write and CAS, respectively. Every implementation of one of these ADTs must eventually reclaim the descriptors it allocates. Otherwise, the algorithm would eventually exhaust memory. We briefly explain why reclaiming descriptors is expensive.

In order to safely free a descriptor, a process must know that the descriptor is no longer *reachable*. This means no other process can reach the descriptor by following pointers in shared memory *or* in its private memory. State of the art lock-free memory reclamation algorithms such as hazard pointers [23] and DEBRA+ [6] can determine when no process has a pointer in its *private* memory to a given object, but they typically require the underlying algorithm to identify a time t after which the object is no longer reachable from *shared* memory. In an algorithm where each operation removes all pointers to its descriptor from shared memory, t is when O completes. However, in some algorithms (e.g., [9]), pointers to descriptors are "lazily" cleaned up by subsequent operations, so t may be difficult to identify. The overhead of reclaiming descriptors comes both from identifying t, and from actually running a lock-free memory reclamation algorithm.

Additionally, in some applications, such as embedded systems, it is important to have a small, predictable number of descriptors in the system. In such cases, one must use memory reclamation algorithms that aggressively reclaim memory to minimize the number of objects that are waiting to be reclaimed at any point in time. Such algorithms incur high overhead. For example, hazard pointers can be used to maintain a small memory footprint, but a process must perform costly memory fences *every* time it tries to access a new descriptor.

To circumvent the aforementioned problems, we introduce a *weak descriptor* ADT (in Section 3) that has slightly *weaker semantics* than the mutable descriptor ADT, but can be implemented *without memory reclamation*. The crucial difference is that each time a process invokes *CreateNew* to create a new descriptor, it *invalidates* all of its previous descriptors. An invocation of *ReadField* on an invalid descriptor *fails* and returns a special value $\perp$. Invocations of *WriteField* and *CASField* on invalid descriptors have no effect. We believe the weak descriptor ADT can be useful in designing new lock-free algorithms, since an invocation of *ReadField* that returns $\perp$ can be used to inform a helper that it no longer needs to continue helping (making further accesses to the descriptor unnecessary).

We also identify a class of lock-free algorithms that use the descriptor ADT, and which can be *transformed* to use the weak descriptor ADT (in Section 3.1). At a high level, these are algorithms in which (1) each operation creates a descriptor and invokes a *Help*

function on it, and (2) *ReadField*, *WriteField* and *CASField* operations occur only inside invocations of *Help*. Intuitively, the fact that these operations occur only in *Help* makes it easy to determine how the transformed algorithm should proceed when it performs an invalid operation: the operation being helped must have already terminated, so it no longer needs help. We demonstrate our approach by transforming a wasteful implementation of a double-compare-single-swap (DCSS) primitive [14].

We then present an extension to our weak descriptor ADT, and show how algorithms that perform *ReadField* operations *outside* of *Help* can be transformed to use this extension (in Section 4). We demonstrate our approach by transforming a wasteful implementation of a k-compare-and-swap (k-CAS) primitive [14]. In the full paper, we also transform the LLX and SCX primitives of Brown et al. [9], and provide proofs for all of our transformations. These primitives can be used to implement a wide variety of advanced lock-free data structures. For example, LLX and SCX have been used to implement lists, chromatic trees, relaxed AVL trees, relaxed (a, b)-trees, relaxed b-slack trees and weak AVL trees [10, 7, 15].

We use mostly known techniques to produce an efficient, provably correct implementation of our extended weak descriptor ADT. The high level idea is to (1) store a sequence number in each descriptor, (2) replace pointers to descriptors with *tagged sequence numbers*, which contain a process name and a sequence number, and (3) increment the sequence number in a descriptor each time it is reused. With this implementation, the transformed algorithms for k-CAS, and LLX and SCX, have some desirable properties. In the original k-CAS algorithm, *each operation attempt* allocates at least $k + 1$ new descriptors. In contrast, the transformed algorithm allocates only two descriptors *per process, once, at the beginning of the execution*, and these descriptors are reused. This entirely eliminates dynamic allocation *and* memory reclamation for descriptors, and results in an extremely small descriptor footprint.

We present extensive experiments on a 64-thread AMD system and a 48-thread Intel system (in Section 5). Our results show that transformed implementations always perform at least as well as their wasteful counterparts, and *significantly* outperform them in some workloads. In a k-CAS microbenchmark, our implementation outperformed wasteful implementations using fast distributed epoch-based reclamation [6], hazard pointers [23] and read-copy-update (RCU) [11] by up to 2.3x, 3.3x and 5.0x, respectively.

The crucial observation in this work is that, in algorithms where descriptors are used only to facilitate helping, a descriptor is no longer needed once its operation has terminated. This allows a process to reuse a descriptor as soon as its operation finishes, instead of allocating a new descriptor for each operation, and waiting considerably longer (and incurring much higher overhead) to reclaim it using standard memory reclamation techniques. The challenge in this work is to characterize the set of algorithms that can benefit from this observation, and to design and prove the correctness of a transformation that takes such algorithms and produces new algorithms that simply reuse a small number of descriptors.

2 Wasteful Algorithms

In this section, we describe two classes of lock-free wasteful algorithms, and give descriptor ADTs that capture their behaviour. First, we consider algorithms with *immutable* descriptors, which are not changed after they are initialized. We then discuss algorithms with *mutable* descriptors, which are modified by helpers.

For the sake of illustration, we start by describing one common way that lock-free wasteful algorithms are implemented. Consider a lock-free algorithm that implements a set of *high-level* operations. Each high-level operation consists of one or more *attempts*, which either

succeed, or fail due to contention. Each high-level operation attempt accesses a set of objects (e.g., individual memory locations or nodes of a tree). Conceptually, a high-level operation attempt locks a subset of these objects and then possibly modifies some of them. These locks are special: instead of providing exclusive access to a *process*, they provide exclusive access to a *high-level operation attempt*. Whenever a high-level operation attempt by a process p is unable to lock an object because it is already locked by another high-level operation attempt O, p first *helps* O to complete, before continuing its own attempt or starting a new one. By helping O complete, p effectively removes the locks that prevent it from making progress. Note that p is able to access objects locked for a different high-level operation attempt (which is not possible in traditional lock-based algorithms), but only for the purpose of helping the other high-level operation attempt complete.

We now discuss how helping is implemented. Each high-level operation or operation attempt allocates a new *descriptor* object, and fills it with information that describes any modifications it will perform. This information will be used by any processes that help the high-level operation attempt. For example, if the lock-free algorithm performs its modifications with a sequence of CAS steps, then the descriptor might contain the addresses, expected values and new values for the CAS steps.

A high-level operation attempt locks each object it would like to access by publishing pointers to its descriptor, typically using CAS. Each pointer may be published in a dedicated field for descriptor pointers, or in a memory location that is also used to store application values. For example, in the BST of Ellen et al., nodes have a separate field for descriptor pointers [12], but in Harris' implementation of multi-word CAS from single-word CAS, high-level operations temporarily replace application values with pointers to descriptors [14].

When a process encounters a pointer *ptr* to a descriptor (for a high-level operation attempt that is not its own), it may decide to help the other high-level operation attempt by invoking a function *Help*(*ptr*). Typically, *Help*(*ptr*) is also invoked by the process that started the high-level operation. That is, the mechanism used to help is the same one used by a process to perform its own high-level operation attempt.

Wasteful algorithms typically assume that, whenever an operation attempt allocates a new descriptor, it uses fresh memory that has never previously been allocated. If this assumption is violated, then an *ABA problem* may occur. Suppose a process p reads an address x and sees A, then performs a CAS to change x from A to C, and interprets the success of the CAS to mean that x contained A at all times between the read and CAS. If another process changes x from A to B and back to A between p's read and CAS, then p's interpretation is invalid, and an ABA problem has occurred. Note that safe memory reclamation algorithms will reclaim a descriptor only if no process has, or can obtain, a pointer to it. Thus, no process can tell whether a descriptor is allocated fresh or reclaimed memory. So, safe memory reclamation will not introduce ABA problems.

2.1 Immutable descriptors

We give a straightforward *immutable descriptor* ADT that captures the way that descriptors are used by the class of wasteful algorithms we just described. A *descriptor* has a set of fields, and each field contains a value. The ADT offers two operations: *CreateNew* and *ReadField*. *CreateNew* takes, as its arguments, a descriptor type and a sequence of values, one for each field of the descriptor. It returns a unique descriptor pointer *des* that has never previously been returned by *CreateNew*. Every descriptor pointer returned by *CreateNew* represents a new immutable descriptor object. *ReadField* takes, as its arguments, a descriptor pointer *des* and a field f, and returns the value of f in *des*. We require the immutable descriptor ADT operations to be lock-free, so they can be used to implement lock-free data structures.

```
1  DCSS(a1, e1, a2, e2, n2) :
2    des := CreateNew(DCSSdes, a1, e1, a2, e2, n2)
3    fdes := flag(des)
4    loop
5      r := CAS(a2, e2, fdes)
6      if r is flagged then DCSSHelp(r)
7      else exit loop
8    if r = e2 then DCSSHelp(fdes)
9    return r

11 DCSSRead(addr) :
12   loop
13     r := *addr
14     if r is flagged then DCSSHelp(r)
15     else exit loop
16   return r
```

```
17 type DCSSdes :
       {ADDR1, EXP1, ADDR2, EXP2, NEW2}

21 DCSSHelp(fdes) :
22   des := unflag(fdes)
23   a1 := ReadField(des, ADDR1)
24   a2 := ReadField(des, ADDR2)
25   e1 := ReadField(des, EXP1)
26   if *a1 = e1 then
27     n2 := ReadField(des, NEW2)
28     CAS(a2, fdes, n2)
29   else
30     e2 := ReadField(des, EXP2)
31     CAS(a2, fdes, e2)
```

Figure 1 Code for the *DCSS* algorithm of Harris et al. [14] using the *immutable descriptor* ADT.

Example Algorithm: DCSS. We use the double-compare single-swap (*DCSS*) algorithm of Harris et al. [14] as an example of a lock-free algorithm that fits the preceding description. Its usage of descriptors is easily captured by the immutable descriptor ADT. A $DCSS(a_1, e_1, a_2, e_2, n_2)$ operation does the following *atomically*. It checks whether the values in addresses a_1 and a_2 are equal to a pair of expected values, e_1 and e_2. If so, it stores the value n_2 in a_2 and returns e_2. Otherwise it returns the current value of a_2.

Pseudocode for the *DCSS* algorithm appears in Figure 1. At a high level, *DCSS* creates a descriptor, and then attempts to lock a_2 by using CAS to replace the value in a_2 with a pointer to its descriptor. Since the *DCSS* algorithm replaces values with descriptor pointers, it needs a way to distinguish between values and descriptor pointers (in order to determine when helping is needed). So, it steals a bit from each memory location and uses this bit to *flag* descriptor pointers.

We now give a more detailed description. *DCSS* starts by creating and initializing a new descriptor *des* at line 2. It then flags *des* at line 3. We call the result *fdes* a *flagged pointer*. *DCSS* then attempts to lock a_2 in the loop at lines 4-7. In each iteration, it tries to store its flagged pointer in a_2 using CAS. If the CAS is successful, then the operation attempt invokes *DCSSHelp* to complete the operation (at line 8). Now, suppose the CAS fails. Then, the *DCSS* checks whether its CAS failed because a_2 contained another *DCSS* operation's flagged pointer (at line 6). If so, it invokes *DCSSHelp* to help the other *DCSS* complete, and then retries its CAS. *DCSS* repeatedly performs its CAS (and helping) until the *DCSS* either succeeds, or fails because a_2 did not contain e_2.

DCSSHelp takes a flagged pointer *fdes* as its argument, and begins by unflagging *fdes* (to obtain the actual descriptor pointer for the operation). Then, it reads a_1 and checks whether it contains e_1 (at line 26). If so, it uses CAS to change a_2 from *fdes* to n_2, completing the *DCSS* (at line 28). Otherwise, it uses CAS to change a_2 from *fdes* to e_2, effectively aborting the *DCSS* (at line 31). Note that this code is executed by the process that created the descriptor, and also possibly by several helpers. Some of these helpers may perform a CAS at line 26 and some may perform a CAS at line 28, but only the first of these CAS steps can succeed.

When a program uses DCSS, some addresses can contain either values or descriptor pointers. So, each read of such an address must be replaced with an invocation of a function called *DCSSRead*. *DCSSRead* takes an address *addr* as its argument, and begins by reading

addr (at line 13). It then checks whether it read a descriptor pointer (at line 14) and, if so, invokes *DCSSHelp* to help that *DCSS* complete. *DCSSRead* repeatedly reads and performs helping until it sees a value, which it returns (at line 16).

2.2 Mutable descriptors

In some more advanced lock-free algorithms, each descriptor also contains information about the *status* of its high-level operation attempt, and this status information is used to coordinate helping efforts between processes. Intuitively, the status information gives helpers some idea of what work has already been done, and what work remains to be done. Helpers use this information to direct their efforts, and update it as they make progress. For example, the state information might simply be a bit that is set (by the process that started the high-level operation, or a helper) once the high-level operation succeeds.

As another example, in an algorithm where high-level operation attempts proceed in several phases, the descriptor might store the current phase, which would be updated by helpers as they successfully complete phases. Observe that, since lock-free algorithms cannot use mutual exclusion, helpers often use CAS to avoid making conflicting changes to status information, which is quite expensive. Updating status information may introduce contention. Even when there is no contention, it adds overhead. Lock-free algorithms typically try to minimize updates to status information. Moreover, status information is usually simplistic, and is encoded using a small number of bits.

Status information might be represented as a single field in a descriptor, or it might be distributed across several fields. Any fields of a descriptor that contain status information are said to be *mutable*. All other fields are called *immutable*, because they do not change during an operation.

Mutable descriptor ADT. We now extend the immutable descriptor ADT to provide operations for changing (mutable) fields of descriptors. The *mutable descriptor* ADT offers four operations: *CreateNew*, *WriteField*, *CASField* and *ReadField*. The semantics for *CreateNew* and *ReadField* are the same as in the immutable descriptor ADT. *WriteField* takes, as its arguments, a descriptor pointer *des*, a field f and a value v. It stores v in field f of *des*. *CASField* takes, as its arguments, a descriptor pointer *des*, a field f, an expected value *exp* and a new value v. Let v_f be the value of f in *des* just before the *CASField*. If $v_f = exp$, then *CASField* stores v in f. *CASField* returns v_f. As in the immutable descriptor ADT, we require the operations of the mutable descriptor ADT to be lock-free.

Example Algorithm: k-CAS. A k-*CAS*$(a_1, ..., a_k, e_1, ..., e_k, n_1, ..., n_k)$ operation atomically does the following. First, it checks if each address a_i contains its expected value e_i. If so, it writes a new value n_i to a_i for all i and returns true. Otherwise it returns false.

The k-CAS algorithm of Harris et al. [14] is an example of a lock-free algorithm that has descriptors with mutable fields. At a high level, a k-CAS operation O starts by creating a descriptor that contains its arguments. It then tries to lock each location a_i *for the operation* O by changing the contents of a_i from e_i to *des*, where *des* is a pointer to O's descriptor. If it successfully locks each location a_i, then it changes each a_i from *des* to n_i, and returns true. If it fails because a_i is locked for another operation, then it helps the other operation to complete (and unlock its addresses), and then tries again. If it fails because a_i contains an application value different from e_i, then the k-CAS fails, and unlocks each location a_j that it locked by changing it from *des* back to e_j, and returns false. (The same thing happens if O fails to lock a_i because the operation has already terminated.)

In addition to the arguments to its k-CAS operation, a k-CAS descriptor contains a 2-bit *state* field that initially contains *Undecided* and is changed to *Succeeded* or *Failed* depending on how the operation progresses. This *state* field is used to coordinate helpers.

Let p be a process performing (or helping) a k-CAS operation O that created a descriptor d. If p fails to lock some address a_i in d, then p attempts to change the *state* of d using CAS from *Undecided* to *Failed*. On the other hand, if p successfully locks each address in d, then p attempts to change the *state* of d using CAS from *Undecided* to *Succeeded*. Since the *state* field changes only from *Undecided* to either *Failed* or *Succeeded*, only the first CAS on the state field of d will succeed. The k-CAS implementation then uses a lock-free DCSS primitive (the one presented in Section 2.1) to ensure that p can lock addresses for O *only* while d's *state* is *Undecided*. This prevents helpers from erroneously performing successful CAS steps after the k-CAS operation is already over.

Recall that the DCSS algorithm allocates a descriptor for each DCSS operation. A k-CAS operation performs potentially *many* DCSS operations (at least k for a successful k-CAS), and also allocates its own k-CAS descriptor. The k-CAS algorithm need not be aware of DCSS descriptors (or of the bit reserved in each memory location by the DCSS algorithm to flag values as DCSS descriptor pointers), since it can simply use the *DCSSRead* procedure described above whenever it accesses a memory location that might contain a DCSS descriptor. However, the k-CAS algorithm performs DCSS on the *state* field of a k-CAS descriptor, which is accessed using the k-CAS descriptor's *ReadField* operation. To allow DCSS to access the *state* field, we must modify DCSS slightly. First, instead of passing an address a_1 to DCSS, we pass a pointer to the k-CAS descriptor and the name of the *state* field. Second, we replace the read of $addr_1$ in DCSS with an invocation of *ReadField*.

Since k-CAS descriptor pointers are temporarily stored in memory locations that normally contain application values, the k-CAS algorithm needs a way to determine whether a value in a memory location is an application value or a k-CAS descriptor pointer. In the DCSS algorithm, the solution was to reserve a bit in each memory location, and use this bit to *flag* the value contained in the location as a pointer to a DCSS descriptor. Similarly, the k-CAS algorithm reserves a bit in each memory location to flag a value as a k-CAS descriptor pointer. The k-CAS and DCSS algorithms need not be aware of each other's reserved bits, but they should not reserve the same bit (or else, for example, a DCSS operation could encounter a k-CAS descriptor pointer, and interpret it as a DCSS descriptor pointer).

When the k-CAS algorithm is used, some memory addresses may contain either values or descriptor pointers, so reads of such addresses must be replaced by a *k-CASRead* operation. This operation reads an address, and checks whether it contains a k-CAS descriptor pointer. If so, it helps the k-CAS operation to complete, and tries again. Otherwise, it returns the value it read. For further details on the k-CAS algorithm, refer to [14].

3 Weak descriptors

In this section we present a *weak descriptor* ADT that has weaker semantics than the mutable descriptor ADT, but can be implemented more efficiently (without requiring any memory reclamation for descriptors). We identify a class of algorithms that use the mutable descriptor ADT, and which can be transformed to use the weak descriptor ADT, instead.

We first discuss a restricted case where operation attempts only create a single descriptor, and we give an ADT and transformation for that restricted case. (In the next section, we describe how the ADT and transformation can be modified slightly to support operation attempts that create multiple descriptors.)

The weak descriptor ADT is a variant of the mutable descriptor ADT that allows some operations to *fail*. To facilitate the discussion, we introduce the concept of descriptor validity. Let *des* be a pointer returned by a *CreateNew* operation O by a process p, and d be the descriptor pointed to by *des*. In each configuration, d is either **valid** or **invalid**. Initially, d is valid. If p performs another *CreateNew* operation O' *after* O, then d becomes invalid immediately after O' (and will never be valid again).

We say that a *ReadField*$(des, ...)$, *WriteField*$(des, ...)$ or *CASField*$(des, ...)$ operation is performed **on a descriptor** d, where *des* is a pointer to d. An operation on a valid (resp., invalid) descriptor is said to be valid (resp., invalid). Invalid operations have no effect on any base object, and return a special value $\perp$ (which is never contained in a field of any descriptor) instead of their usual return value. We say that a *CreateNew* operation O is performed **on a descriptor** d if O returns a pointer to d. Observe that a *CreateNew* operation is always valid. We say that a process p **owns** a descriptor d if it performed a *CreateNew* operation that returned a pointer *des* to d.

The semantics for *CreateNew* are the same as in the mutable descriptor ADT. The semantics for the other three operations are the same as in the mutable descriptor ADT, except that they can be invalid. As in the previous ADTs, these operations must be lock-free.

3.1 Transforming a class of algorithms to use the weak descriptor ADT

We now formally define a class of lock-free algorithms that use the mutable descriptor ADT, and can easily be transformed so that they use the weak descriptor ADT, instead. We say that a step s of an execution is *nontrivial* if it changes the state of an object o in shared memory, and *trivial* otherwise. In particular, all invalid operations are trivial, and an unsuccessful CAS or a CAS whose expected and new values are the same are both trivial. In the following, we abuse notation slightly by referring interchangeably to a descriptor and a pointer to it.

▶ **Definition 1.** Weak-compatible algorithms (WCA) are lock-free wasteful algorithms that use the mutable descriptor ADT, and have the following properties:

1. Each high-level operation attempt O by a process p may create (and initialize) a single descriptor d. Inside O, p may perform at most one invocation of a function *Help*(d) (and p may not invoke *Help*(d) outside of O).
2. A process may help any operation attempt O' by another process by invoking *Help*(d') where d' is the descriptor that was created by O'.
3. If O terminates at time t, then any steps taken in an invocation of *Help*(d) after time t are *trivial* (i.e., do not **change** the state of **any** shared object, incl. d).
4. While a process $q \neq p$ is performing *Help*(d), q cannot change any variables in its private memory that are still defined once *Help*(d) terminates (i.e., variables that are local to the process q, but are not local to *Help*).
5. All accesses (read, write or CAS) to a field of d occur inside either *Help*(d) or O.

At a high level, properties 1 and 2 of WCA describe how descriptors are created and helped. Property 4 intuitively states that, whenever a process q finishes helping another process perform its operation attempt, q knows only that it finished helping, and does not remember anything about what it did while helping the other process. In particular, this means that q cannot pay attention to the return value of *Help*. We explain why this behaviour makes sense. If q creates a descriptor d as part of a high-level operation attempt O and invokes *Help*(d), then q might care about the return value of *Help*, since it needs to compute the response of O. However, if q is just helping another process p's high-level

operation attempt O, then it does not care about the response of *Help*, since it does not need to compute the response of O. The remaining properties, 3 and 5, allow us to argue that the contents of a descriptor are no longer needed once the operation that created it has terminated (and, hence, it makes sense for the descriptor to become invalid). In Section 4, we will study a larger class of algorithms with a weaker version of property 5.

The transformation. Each algorithm in WCA can be transformed in a straightforward way into an algorithm that uses the weak descriptor ADT as follows. Consider any *ReadField* or *CASField* operation *op* performed by a high-level operation attempt O in an invocation of *Help*(d), where d was created by a *different* high-level operation attempt O'. Note that *op* is performed while O is *helping* O'. After *op*, a check is added to determine whether *op* was invalid, in which case p returns from *Help* immediately. (In this case, *Help* does not need to continue, since *op* will be invalid only if O' has already been completed by the process that owns d or a helper.)

Reading immutable fields efficiently. If an invocation of *Help*(des) accesses many immutable fields of a descriptor, then we can optimize it by replacing many *ReadField* operations with a single, more efficient operation. Details appear in the full paper.

4 Extended Weak Descriptors

In this section, we describe an extended version of the weak descriptor ADT, and an extended version of the transformation in Section 3.1. This extended transformation weakens property 5 of WCA so that *ReadField* operations on a descriptor d can also be performed *outside* of *Help*(d). At a high level, we handle *ReadField* operations performed outside of *Help* as follows. For *ReadField*s performed inside *Help*, we have seen that we can simply stop helping when $\perp$ is returned. However, for *ReadField*s performed outside of *Help*, it is not clear, in general, how we should respond if $\perp$ is returned. Intuitively, the goal is to find a value that *ReadField* can return so that the algorithm will behave the same way as it would if the descriptor were still valid. In some algorithms, just knowing that an operation has been completed gives us enough information to determine what a *ReadField* operation should return (as we will see below).

Extended weak descriptor ADT. This ADT is the same as the weak descriptor ADT, except that *ReadField* is extended to take, as an additional argument, a default value dv that is returned instead of $\perp$ when the operation is invalid. Observe that the weak descriptor ADT is a special case of the extended weak descriptor ADT where each argument dv to an invocation of *ReadField* is $\perp$.

The extended transformation. *CASField* and *WriteField* operations are handled the same way as in the WCA transformation. However, an invocation of *ReadField*(des, f) is handled differently depending on whether it occurs inside an invocation of *Help*(des). If it does, it is replaced with an invocation of *ReadField*($des, f, \perp$) followed by the check, as in the WCA transformation. If not, it is replaced with an invocation of *ReadField*(des, f, dv), where the choice of dv is specific to the algorithm being transformed.

Let $\mathcal{A}$ be any algorithm that uses mutable descriptors, and satisfies properties 1-4 of WCA algorithms (see Definition 1), as well as a weaker version of property 5, called property 5′, which states: every write or CAS to a field of a descriptor d must occur in an invocation

of *Help*(d). Let e be an execution of $\mathcal{A}$ and let e' be an execution that is the same as e, except that one (arbitrary) descriptor d becomes invalid at some point t after the high-level operation attempt O that created d terminates. (When we say that d becomes invalid at time t, we mean that after t, each invocation of *ReadField*(d, f, dv) that is performed outside of *Help*(d) returns its default value dv.)

Let O' be any high-level operation attempt in e' which, after t, performs *ReadField* on d outside of *Help*(d). We say that an extended transformation is *correct for* $\mathcal{A}$ if, for all choices of e, e', d, t, and O', the exact same changes are performed by O' in e and e' to any variables that are still defined once O' terminates (i.e., variables that are local to the process performing O', but are not local to O', and variables in shared memory), and O' returns the same response in both executions. An algorithm $\mathcal{A}$ is an *extended weak-compatible algorithm* (and is in the class *EWCA*) if there is an extended transformation that is correct for $\mathcal{A}$.

Multiple descriptors per operation attempt. In some lock-free algorithms, an operation can create several different types of descriptors, and invoke different *Help* procedures. For simplicity, we think of there being a single *Help* procedure that checks the type of the descriptor passed to it, and behaves differently for different types. To support such algorithms, we make the following minor changes. We redefine *CreateNew* so it only invalidates previous descriptors of the *same type*. We also update Property 1 as follows: Each high-level operation attempt O by a process p may create a *sequence* D of descriptors, each with a unique type. Inside O, p may perform at most one invocation of a function *Help*(d) for each $d \in D$ (and may not invoke *Help*(d) outside of O). Details appear in the full paper.

Example Algorithm: k-CAS. In this section, we explain how the extended transformation is applied to the k-CAS algorithm presented in Section 2.2. Note that no invocations of *ReadField* on a DCSS descriptor *des* are performed outside of *HelpDCSS*(*des*). There is only one place in the algorithm where an invocation I of *ReadField* on a k-CAS descriptor *des* is performed *outside* of *Help*(*des*) (the *Help* procedure for k-CAS). Specifically, I reads the *state* field of a k-CAS descriptor inside the modified version of *HelpDCSS*. Recall that the k-CAS algorithm passes a k-CAS descriptor pointer and the name of the *state* field as the first argument to DCSS, and the DCSS algorithm is modified to use *ReadField* (at line 26 of Figure 1) to read this *state* field. We choose the default value $dv = Succeeded$ for this invocation of *ReadField*. We explain why this extended transformation is correct.

When I is performed at line 26 of *DCSSHelp* (in Figure 1), its response is compared with e_1, which contains *Undecided*. If I returns *Undecided*, then the CAS at line 28 is performed, and the process p performing I returns from *HelpDCSS*. Otherwise, the CAS at line 31 is performed, and p returns from *HelpDCSS*.

Suppose I is invalid. Then, we know the k-CAS operation attempt that created *des* has been completed. We use the following algorithm specific knowledge. After a k-CAS operation attempt has completed, its k-CAS descriptor has *state Succeeded* or *Failed* (and is never changed back to *Undecided*). (This can be determined by inspection of the code.) Thus, if I were valid, its response would *not* be *Undecided*, and p would perform the CAS at line 31 and return from *HelpDCSS*. Since $dv = Succeeded$, p does exactly the same thing when I is invalid. (Note that the exact value of *state* is unimportant. It is only important that it is not *Undecided*.)

Example Algorithm: LLX and SCX. In the full paper, we also transform a wasteful implementation of the LLX and SCX primitives of Brown et al. [9].

Implementing the extended weak descriptor ADT. We give a brief high-level overview, here. Details appear in the full paper. It uses largely known techniques (similar to [21]), and is not the main contribution of this work. Each process p uses a *single* descriptor object $D_{T,p}$ in shared memory to represent *all* descriptors of type T that it ever creates. The descriptor object $D_{T,p}$ conceptually represents p's *current* descriptor of type T. At different times in an execution, $D_{T,p}$ represents different *abstract descriptors* created by p. We store a sequence number in $D_{T,p}$ that is incremented every time p performs $CreateNew(T, -)$. Instead of using traditional descriptor pointers, we represent each descriptor pointer as a pair of fields stored in a single word. These fields contain the name of the process who owns the descriptor, and a sequence number that indicates which invocation of *CreateNew* conceptually created this descriptor. When a descriptor pointer is passed to an operation O on the abstract descriptor, O compares the sequence number in *des* with the current sequence number in $D_{T,p}$ to determine whether the operation is valid or invalid. Thus, incrementing the sequence number in $D_{T,p}$ effectively makes all abstract descriptors of type T that were previously created by p *invalid*. Mutable fields are stored in a single word alongside a sequence number, so they can be updated with CAS, preventing invalid operations from making changes. (If the mutable fields and a sequence number cannot fit in one word, then one can use multiple words and attach the sequence number to each word.)

5 Experiments

Our experiments were run on two large-scale systems. The first is a 2-socket Intel E7-4830 v3 with 12 cores per socket and 2 hyperthreads (HTs) per core, for a total of 48 threads. Each core has a private 32KB L1 cache and 256KB L2 cache (which is shared between HTs on a core). All cores on a socket share a 30MB L3 cache. The second is a 4-socket AMD Opteron 6380 with 8 cores per socket and 2 HTs per core, for a total of 64 threads. Each core has a private 16KB L1 data cache and 2MB L2 cache (which is shared between HTs on a core). All cores on a socket share a 6MB L3 cache.

Since both machines have multiple sockets and a non-uniform memory architecture (NUMA), in all of our experiments, we pinned threads to cores so that the first socket is filled first, then the second socket is filled, and so on. Furthermore, within each socket, each core has one thread pinned to it before hyperthreading is engaged. Consequently, our graphs clearly show the effects of hyperthreading and NUMA.

Both machines have 128GB of RAM. Each runs Ubuntu 14.04 LTS. All code was compiled with the GNU C++ compiler (G++) 4.8.4 with build target x86_64-linux-gnu and compilation options `-std=c++0x -mcx16 -O3`. Thread support was provided by the POSIX Threads library. We used the Performance Application Programming Interface (PAPI) library to collect statistics from hardware counters to determine cache miss rates, stall times, etc. We used the scalable allocator jemalloc 4.2.1, which greatly improved performance.

***k*-CAS microbenchmark.** In order to compare our reusable descriptor technique with algorithms that reclaim descriptors, we implemented k-CAS with several memory reclamation schemes. Specifically, we implemented a lock-free memory reclamation scheme that aggressively frees memory called *hazard pointers* [23], a (blocking) epoch-based reclamation scheme called *DEBRA* [6], and reclamation using the read-copy-update (RCU) primitives [11] (also blocking). We use *Reuse* as shorthand for our reusable descriptor based algorithm, and *DEBRA*, *HP* and *RCU* to denote the other algorithms.

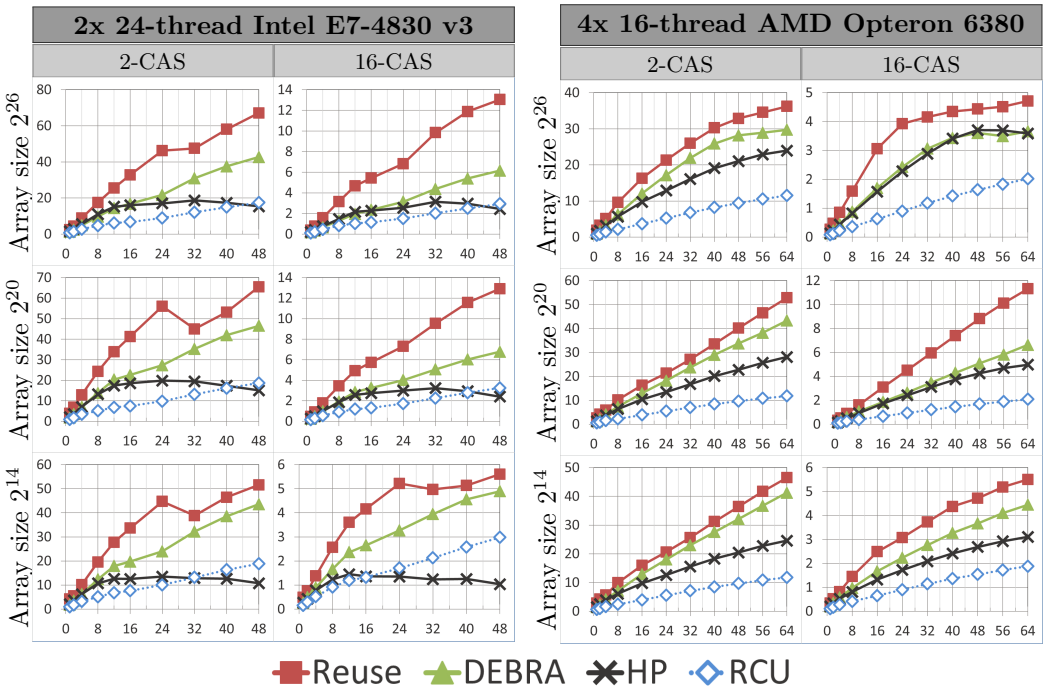


Figure 2 Results for a *k*-**CAS microbenchmark**. The x-axis represents the number of concurrent threads. The y-axis represents operations per microsecond.

The paper by Harris et al. also describes an optimization to reduce the number of DCSS descriptors that are allocated by embedding them in the k-CAS descriptor. We applied this optimization, and found that it did not significantly improve performance. Furthermore, it complicated reclamation with hazard pointers. Thus, we did not use this optimization.

Methodology. We compared our implementations of k-CAS using a simple array-based microbenchmark. For each algorithm $A \in \{Reuse, DEBRA, HP, RCU\}$, array size $S \in \{2^{14}, 2^{20}, 2^{26}\}$ and k-CAS parameter $k \in \{2, 16\}$, we run ten timed *trials* for several thread counts n. In each trial, an array of a fixed size S is allocated and each entry is initialized to zero. Then, n concurrent threads run for one second, during which each thread repeatedly chooses k uniformly random locations in the array, reads those locations, and then performs a k-CAS (using algorithm A) to increment each location by one.

As a way of validating correctness in each trial, each thread keeps track of how many successful k-CAS operations it performs. At the end of the trial, the sum of entries in the array must be k times the total number of successful k-CAS operations over all threads.

Results. The results for this benchmark appear in Figure 2. Error bars are not drawn on the graphs, since more than 97% of the data points have a standard deviation that is less than 5% of the mean (making them essentially too small to see).

Overall, *Reuse* outperforms every other algorithm, in every workload, on both machines. Notably, on the Intel machine, its throughput is *2.2 times* that of the next best algorithm at 48 threads with $k = 16$ and array size 2^{26}. On the AMD machine, its throughput is 1.7 times that of the next best algorithm at 64 threads with $k = 16$ and array size 2^{20}.

On the Intel machine, with $k = 2$, NUMA effects are quite noticeable for *Reuse* in the jump from 24 to 32 threads, as threads begin running on the second socket. According the statistics we collected with PAPI, this decrease in performance corresponds to an increase in cache misses. For example, with $k = 2$ and an array of size 2^{26} in the Intel machine, jumping from 24 threads to 25 increases the number of L3 cache misses per operation from 0.7 to 1.6 (with similar increases in L1 and L2 cache misses and pipeline stalls). We believe this is due to cross-socket cache invalidations.

From the three graphs for $k = 2$ on Intel, we can see that the effect is more severe with larger absolute throughput (since the additive overhead of a cache miss is more significant). Conversely, the effect is masked by the much smaller throughput of the slower algorithms, and by the substantially lower throughputs in the $k = 16$ case, except when the array is of size 2^{14}. In the array of size 2^{14}, contention is extremely high, since each of the 48 threads are accessing 16 k-CAS addresses, each of which causes contention on the entire cache line of 8 words, for a total of 6144 array entries contended at any given time. Thus, cache misses become a dominating factor in the performance on two sockets. These effects were not observed on the AMD machine. There, the number of cache misses is not significantly different when crossing socket boundaries, which suggests a robustness to NUMA effects that is not seen on the Intel machine.

Interestingly, absolute throughputs on the AMD machine are larger with array size 2^{20} than with sizes 2^{14} and 2^{26}. This is because the 2^{20} array size represents a sweet spot with less contention than the 2^{14} size and better cache utilization than the 2^{26} size. For example, with 64 threads and $k = 16$, *Reuse* incurred approximately 50% more cache misses with size 2^{26} than with size 2^{20}, and approximately 50% of operations helped one another with size 2^{14}, whereas less than 1% of operations helped one another with size 2^{20}.

Note, however, that this is not true on the Intel machine. There, 2^{26} is almost always as fast as 2^{20}, because of the very large shared L3 cache (which is 5x larger than on the AMD machine). This is reflected in the increased number of cycles where the processor is stalled (e.g., waiting for cache misses to be served) when moving from size 2^{20} to 2^{26}. On the Intel machine, stalled cycles increase by 85% per operation, whereas on the AMD machine they increase by a whopping 450% per operation.

Several additional experiments appear in the full paper, including empirical studies of memory usage, and of the performance of a transformed LLX and SCX implementation.

6 Related Work

Several papers have presented universal constructions or strong primitives for non-blocking algorithms in which operations create descriptors [17, 2, 1, 24, 14, 20, 18, 21, 3, 9]. A subset of these algorithms employ ad-hoc techniques for reusing descriptors [17, 2, 1, 24, 21, 20, 18]. The rest assume descriptors will be allocated for each operation and eventually reclaimed.

Most of the ad-hoc techniques for reusing descriptors have significant downsides. Some are complex and tightly integrated into the underlying algorithm, or rely on highly specific algorithmic properties (e.g., that descriptors contain only a single word). Others use synchronization primitives that atomically operate on large words, which are not available on modern systems, and are inefficient when implemented in software. Yet others introduce high space overhead (e.g., by attaching a sequence number to *every* memory word). Some techniques also incur significant runtime overhead (e.g., by invoking expensive synchronization primitives just to *read fields* of a descriptor). Furthermore, these techniques give, at best, a vague idea of how one might reuse descriptors for arbitrary algorithms, and it would be difficult to determine how to use them in practice. Our work avoids all of these downsides, and provides a concrete approach for transforming a large class of algorithms.

Barnes [4] introduced a technique for producing non-blocking algorithms that can be more efficient (and sometimes simpler) than the universal constructions described above. With Barnes' technique, each operation creates a new descriptor. Creating a new descriptor for each operation allows his technique to avoid the ABA problem while remaining conceptually simple. Each operation conceptually locks each location it will modify by installing a pointer to its descriptor, and then performs it modifications and unlocks each location. Barnes' technique is the inspiration for the class WCA. Many algorithms have since been introduced using variants of this technique [14, 12, 3, 16, 26, 9, 10]. Several of these algorithms are quite efficient in practice despite the overhead of creating and reclaiming descriptors. Our technique can significantly improve the space and time overhead of such algorithms.

Recent work has identified ways to use hardware transactional memory (HTM) to reduce descriptor allocation [8, 19]. Currently, HTM is supported only on recent Intel and IBM processors. Other architectures, such as AMD, SPARC and ARM have not yet developed HTM support. Thus, it is important to provide solutions for systems with no HTM support. Additionally, even with HTM support, our approach is useful. Current (and likely future) implementations of HTM offer no progress guarantees, so one must provide a lock-free fallback path to guarantee lock-free progress. The techniques in [8, 19] accelerate the HTM-based code path(s), but do nothing to reduce descriptor allocations on the fallback path. In some workloads, many operations run on the fallback path, so it is important for it to be efficient. Our work provides a way to accelerate the fallback path, and is orthogonal to work that optimizes the fast path.

The *long-lived renaming* (LLR) problem is related to our work (see [5] for a survey), but its solutions do not solve our problem. LLR provides processes with operations to *acquire* one unique resource from a pool of resources, and subsequently *release* it. One could imagine a scheme in which processes use LLR to reuse a small set of descriptors by invoking *acquire* instead of allocating a new descriptor, and eventually invoking *release*. Note, however, that a descriptor can safely be released only once it can no longer be accessed by any other process. Determining when it is safe to release a descriptor is as hard as performing general memory reclamation, and would also require delaying the release (and subsequent acquisition) of a descriptor (which would increase the number of descriptors needed). In contrast, our weak descriptors eliminate the need for memory reclamation, and allow immediate reuse.

7 Conclusion

We presented a novel technique for transforming algorithms that throw away descriptors into algorithms that reuse descriptors. Our experiments show that our transformation yields significant performance improvements for a lock-free k-CAS algorithm. Furthermore, our transformation reduces peak memory usage by nearly three orders of magnitude over the next best implementation. We believe our transformation can be used to improve the performance and memory usage of many other algorithms that throw away descriptors. Moreover, we hope that our extended weak descriptor ADT will aid in the design of more efficient, complex algorithms, by allowing algorithm designers to benefit from the conceptual simplicity of throwing away descriptors without paying the practical costs of doing so.

Acknowledgments. We thank Faith Ellen for her gracious help in proving correctness for our transformations, and her insightful comments. Some of this work was done while Trevor was a student at the University of Toronto, and while Maya was visiting him there.

References

1 Yehuda Afek, Dalia Dauber, and Dan Touitou. Wait-free made fast. In *Proceedings of STOC'95*, pages 538–547, 1995.

2 James H. Anderson and Mark Moir. Universal constructions for multi-object operations. In *Proceedings of PODC'95*, pages 184–193, 1995.

3 Hagit Attiya and Eshcar Hillel. Highly concurrent multi-word synchronization. *Theoretical Computer Science*, 412(12):1243–1262, 2011.

4 Greg Barnes. A method for implementing lock-free shared-data structures. In *Proceedings of SPAA'93*, pages 261–270, 1993.

5 Alex Brodsky, Faith Ellen, and Philipp Woelfel. Fully-adaptive algorithms for long-lived renaming. *Distributed Computing*, 24(2):119, 2011.

6 Trevor Brown. Reclaiming memory for lock-free data structures. In *Proceedings of PODC'15*, pages 261–270, 2015.

7 Trevor Brown. *Techniques for Constructing Efficient Data Structures*. PhD thesis, University of Toronto, 2017.

8 Trevor Brown. A template for implementing fast lock-free trees using HTM. In *Proceedings of PODC'17*, pages 293–302, 2017.

9 Trevor Brown, Faith Ellen, and Eric Ruppert. Pragmatic primitives for non-blocking data structures. In *Proceedings of PODC'13*, pages 13–22, 2013.

10 Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In *Proceedings of PPoPP'14*, pages 329–342, 2014.

11 Mathieu Desnoyers, Paul E. McKenney, Alan S. Stern, Michel R. Dagenais, and Jonathan Walpole. User-level implementations of Read-Copy Update. *IEEE Transactions on Parallel and Distributed Systems*, 23(2):375–382, 2012.

12 Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceedings of PODC'10*, pages 131–140, 2010.

13 Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of DISC'01*, pages 300–314, 2001.

14 Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of DISC'02*, pages 265–279, 2002.

15 Meng He and Mengdu Li. Deletion without rebalancing in non-blocking binary search trees. In *Proceedings of OPODIS'16*, pages 34:1–34:17, 2017.

16 Shane V. Howley and Jeremy Jones. A non-blocking internal binary search tree. In *Proceedings of SPAA '12*, pages 161–171, 2012.

17 Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of PODC'94*, pages 151–160, 1994.

18 Prasad Jayanti and Srdjan Petrovic. Efficiently implementing a large number of LL/SC objects. In *Proceedings of OPODIS'05*, pages 17–31, 2005.

19 Yujie Liu, Tingzhe Zhou, and Michael Spear. Transactional acceleration of concurrent data structures. In *Proceedings of SPAA'15*, pages 244–253, 2015.

20 Victor Luchangco, Mark Moir, and Nir Shavit. Nonblocking k-compare-single-swap. *Theory of Computing Systems*, 44(1):39–66, January 2009.

21 Virendra Jayant Marathe and Mark Moir. Toward high performance nonblocking software transactional memory. In *Proceedings of PPoPP'08*, pages 227–236, 2008.

22 Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of SPAA'02*, pages 73–82, 2002.

23 Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, June 2004.

24 Mark Moir. Practical implementations of non-blocking synchronization primitives. In *Proceedings of PODC '97*, pages 219–228, 1997.

25 Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *Proceedings of PPoPP '14*, pages 317–328, 2014.

26 Niloufar Shafiei. Non-blocking patricia tries with replace operations. In *Proceedings of ICDCS'13*, pages 216–225, 2013.

27 John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of PODC'95*, pages 214–222, 1995.

Demand-Aware Network Designs of Bounded Degree*

Chen Avin[1], Kaushik Mondal[2], and Stefan Schmid[3]

1 **Communication Systems Engineering Department, Ben Gurion University of the Negev, Be'er Scheva, Israel**
avin@cse.bgu.ac.il

2 **Communication Systems Engineering Department, Ben Gurion University of the Negev, Be'er Scheva, Israel**
mondal@post.bgu.ac.il

3 **Department of Computer Science, Aalborg University, Denmark**
schmiste@cs.aau.dk

Abstract

Traditionally, networks such as datacenter interconnects are designed to optimize worst-case performance under *arbitrary* traffic patterns. Such network designs can however be far from optimal when considering the *actual* workloads and traffic patterns which they serve. This insight led to the development of demand-aware datacenter interconnects which can be reconfigured depending on the workload.

Motivated by these trends, this paper initiates the algorithmic study of demand-aware networks (DANs), and in particular the design of bounded-degree networks. The inputs to the network design problem are a discrete communication request distribution, $\mathcal{D}$, defined over communicating pairs from the node set V, and a bound, Δ, on the maximum degree. In turn, our objective is to design an (undirected) demand-aware network $N = (V, E)$ of bounded-degree Δ, which provides short routing paths between frequently communicating nodes distributed across N. In particular, the designed network should minimize the *expected path length* on N (with respect to $\mathcal{D}$), which is a basic measure of the efficiency of the network.

We show that this fundamental network design problem exhibits interesting connections to several classic combinatorial problems and to information theory. We derive a general lower bound based on the entropy of the communication pattern $\mathcal{D}$, and present asymptotically optimal network-aware design algorithms for important distribution families, such as sparse distributions and distributions of locally bounded doubling dimensions.

1998 ACM Subject Classification F.2 Analysis of Algorithms and Problem Complexity, C.2.1 Network Architecture and Design

Keywords and phrases Network design, reconfigurable networks, datacenter topology, peer-to-peer computing, entropy, sparse spanners

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.5

1 Introduction

The problem studied in this paper is motivated by the advent of more flexible datacenter interconnects, such as ProjecToR [16, 17]. These interconnects aim to overcome a fundamental

* This work was supported by the German-Israeli Foundation for Scientific Research (GIF) Grant I-1245-407.6/2014

© Chen Avin, Kaushik Mondal, and Stefan Schmid;
licensed under Creative Commons License CC-BY

drawback of traditional datacenter network designs: the fact that network designers must decide *in advance* on how much capacity to provision between electrical packet switches, e.g., between Top-of-Rack (ToR) switches in datacenters. This leads to an undesirable tradeoff [25]: either capacity is over-provisioned and therefore the interconnect expensive (e.g., a fat-tree provides full-bisection bandwidth), or one may risk congestion, resulting in a poor cloud application performance. Accordingly, systems such as ProjecToR provide a reconfigurable interconnect, allowing to establish links flexibly and in a *demand-aware manner*. For example, direct links or at least short communication paths can be established between frequently communicating ToR switches. Such links can be implemented using a bounded number of lasers, mirrors, and photodetectors per node [17]. First experiments with this technology demonstrated promising results: although the interconnecting networks is of bounded degree, short routing paths can be provided between communicating nodes.

The problem of designing demand-aware networks is a fundamental one, and finds interesting applications in many distributed and networked systems. For example, while many peer-to-peer overlay networks today are designed towards optimizing the *worst-case performance* (e.g., minimal diameter and/or degree), it is an intriguing question whether the "hard instances" actually show up in real life, and whether better topologies can be designed if we are given more information about the actual communication patterns these networks serve in practice.

While the problem is natural, surprisingly little is known today about the design of demand-aware networks. At the same time, as we will show in this paper, the design of demand-aware networks is related to several classic combinatorial problems.

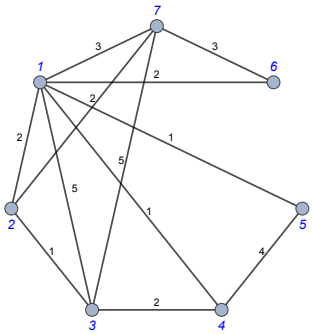
Our vision is reminiscent in spirit to the question posed by Sleator and Tarjan over 30 years ago in the context of binary search trees [10, 26]: While there is an inherent lower bound of $\Omega(\log n)$ for accessing an arbitrary element in a binary search tree, can we do better on some "easier" instances? The authors identified the *entropy* to be a natural metric to measure the performance under actual demand patterns. We will provide evidence in this paper that the entropy, in a slightly different flavor, also plays a crucial role in the context of network designs, establishing an interesting connection.

The Problem: Bounded Network Design. We consider the following network design problem, henceforth referred to as the *Bounded Network Design* problem, short *BND*. We consider a set of n nodes (e.g., top-of-rack switches, servers, peers) $V = \{1, \ldots, n\}$ interacting according to a certain *communication pattern*. The pattern is modelled by $\mathcal{D}$, a discrete distribution over *communication requests* defined over $V \times V$. We represent this distribution using a communication matrix $M_{\mathcal{D}}[\mathrm{p}(i,j)]_{n \times n}$ where the (i,j) entry indicates the communication frequency, $p(i,j)$, from the (communication) source i to the (communication) destination j. The matrix is normalized, i.e., $\sum_{ij} \mathrm{p}(i,j) = 1$. Moreover, we can interpret the distribution $\mathcal{D}$ as a weighted directed *demand graph* $G_{\mathcal{D}}$, defined over the same set of nodes V: A directed edge $(u,v) \in E(G_{\mathcal{D}})$ exists iff $\mathrm{p}(u,v) > 0$. We set the edge weight to the communication probability: $w(i,j) = \mathrm{p}(i,j)$.

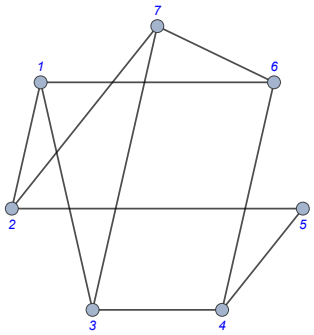
In turn, our objective is to design an unweighted, undirected *Demand-Aware Network* (`DAN`) defined over the set of nodes V and the distribution $\mathcal{D}$, henceforth denoted as $N(\mathcal{D})$ or just N when $\mathcal{D}$ is clear from the context. The objective is that $N(\mathcal{D})$ optimally serves the communication requests from $\mathcal{D}$ under the constraint that N must be chosen from a certain family of *desired topologies* $\mathcal{N}$. In particular, we are interested in *sparse* networks (i.e., having a *linear number* of edges) with *bounded* degree Δ (i.e., nodes have a small number of lasers [17]), and we denote the family of Δ-bounded degree graphs by $\mathcal{N}_\Delta$.

	1	2	3	4	5	6	7
1	0	$\frac{2}{65}$	$\frac{1}{13}$	$\frac{1}{65}$	$\frac{1}{65}$	$\frac{2}{65}$	$\frac{3}{65}$
2	$\frac{2}{65}$	0	$\frac{1}{65}$	0	0	0	$\frac{2}{65}$
3	$\frac{1}{13}$	$\frac{1}{65}$	0	$\frac{2}{65}$	0	0	$\frac{1}{13}$
4	$\frac{1}{65}$	0	$\frac{2}{65}$	0	$\frac{4}{65}$	0	0
5	$\frac{1}{65}$	0	$\frac{3}{65}$	$\frac{4}{65}$	0	0	0
6	$\frac{2}{65}$	0	0	0	0	0	$\frac{3}{65}$
7	$\frac{3}{65}$	$\frac{2}{65}$	$\frac{1}{13}$	0	0	$\frac{3}{65}$	0

(a)



(b)



(c)

Figure 1 Example of the *bounded network design* problem. (a) A given demand distribution $\mathcal{D}$ (which in this case is *symmetric*). (b) The demand graph $G_{\mathcal{D}}$ (with non-normalized weights). Nodes 1, 3, and 7 have a degree more than 3. (c) An optimal solution DAN N with $\Delta = 3$. In this case, the solution is not a subgraph but contains auxiliary edges (e.g., $\{2, 5\}$), and $\mathrm{EPL}(\mathcal{D}, N) = 1.19$ while $H(X \mid Y) = 1.08$ (the Shannon entropy to the base 3 is $H(X) = 1.68$).

Note that the designed network can be seen as "hosting" the served communication pattern, i.e., the demand graph is embedded on the designed network. Accordingly, we will sometimes refer to the demand graph as the *guest network* and to the designed network as the *host network*.

Our objective is to minimize the *expected path length* [1, 2, 24] of the designed host network $N \in \mathcal{N}$: For $u, v \in V(N)$, let $\mathrm{d}_N(u, v)$ denote the shortest path between u and v in N. Given a distribution $\mathcal{D}$ over $V \times V$ and a graph N over V, the *Expected Path Length* (EPL) of route requests is defined as:

$$\mathrm{EPL}(\mathcal{D}, N) = \mathbb{E}_{\mathcal{D}}[\mathrm{d}_N(\cdot, \cdot)] = \sum_{(u,v)\in\mathcal{D}} \mathrm{p}(u, v) \cdot \mathrm{d}_N(u, v)$$

Since routing across the host network usually occurs along shortest paths, the expected path length captures the average hop-count of a route (e.g., latency incurred or energy consumed along the way).

Succinctly, the Bounded Network Design (BND) problem is to minimize the expected path length and is defined as follows:

▶ **Definition 1** (Bounded Network Design)**.** Given a communication distribution, $\mathcal{D}$ and a maximum degree Δ, find a host graph $N \in \mathcal{N}_\Delta$ that minimizes the expected path length:

$$\mathrm{BND}(\mathcal{D}, \Delta) = \min_{N\in\mathcal{N}_\Delta} \mathrm{EPL}(\mathcal{D}, N)$$

See Figure 1 for an example of these definitions.

Our Contributions. This paper initiates the study of a fundamental problem: the design of demand-aware communication networks. While our work is motivated by recent trends in datacenter network designs, our model is natural and finds applications in many distributed and networked systems (e.g., peer-to-peer overlays). The main contribution of this paper is to establish an interesting connection of the network design problem to the conditional entropy of the communication matrix. In particular, we present a lower bound on the expected path length of a network with maximum degree Δ which is proportional to the conditional entropy

of $\mathcal{D}$, $H_\Delta(X \mid Y) + H_\Delta(Y \mid X)$ where Δ is the base of the logarithm used for calculating the entropy. While this lower bound can be as high as $\log n$, for many distributions it can be much lower (even constant). Our main results are presented in Theorem 7 which proves a matching upper bound for the case when $\mathcal{D}$ is a sparse distribution. It is important to note the real datacenters traffic shows evidence that the demand distributions are indeed sparse [23, 17]. Additionally Theorem 12 proves a matching upper bound for the case when $\mathcal{D}$ is a regular and uniform (but maybe dense) distribution of a locally bounded doubling dimension. Also in these two cases the conditional entropy could range from a constant and up to $\log n$. At the heart of our technical contribution is a novel technique to transform a low-distortion network of maximum degree Δ to a low-degree network whose maximum degree equals the average degree of the original network, while maintaining an expected path length in the order of the conditional entropy. Moreover, we show an interesting reduction of uniform and regular distributions to graph spanners in Theorem 8.

Paper Organization. The remainder of this paper is organized as follows. We put our work into perspective with respect to related work in Section 2 and introduce some preliminaries in Section 3. We derive lower bounds in Section 4 and present algorithms to design networks for sparse distributions resp. regular and uniform distributions in Section 5 resp. Section 6. We conclude our work and outline directions for future research in Section 7. Due to space constraints, some details are omitted in this paper, and we refer the reader to our accompanying technical report [3].

2 Putting Things Into Perspective and Related Work

There are at least three interesting perspectives on our problem. The first one arises when trying to gain some intuition about the problem complexity. If $\Delta = n$, the problem is simple: the demand (or guest) graph $G_\mathcal{D}$ itself can be used as the host graph or `DAN` $N \in \mathcal{N}_\Delta$, providing an ideal expected path length 1. If a sparse host graph is desired, a star topology could be used as a `DAN` to provide an expected path length of at most 2. At the other end of the spectrum, if $\Delta = 2$ (and the host network is required to be connected) the `DAN` N must be a line or a ring graph. However, the problem of how to arrange nodes on the linear chain or the ring such that the expected path length is minimized, is already NP-hard: the problem is essentially a Minimum Linear Arrangment (MinLA) problem [7, 11, 15]. One perspective to see our contribution is that in this paper, we are interested in what happens between these extremes, for other values of Δ, in particular for a constant Δ which guarantees that our host network will be sparse, i.e., has a linear number of edges. In contrast to the general arrangement problem which asks for an embedding of the guest graph on a *specific* and *given* host graph, in our network design problem we are free to *choose the best* host graph from a given family of graphs (i.e., bounded degree graphs). One might wonder: does this flexibility make the problem easier? Existing works on low maximum resp. low average degree networks, e.g., in the context of publish/subscribe overlays [8, 20, 21], do not provide formal performance guarantees.

Sparse and distance-preserving spanners open a second perspective on our work: intuitively, a good host graph N for $G_\mathcal{D}$ "looks similar" to $G_\mathcal{D}$. But in contrast to classic spanner problems in the literature which are primarily concerned with minimizing the worst-case *distortion* (resp. the average distortion) among *all* node pairs [4, 22], we are only interested in the *local distortion*. Namely, we aim to find a good "spanner" which preserves *locality of neighborhoods*, i.e., 1-hop neighborhoods in the demand graph. Second, unlike classic spanner

problems but similar to geometric (metric) spanners, the designed network N does not have to be a subgraph and may include edges which do not exist in the demand network $G_{\mathcal{D}}$, i.e., 0-entries in the corresponding communication matrix $M_{\mathcal{D}}$. We refer the corresponding edges as *auxiliary edges* (a.k.a. shortcut edges [19]). It is easy to see that auxiliary edges can indeed be required to compute optimal network designs, and yield strictly lower communication costs than subgraph spanners (e.g. Figure 1). Third, in contrast to the frequently studied sparse graph spanner problem variants, we require that nodes in the designed network are of degree at most Δ. Finally, we are not aware of any work studying the relationship between spanners and entropy. This makes our model fundamentally different from existing models studied in the literature.

The fact that our matrix represents a distribution provides some interesting structure. In particular, it leads us to a third connection, namely to information and coding theory. It is known that the expected path length in binary search trees [26] as well as in network designs providing local routing [2, 24] is upper bounded by the entropy $H(X)$ (over the (empirical) distribution of accessed elements X in the data structure). The conditional entropy of the distribution, $H(X|Y) + H(Y|X)$, is a lower bound on the expected path length of local routing tree designs [24] where X, Y are the random variables distributed according to the marginal distribution of the sources and destinations in $\mathcal{D}$. This bound is tight for the limited case where D is a product distribution (i.e., $p(i,j) = p(i)p(j)$). Additionally the optimal binary search tree can be computed efficiently for every $\mathcal{D}$ using dynamic programming [24]. In the current work we extend this line of research by studying more general distributions and a larger family of host networks.

3 Preliminaries

We start with some notation about $\mathcal{D}$. Let $\mathcal{D}[i,j]$ or $p(i,j)$ denote the probability that source i routes to destination j. Let $p(i)$ denote the probability that i is a source, i.e., $p(i) = \sum_j p(i,j)$. Similarly let $q(j)$ denote the probability that j is a destination. Let X, Y be random variables describing the marginal distribution of the sources and destinations in $\mathcal{D}$, respectively. Let $\overrightarrow{\mathcal{D}}[i]$ denote the *normalized* i'th row of $\mathcal{D}$, that is, the probability distribution of destinations given that the source is i. Similarly let $\overleftarrow{\mathcal{D}}[j]$ denote the normalized j'th column of $\mathcal{D}$, that is the probability distribution of sources given that the destination is j. Let Y_i and X_j be random variables that are distributed according to $\overrightarrow{\mathcal{D}}[i]$ and $\overleftarrow{\mathcal{D}}[j]$, respectively. We say that $\mathcal{D}$ is *regular* if $G_{\mathcal{D}}$ is a regular graph (both in terms of in and out degrees). We say that $\mathcal{D}$ is *uniform* if for every $\mathcal{D}[i,j] > 0$, $\mathcal{D}[i,j] = \frac{1}{m}$ and m is the number of edges in $G_{\mathcal{D}}$. We say that $\mathcal{D}$ is *symmetric* if $\mathcal{D}[i,j] = \mathcal{D}[j,i]$.

We will show that a natural measure to assess the quality of a designed network relates to the *entropy* of the communication pattern. For a discrete random variable X with possible values $\{x_1, \ldots, x_n\}$, the entropy $H(X)$ of X is defined as

$$H(X) = \sum_{i=1}^{n} p(x_i) \log_2 \frac{1}{p(x_i)} \tag{1}$$

where $p(x_i)$ is the probability that X takes the value x_i. Note that, $0 \cdot \log_2 \frac{1}{0}$ is considered as 0. If $\bar{p}$ is a discrete distribution vector (i..e, $p_i \geq 0$ and $\sum_i p_i = 1$) then we may write $H(\bar{p})$ or $H(p_1, p_2, \ldots p_n)$ to denote the entropy of a random variable that is distributed according to $\bar{p}$. If $\bar{p}$ is the uniform distribution with support s (s being the number of places in the distribution with $p_i > 0$, i.e., $p_i = 1/s$) then $H(\bar{p}) = \log s$.

Using the decomposition (a.k.a. grouping) properties of entropy the following are well-known [9]:

$$H(p_1, p_2, p_3 \dots p_m) \geq H(p_1 + p_2, p_3 \dots p_m) \tag{2}$$

$$H(p_1, p_2, p_3 \dots p_m) \geq (1 - p_1) H(\frac{p_2}{1-p_1}, \frac{p_3}{1-p_1} \dots \frac{p_m}{1-p_1}) \tag{3}$$

For a joint distribution over X, Y, the *joint entropy* is defined as

$$H(X,Y) = \sum_{i,j} p(x_i, y_j) \log_2 \frac{1}{p(x_i, y_j)} \tag{4}$$

Also recall the definition of the *conditional entropy* $H(X|Y)$:

$$\begin{aligned} H(X|Y) &= \sum_{i,j} p(x_i, y_j) \log_2 \frac{1}{p(x_i \mid y_j)} = \sum_j p(y_j) \sum_i p(x_i \mid y_j) \log_2 \frac{1}{p(x_i \mid y_j)} \\ &= \sum_{j=1}^{n} p(y_j) H(X|Y = y_j) \end{aligned} \tag{5}$$

For X and Y defines as above from $\mathcal{D}$ we also have

$$H(X|Y) = \sum_{j=1}^{n} p(y_j) H(X|Y = y_j) = \sum_{j=1}^{n} q(j) H(\overleftarrow{\mathcal{D}}[j]) = \sum_{j=1}^{n} q(j) H(X_j) \tag{6}$$

$H(Y|X)$ is defined similarly and we note that it may be the case that $H(X|Y) \neq H(Y|X)$. We may simply write H for the entropy if the purpose is given by the context. By default, we will denote by H the entropy computed using the binary logarithm; if a different logarithmic basis Δ is used to compute the entropy, we will explicitly write H_Δ.

We recall the definition of a graph *spanner*. Given a graph $G = (V, E)$, a subgraph $G' = (V, E')$ is a t-spanner of G if for every $u, v \in V$, $t \cdot d_G(u, v) \geq d_{G'}(u, v)$ and t is the *distortion* of the spanner. We say that $G' = (V, E')$ is a *graph metric* t-spanner if it is not a subgraph of G, i.e., it may have additional edges that are not in G.

4 A Lower Bound

We now establish an interesting connection to information theory and show that the conditional entropy serves as a natural metric for bounded network designs. In particular, we prove that the expected path length $\mathrm{BND}(\mathcal{D}, \Delta)$ in any demand-aware bounded network design, is at least in the order of the conditional entropy. Formally:

▶ **Theorem 2.** *Consider the joint frequency distributions $\mathcal{D}$. Let X, Y be the random variables distributed according to the marginal distribution of the sources and destinations in $\mathcal{D}$, respectively. Then*

$$\mathrm{BND}(\mathcal{D}, \Delta) \geq \Omega(\max(H_\Delta(Y|X), H_\Delta(X|Y))$$

Before delving into the proof, let $\mathrm{EPL}(\bar{p}, T)$ denote the expected path length in a tree T from the root to its nodes where the expectation it taking over a distribution $\bar{p}$. That is $\mathrm{EPL}(\bar{p}, T) = \sum_i p_i d_T(root, i)$. We recall the following well-known theorem:

▶ **Theorem 3** ([18], restated.). *Let $H(\bar{p})$ be the entropy of the frequency distribution $\bar{p} = (p_1, p_2, \dots, p_n)$. Let T be an optimal binary search tree built for the above frequency distribution. Then* $\mathrm{EPL}(\bar{p}, T) \geq \frac{1}{\log 3} H(\bar{p})$.

Namely, the entropy $H(\bar{p})$, is a lower bound on the expected path length from the root to the nodes in the tree. Note that, the proof of Theorem 3 in [18] holds for any optimal binary tree T, not necessarily a search tree. For higher degree graphs, we can extend the result:

▶ **Lemma 4.** *Let $H_\Delta(\bar{p})$ be the entropy (calculated using the logarithm of base Δ) of frequency distribution $\bar{p} = (p_1, p_2, \ldots, p_n)$. Let T be an optimal Δ-ary tree built for the above frequency distribution. Then,* $\mathrm{EPL}(\bar{p}, T) \geq \frac{1}{\log(\Delta+1)} H_\Delta(\bar{p})$.

The proof almost directly follows from the proof of Theorem 3 in [18], by extending properties of binary trees to Δ-ary trees, see [3] for details. We now prove the lower bound.

Proof of Theorem 2. The proof idea is to view any network as the union of n optimal trees, one for each individual node. While the resulting network may violate the degree requirement, it constitutes a valid lower bound. So we start by finding an optimal structure for each source node i, according to all its communication destinations $\overrightarrow{\mathcal{D}}[i]$: We construct n Δ-ary trees, and let T^i_Δ be the optimal tree for source node i built using $\overrightarrow{\mathcal{D}}[i]$. From Lemma 4, we have:

$$\mathrm{EPL}(\overrightarrow{\mathcal{D}}[i], T^i_\Delta) = \sum_{j=1}^{n} p(j|i) d_{T^i_\Delta}(i,j) = \Omega(H_\Delta(Y \mid X = i))$$

where $\mathrm{EPL}(\overrightarrow{\mathcal{D}}[i], T^i_\Delta)$ denotes the expected path length of T^i_Δ given $\overrightarrow{\mathcal{D}}[i]$ and $d_{T^i_\Delta}$ denotes the shortest path in T^i_Δ. Now consider any bounded degree network N_Δ and compare it to the forest T made up of n trees $T^1_\Delta, T^2_\Delta, \ldots, T^n_\Delta$. Then,

$$\begin{aligned}\mathrm{EPL}(\mathcal{D}, N_\Delta) &= \sum_{i=1}^{n} p(i) \cdot \mathrm{EPL}(\overrightarrow{\mathcal{D}}[i], N_\Delta) \geq \sum_{i=1}^{n} p(i) \cdot \mathrm{EPL}(\overrightarrow{\mathcal{D}}[i], T^i_\Delta) \\ &\geq \sum_{i=1}^{n} p(i) \cdot H_\Delta(Y \mid X = i) = \Omega(H_\Delta(Y|X))\end{aligned}$$

Similarly we can consider a set of trees optimized toward the incoming communication of node j, $\overleftarrow{\mathcal{D}}[j]$, and the marginal destination probability. We show:

$$\mathrm{EPL}(\mathcal{D}, N_\Delta) \geq \Omega(H_\Delta(X \mid Y))$$

Hence the theorem follows. ◀

5 Network Design for Sparse Distributions

We now present families of distributions which enable DANs that match the lower bound. Our approach will be based on replacing neighborhoods with near optimal binary (or Δ-ary) trees. Following the lower bound of Lemma 4, it is easy to show a matching upper bound (for a constant Δ).

▶ **Lemma 5.** *Let $\bar{p}$ be a probability distribution on a set of node destinations (sources) V, and let u be a single source (destination) node. Then one can design a tree T with u as a root node with degree one, connected to a Δ-ary tree over V such that the expected path length from u to all destinations (or from all sources to u) is:*

$$\mathrm{EPL}(\bar{p}, T) = \sum_i p_i \cdot d_T(u,i) \leq O(H_\Delta(\bar{p})) \tag{7}$$

Proof. The proof follows by designing a Huffman Δ-ary code over $\bar{p}$ (with expected code length less than $H_\Delta(\bar{p}) + 1$ [9]) and using it to build a rooted Δ-ary tree. While the nodes in the Huffman code are tree leaves, we can move them up to become internal nodes, which only improves the expected path length. ◀

5.1 Tree Distributions

A most fundamental class of distributions for which we can construct optimal network designs is based on trees.

▶ **Theorem 6.** *Let $\mathcal{D}$ be a communication request distribution such that $G_\mathcal{D}$ is a tree (i.e., ignoring the edge direction, $G_\mathcal{D}$ forms a tree). Let X, Y be the random variables of the sources and destinations in $\mathcal{D}$, respectively. Then, it is possible to generate a* DAN *N with maximum degree 8, such that*

$$\mathrm{EPL}(\mathcal{D}, N) \leq O(H(Y \mid X) + H(X \mid Y))$$

This is asymptotically optimal.

Proof. We generate N as follows. Consider an arbitrary node as the root of the tree $G_\mathcal{D}$, call this tree $T_\mathcal{D}$, and consider the parent-child relationship implied by the root. Let $\pi(i)$ denote the parent of node i. Let $\overrightarrow{c}_i$ denote the communication distribution *from* v_i to its children (not including its single parent) and $\overrightarrow{\mathcal{D}}[i]$ denote the communication distribution *from* i to its neighbors (children and parent). Let $p_i^\pi = \overrightarrow{\mathcal{D}}[i][\pi(i)]$ denote the corresponding entry in $\overrightarrow{\mathcal{D}}[i]$ for the parent of i. From entropy Eq. (3), we have that $(1 - p_i^\pi)H(\overrightarrow{c}_i) \leq H(\overrightarrow{\mathcal{D}}[i])$. Similarly we define $\overleftarrow{c}_i$ and $\overleftarrow{\mathcal{D}}[i]$ as the communication distribution *to* v_i, from its children and neighbors respectively.

The construction has two phases. In the first phase we replace outgoing edges. For each node i replace the edges between i and its *children* with a binary tree according to $\overrightarrow{c}_i$ and the method of [18] (or Lemma 5 for a general Δ) for creating a near optimal binary tree. Let $\overrightarrow{B}_i$ denote this tree and recall that $\mathrm{EPL}(\overrightarrow{c}_i, \overrightarrow{B}_i) \leq O(H(\overrightarrow{c}_i))$. Note that every node i may appear in at most two trees $\overrightarrow{B}_i$ and $\overrightarrow{B}_{\pi(i)}$; in $\overrightarrow{B}_i$ its degree is one and in $\overrightarrow{B}_{\pi(i)}$ its degree is at most 3, so the outgoing degree of each node is at most 4 after this phase.

In the second phase we take care of the remaining incoming edges from children to parents. For each node i replace the edges from its *children* to it with a binary tree according to $\overleftarrow{c}_i$ and the method of [18] for creating a near optimal binary tree. Let $\overleftarrow{B}_i$ denote this tree and recall that $\mathrm{EPL}(\overleftarrow{c}_i, \overleftarrow{B}_i) \leq O(H(\overleftarrow{c}_i))$. Note that every node i may appear in at most two trees $\overleftarrow{B}_i$ and $\overleftarrow{B}_{\pi(i)}$; in $\overleftarrow{B}_i$ i's degree is one and in $\overleftarrow{B}_{\pi(i)}$ i's degree is at most 3. Thus, the incoming degree of each node is bounded by 4 after this phase.

Now we bound $\mathrm{EPL}(\mathcal{D}, N)$ by bounding the expected path lengths in the corresponding binary trees of each node, first considering all edges from parent to children and then all edges from children to parent. Let $p(i)$ and $q(i)$ denote the probabilities that node i will be a source and a destination of a communication request, respectively. Then:

$$\begin{aligned}\mathrm{EPL}(\mathcal{D},N) &\leq \sum_{(u,v)\in\mathcal{D}} p(u,v)d_N(u,v)\\ &= \sum_{(\pi(i),i)\in T_{\mathcal{D}}} p(\pi(i),i)d_N(\pi(i),i) + \sum_{(i,\pi(i))\in T_{\mathcal{D}}} p(i,\pi(i))d_N(i,\pi(i))\\ &= \sum_{i=1}^{n} p(i)\mathrm{EPL}(\overrightarrow{c}_i,\overrightarrow{B}_i) + \sum_{i=1}^{n} q(i)\mathrm{EPL}(\overleftarrow{c}_i,\overleftarrow{B}_i)\\ &\leq \sum_{i=1}^{n} p(i)H(\overrightarrow{\mathcal{D}}[i]) + \sum_{i=1}^{n} q(i)H(\overleftarrow{\mathcal{D}}[i]) = H(Y\mid X)+H(X\mid Y)\end{aligned}$$

This matches our lower bound in Theorem 2. ◀

5.2 General Sparse Distributions

Asymptotically optimal demand-aware networks can even be designed for general sparse distributions.

▶ **Theorem 7.** *Let $\mathcal{D}$ be a communication request distribution where Δ_{avg} is the average degree in $G_{\mathcal{D}}$ (so the number of edges $m = \frac{\Delta_{\mathrm{avg}}\cdot n}{2}$). Let X, Y be the random variables of the sources and destinations in $\mathcal{D}$, respectively. Then, it is possible to generate a* DAN *N with maximum degree $12\Delta_{\mathrm{avg}}$, such that*

$$\mathrm{EPL}(\mathcal{D},N) \leq O(H(Y\mid X)+H(X\mid Y)) \tag{8}$$

This is asymptotically optimal when Δ_{avg} is a constant.

Proof. Recall that $G_{\mathcal{D}}$ (for short G) is a directed graph and define in-degree and out-degree in the canonical way. Let the (undirected) degree of a node, be the sum of its in-degree and out-degree and denote the average degree as Δ_{avg}. Denote the $n/2$ nodes with the lowest degree in G as *low degree* nodes and the rest as *high degree* nodes. Note that each low degree node has a degree at most $2\Delta_{\mathrm{avg}}$ and both its in-degree and out-degree must be low. A node with out-degree (in-degree) larger than $2\Delta_{\mathrm{avg}}$ is called a *high out-degree* (*high in-degree*) node (some nodes are neither low or high degree nodes).

The construction of N will be done in two phases. In the first phase, we consider only (directed) edges (u,v) between a high out-degree u and a high in-degree node v. We subdivide each such edge with two edges that connect u to v via a helping low degree node ℓ, i.e., removing the directed edge (u,v) and adding the edges (u,ℓ) and (v,ℓ). Note that there are at most m such edges, so we can distribute the help between low degree nodes in such a way that each low degree node helps at most Δ_{avg} such edges. Call the resulting graph G'.

Accordingly, we also create a new matrix $\mathcal{D}'$ which, initially, is identical to $\mathcal{D}$. Then for each (u,v) and ℓ as above we set $\mathcal{D}'(u,v)=0$, $\mathcal{D}'(u,\ell)=\mathcal{D}(u,l)+\mathcal{D}(u,v)$ and $\mathcal{D}'(\ell,v)=\mathcal{D}(l,v)+\mathcal{D}(u,v)$. Note that $\mathcal{D}'$ is not a distribution matrix anymore, as the sum of all the entries is more than one, but it has the following property: For each high degree node i, we have $H(\overrightarrow{\mathcal{D}'}[i]) \leq H(\overrightarrow{\mathcal{D}}[i]$ and $H(\overleftarrow{\mathcal{D}'}[i]) \leq H(\overleftarrow{\mathcal{D}}[i])$ (see Eq. (2)).

In the second phase, we construct N from G'. Consider each node i with high out-degree and create a nearly optimal binary tree $\overrightarrow{B}^i$ according to $\overrightarrow{\mathcal{D}'}[i]$ using the method of [18]. Add an edge from i to the root of $\overrightarrow{B}^i$ and delete all the out-edges from i from G'. Similarly consider each node j with high in-degree and create a nearly optimal binary tree $\overleftarrow{B}_j$ according to $\mathcal{D}'[i]$ using the method of [18]. Add an edge from j to the root of $\overleftarrow{B}_j$ and delete all the in-edges of j from G'. This completes the construction of N.

We first bound the maximum degree in N. First consider a low degree node ℓ, helping an edge (u, v), i.e., u is high out-degree and v is high-indegree. So ℓ is part of both u's and v's binary tree, hence ℓ's degree increases by at most 6 (two times degree 3 for being an internal node). Note that ℓ needs to help at most Δ_{avg} edges itself. For each of these Δ_{avg} edges, ℓ's degree will be at most 6, resulting in a degree of $6\Delta_{\text{avg}}$. Since ℓ's degree was at most $2\Delta_{\text{avg}}$, in the worst case, ℓ was associate with $2\Delta_{\text{avg}}$ high in-degree or out-degree nodes, i.e., ℓ will be present in all these $2\Delta_{\text{avg}}$ trees, which results in another $6\Delta_{\text{avg}}$ degrees for ℓ. In total, ℓ's degree is $12\Delta_{\text{avg}}$. If a node h has both high out-degree and high in-degree, then its degree will be two: h is connected to the root of the tree created of its out-edges and to the root of the tree created of its in-edges. If a node u is only a high out-degree node, its degree in N is bounded by $6\Delta_{\text{avg}} + 1$ (and similarly for a node u which is only a high in-degree node). If a node is neither high nor low degree, then its degree in N is bounded by $12\Delta_{\text{avg}}$ (originally it was up to $4\Delta_{\text{avg}}$ in G'). We now bound $\text{EPL}(\mathcal{D}, N)$. Recall that from Lemma 5 and Eq. (2), we have,

$$\text{EPL}(\overrightarrow{\mathcal{D}'}[i], \overrightarrow{B}_i) \leq O(H(Y \mid X = i))$$

and

$$\text{EPL}(\overleftarrow{\mathcal{D}'}[j], \overleftarrow{B}_j) \leq O(H(X \mid Y = j))$$

For each request (i, j) in $\mathcal{D}$ there are two possibilities for the route on N: either the edge $(i, j) \in N$ is a direct route, or the route goes via $\overrightarrow{B}_i$ or $\overleftarrow{B}_j$ or both. Let $\mathcal{O}$ and $\mathcal{I}$ be the set of high out-degree and in-degree nodes, respectively. Then:

$$\begin{aligned}
\text{EPL}(\mathcal{D}, N) &= \sum_{(u,v)\in\mathcal{D}} p(u,v) d_N(u,v) \\
&\leq \sum_{(i,j)\notin\mathcal{O}\cup\mathcal{I}} p(u,v) + \sum_{i\in\mathcal{O}} p(i)\text{EPL}(\overrightarrow{\mathcal{D}}[i]), \overrightarrow{B}_i) + \sum_{j\in\mathcal{I}} q(j)\text{EPL}(\overleftarrow{\mathcal{D}}[j])\overleftarrow{B}_j) \\
&= \sum_{i\notin\mathcal{O}} p(i) + \sum_{j\notin\mathcal{I}} q(j) + \sum_{i\in\mathcal{O}} p(i)\text{EPL}(\overrightarrow{\mathcal{D}}[i]), \overrightarrow{B}_i) + \sum_{j\in\mathcal{I}} q(j)\text{EPL}(\overleftarrow{\mathcal{D}}[j])\overleftarrow{B}_j) \\
&\leq O(H(X \mid Y) + H(Y \mid X))
\end{aligned}$$

This matches our lower bound in Theorem 2. ◀

6 Regular and Uniform Distributions

Another large family of distributions for which demand-aware networks can be designed are regular and uniform distributions $\mathcal{D}$. While it is easy to see that both conditions can be relaxed such that the supported distributions can be "*nearly* regular" and "*nearly* uniform", for ease of presentation, we keep the conditions strict in what follows.

We first establish an interesting connection to spanners. As we will see, this connection will provide a simple and powerful technique to design a wide range of demand-aware networks meeting the conditional entropy lower bound.

▶ **Theorem 8.** *Let $\mathcal{D}$ be an arbitrary (possibly dense) regular and uniform request distribution. It holds that if we can find a constant and sparse (i.e., constant distortion, linear sized) spanner for $G_{\mathcal{D}}$, we can design a constant degree* DAN *N providing an expected path length of*

$$\text{EPL}(\mathcal{D}, N) \leq O(H(Y \mid X) + H(X \mid Y)) \tag{9}$$

This is asymptotically optimal.

In other words, for regular and uniform distributions, the network design problem boils down to finding a constant[1] sparse spanner: as we will see, we can automatically transform this spanner into an efficient network (which may contain auxiliary edges). The remainder of this section is devoted to the proof of the theorem.

Assume that $\mathcal{D}$ is r-regular and uniform. Recall that in this case $H(Y \mid X) = H(X \mid Y) = \log r$, so $\text{BND}(\mathcal{D}, \Delta) \geq \Omega(H(Y \mid X))$ where Δ is a constant. We now describe how to transform a constant sparse (but potentially irregular) spanner for $G_{\mathcal{D}}$ into a constant-degree host network N with $\text{EPL}(\mathcal{D}, N) \leq O(\log r)$. This will be done using a similar degree reduction technique as discussed earlier (in the proof of Theorem 7).

▶ **Lemma 9.** *Let G be a graph of maximum degree $\Delta_{\max}$ and an average degree Δ_{avg}. Then, we can construct a graph G' with maximum degree $8\Delta_{\text{avg}}$ which is a graph metric $\log \Delta_{\max}$-spanner of G, i.e., $d_{G'}(u, v) \leq 2 \log \Delta_{\max} \cdot d_G(u, v)$.*

Proof. Let us call the $n/2$ nodes with the lowest degree in G the *low degree* nodes and the remaining nodes *high degree* nodes. By the pigeon hole principle, each low degree node has a degree at most $2\Delta_{\text{avg}}$. The construction of G' proceeds in two phases. In the first phase we take every edge between high degree nodes u, v and subdivide it with two edges that connect u to v via a helping low degree node ℓ, i.e., removing the edge (u, v) and adding the edges (u, ℓ) and (v, ℓ). Note that there are at most m edges connecting high degree nodes so we can distribute the help between low degree nodes such that each low degree node helps to at most Δ_{avg} such edges.

In the second phase we consider each high degree node u and replace the set of edges between u and its neighbors, $\Gamma(u)$, with a balanced binary tree that connects u as the root and $\Gamma(u)$ as remaining nodes of the tree. Denote as B_u this tree and note that the height of B_u is at most $\log(|\Gamma(u)| + 1)$. We leave edges between low degree nodes as in G.

Let us analyze the degrees in G'. Since every high degree node u in G' only connects to low degree nodes, it is only a member of B_u and its degree reduces to 2 in G'. Now consider a low degree node ℓ: for each edge (u, v) it helps, ℓ participates in B_u and B_v. Hence, its degree increases by at most 6 in G' compared to G. Overall, for helping high degree nodes, the degree of ℓ can increase by $6\Delta_{\text{avg}}$. Together with its original neighbors from G, the degree of ℓ in G' can be at most $8\Delta_{\text{avg}}$.

Next consider the distortion of G'. The distortion between neighboring low degree nodes is one. The distortion between neighboring high degree nodes is at most twice $\log \Delta_{\max}$ and the distortion between a neighboring high and low degree is at most $\log \Delta_{\max}$.

So, $d_{G'}(u, v) \leq 2 \log \Delta_{\max} \cdot d_G(u, v)$ for all u, v in G'. ◀

We will apply Lemma 9 to prove Theorem 8.

Proof of Theorem 8. Let S be a constant and sparse spanner of $G_{\mathcal{D}}$ (G could be either a subgraph or a metric spanner of max degree asymptotically not larger than $G_{\mathcal{D}}$) of degree at most r. Lemma 9 then tells us how to transform S to a DAN N of degree Δ_{avg}. Since S is a constant spanner there is a constant c such that,

$$\text{EPL}(\mathcal{D}, S) = \sum_{(u,v)\in\mathcal{D}} \text{p}(u, v) \cdot d_S(v, v) = c \tag{10}$$

[1] To be precise, a spanner with constant *average* distortion will be sufficient (see [3] for details). However, for simplicity, we leave it as a constant spanner.

Since S has maximum degree r, according to Lemma 9, it has a graph metric spanner N such that, the distance of any source-destination pair of $G(\mathcal{D})$ in N is at most $2\log r$ times their distance in S. Hence:

$$\begin{aligned}\mathrm{EPL}(\mathcal{D}, N) &= \sum_{(u,v)\in\mathcal{D}} \mathrm{p}(u,v)\cdot d_N(u,v) \leq \sum_{(u,v)\in\mathcal{D}} \mathrm{p}(u,v)\cdot d_S(u,v)\cdot 2\log r \\ &\leq \log r \cdot\ \mathrm{EPL}(\mathcal{D}, S) = O(\log r) = O(H(Y \mid X))\end{aligned}$$

The last equality holds since $\mathcal{D}$ is r-regular and uniform. The bound is asymptotically optimal when Δ is a constant: it matches our lower bound in Theorem 2. ◀

Theorem 8 allows us to simplify the design of asymptotically optimal networks for uniform and regular distributions $\mathcal{D}$ where $G_{\mathcal{D}}$ has a constant sparse spanner. In particular, the approach can be used to design optimal networks for the following large families of distributions which are known to have a constant and sparse graph spanners.

▶ **Corollary 10.** *Let $\mathcal{D}$ describe a uniform and regular communication request distribution. Then, it is possible to generate a constant degree* DAN *N such that*

$$\mathrm{EPL}(\mathcal{D}, N) \leq O(H(Y \mid X) + H(X \mid Y)) \tag{11}$$

in the following scenarios:

- *If, for a constant $c \geq 1$, $G_{\mathcal{D}}$ has a minimum degree $\Delta \geq n^{\frac{1}{c}}$.*[2]
- *If $G_{\mathcal{D}}$ forms a hypercube with $n\log n$ edges.*
- *If $G_{\mathcal{D}}$ forms a (possibly dense) chordal graph.*

See [3] for the proof.

We round off our study of uniform and regular distributions by considering one more interesting family of (possibly very dense) distributions: distributions $\mathcal{D}$ which describe a bounded and *local* doubling dimension, note that this family is more general than the standard bounded doubling dimension graphs which are sparse.

First, recall that a metric space (V, d) has a constant doubling dimension if and only if there exists a constant λ such that every ball of radius r in V can be covered by λ balls of half the radius $r/2$, for all $r \geq 1$. In general, the smallest λ which satisfies this property for a metric space is called *doubling constant* and $\log_2 \lambda$ is called the *doubling dimension* [6, 12, 13, 14]. A metric space is called *bounded* (a.k.a. constant or low) doubling dimension if λ is a constant. There has been a wide range of work on spanners for bounded doubling dimension metrics [5, 6, 13, 14]. However, if the metric is imposed by a graph metric (via shortest paths) then a bounded doubling dimension implies that the graph is nearly regular, of bounded (constant) degree and sparse. Theorem 7 already solved the case of sparse $G_{\mathcal{D}}$, even for non-uniform and irregular distributions.

In the following, however, we are interested in a more general notion of doubling dimension, which allows a higher density, unbounded degree: we call it *locally-bounded doubling dimension*:

▶ **Definition 11** (Locally-Bounded Doubling Dimension (LDD))**.** $G_{\mathcal{D}}$ implied by the distribution $\mathcal{D}$ has a *locally-bounded doubling dimension* if and only if there exists a constant λ such that

[2] In this case the constant in the O notation depends linearly on c.

the 2-hop neighbors of any node u are covered by at most λ 1-hop neighbors. Formally, for each $u \in V$, there exists a set of nodes $y_1, y_2, ...y_\lambda$, such that:

$$B(u,2) \subseteq \bigcup_{i=1}^{\lambda} B(y_i,1)$$

where $B(u,r)$ are the set of nodes that are at distance at most r-hops from u in $G_\mathcal{D}$.

Clearly, every bounded doubling dimension graph is also of locally-bounded doubling dimension, but the converse is not true. In particular, the latter enables graphs which could be dense, with unbound degree, and possibly with irregularity of degree.

In the remainder of this section, we will prove the following theorem.

▶ **Theorem 12.** *Let $\mathcal{D}$ describe a uniform and regular communication request distribution of locally-bounded doubling dimension. Then it is possible to design a constant degree* DAN N *such that*

$$\mathrm{EPL}(\mathcal{D}, N) \leq O(H(Y \mid X) + H(X \mid Y)) \tag{12}$$

This is asymptotically optimal.

Proof. Again, our proof strategy is to employ Theorem 8. Accordingly, we show that a constant sparse spanner exists for locally-bounded doubling dimension networks. In particular, we will design this spanner based on an ϵ-net construction. We first recall the definition of *ϵ-nets* [6].

▶ **Definition 13** (ϵ-net)**.** A subset V' of V is an *ϵ-net* for a graph $G = (V, E)$ if it satisfies the following two conditions:

1. for every u, $v \in V'$, $d_G(u,v) > \epsilon$
2. for each $w \in V$, there exists at least one $u \in V'$ such that, $d_G(u,w) \leq \epsilon$

Let $G_\mathcal{D} = (V, E)$ be a locally-bounded doubling dimension network. We now first construct a spanner S' of $G_\mathcal{D}$ which is a subgraph of $G_\mathcal{D}$, using the following ($\epsilon = 2$)-net: we sort all nodes according to decreasing (remaining) degrees, and iteratively select the high-degree nodes into the 2-net one-by-one and remove their 2-neighborhoods. Clearly, after this process, each node is either part of the 2-net or has a 2-net node at distance at most 2-hops, and we have computed a legal 2-net.

To form the spanner S, we next arbitrarily match each node u not in the 2-net to one of its nearest 2-net nodes v, and select the edges along a shortest path from u to v into the spanner S. This results in a forest of connected components (2-layered stars). We call these connected components *clusters* and the corresponding nodes in the 2-net *cluster heads*. We denote the cluster associated to the net node u by $Cl(u)$.

We next connect the connected clusters to each other, in a sparse manner. Towards this end, we connect each pair of clusters, with an arbitrary single edge, if they contain at least one pair of communicating nodes in $G_\mathcal{D}$. We can claim the following.

▶ **Lemma 14.** *S is a constant and sparse spanner of $G_\mathcal{D}$ (with distortion 9) .*

Proof. Let (u,v) be an edge in $G_\mathcal{D}$ and $u \in Cl(u)$, $v \in Cl(v)$. By construction, there are nodes $x \in Cl(u)$ and $y \in Cl(v)$ that are connected by an edge in S, and hence there is a path $u, C(u), x, y, C(v), v$ in S. Therefore, $d_S(u,v) \leq d_S(u, Cl(u)) + d_S(Cl(u), x) + d_S(x,y) + d_S(y, Cl(v)) + d_S(Cl(v), v) \leq 9$.

The spanner is also sparse: in a nutshell, due to the 2-net properties, we know that the distance between communicating cluster heads is at most 5: since in a locally doubling dimension graph the number of cluster heads at distance 5 is bounded, only a small number of neighboring clusters will communicate. More formally, after associating each node to some unique cluster, we have a linear number of edges in the spanner. Next we bound the number of outgoing edges from each cluster. Let (u, v) be such an edge where $u \in Cl(u)$, $v \in Cl(v)$. Let the cluster heads of $Cl(u)$ and $Cl(v)$ be i and j, respectively. By construction i and j are at most at distance 5 in $G_{\mathcal{D}}$, i.e., $d_{G_{\mathcal{D}}}(i, j) \leq 5$. So, if we can bound the number of 2-net nodes which lie within 5 hops from some net node i, it will give us a bound on the number of edges which we add between $Cl(u)$ and other clusters. According to Definition 11, all the two hop neighbors of i can be covered within one hop neighbors of λ nodes, where λ is the corresponding doubling constant. If we consider two hop neighbors of all these λ many nodes, they cover all the 3 hop neighbors of i. To cover the 2 hop neighbors of each of these nodes, we again require one hop neighbors of λ nodes. So, to cover all 3 hop neighbors of i, we require at most λ^2 one hop neighbors. Inductively, by repeating this argument, we require one hop neighbors of at most λ^4 nodes to cover all the 5 hop neighbors of i. Since we constructed a 2-net, each of these λ^4 balls with radius one contains at most one 2-net node. Hence there are at most λ^4 2-net nodes which are at a distance 5 hops or less from i. Consequently, there are at most λ^4 inter-cluster edges associated to some cluster $Cl(u)$, or cluster head i. Since there can not be more than linear number of clusters, hence the number of edges in S' is also linear. ◀

Using Lemma 14 we can directly use Theorem 8 and conclude the proof of Theorem 12. ◀

In fact, it turns out that if we consider a *metric* spanner, and by using auxiliary edges, we can improve the above distortion and constract better constant sparse spanner S'. The idea is to add inter-cluster edges only between the cluster heads. This will reduce the distortion to 5 while keeping the same number to total edges. The degree of each node in S' will increase by at most a constant, λ^4. Adjusting Theorem 8 respectively to support metric spanners (and only a subgraph spanner) will enable us to use S' instead of S.

7 Conclusion

This paper initiated the study of a fundamental network design problem. While our work is motivated in particular by emerging technologies for more flexible datacenter interconnects as well as by peer-to-peer overlays, we believe that our model is very natural and of interest beyond this specific application domain considered in this paper. For example, the design of a sparse, bounded-degree and distance-preserving network can also be understood from the perspective of graph sparsification [27]: the designed network can be seen as a compact representation of the original network.

The subject of bounded network design offers several interesting avenues for future research. In particular, while we presented asymptotically optimal network design algorithms for a wide range of distributions and while we believe that the entropy is the right measure to assess network designs, there remain several (dense) distributions for which the quest for optimal network designs remains open, perhaps also requiring us to explore alternative flavors of graph entropy.

Acknowledgments. We would like to thank Michael Elkin for many inputs and discussions.

References

1 Chen Avin, Michael Borokhovich, Bernhard Haeupler, and Zvi Lotker. Self-adjusting grid networks to minimize expected path length. In *Proc. SIROCCO*, pages 36–54, 2013.

2 Chen Avin, Michael Borokhovich, and Stefan Schmid. OBST: A self-adjusting peer-to-peer overlay based on multiple BSTs. In *Peer-to-Peer Computing (P2P), 2013 IEEE Thirteenth International Conference on*, pages 1–5. IEEE, 2013.

3 Chen Avin, Kaushik Mondal, and Stefan Schmid. Demand-aware network designs of bounded degree. In *arXiv report no. 1705.06024*, 2017.

4 Hubert T-H Chan, Michael Dinitz, Anupam Gupta, et al. Spanners with slack. In *ESA*, volume 6, pages 196–207. Springer, 2006.

5 T-H Hubert Chan and Anupam Gupta. Small hop-diameter sparse spanners for doubling metrics. *Discrete & Computational Geometry*, 41(1):28–44, 2009.

6 T.-H. Hubert Chan, Anupam Gupta, Bruce M. Maggs, and Shuheng Zhou. On hierarchical routing in doubling metrics. *ACM Trans. Algorithms*, (4):55:1–55:22, 2016.

7 Moses Charikar, Mohammad Taghi Hajiaghayi, Howard Karloff, and Satish Rao. l_2^2 spreading metrics for vertex ordering problems. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 1018–1027. Society for Industrial and Applied Mathematics, 2006.

8 C. Chen, R. Vitenberg, and H. A. Jacobsen. Scaling construction of low fan-out overlays for topic-based publish/subscribe systems. In *2011 31st International Conference on Distributed Computing Systems*, pages 225–236, June 2011.

9 Thomas M Cover and Joy A Thomas. *Elements of information theory*. John Wiley & Sons, 2012.

10 Erik D Demaine, Dion Harmon, John Iacono, Daniel Kane, and Mihai Patraşcu. The geometry of binary search trees. In *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 496–505. SIAM, 2009.

11 Uriel Feige and James R Lee. An improved approximation ratio for the minimum linear arrangement problem. *Information Processing Letters*, 101(1):26–29, 2007.

12 Pierre Fraigniaud, Emmanuelle Lebhar, and Zvi Lotker. A doubling dimension threshold θ (log log n) for augmented graph navigability. In *ESA*, pages 376–386. Springer, 2006.

13 Anupam Gupta, Robert Krauthgamer, and James R Lee. Bounded geometries, fractals, and low-distortion embeddings. In *Proc. IEEE FOCS*, pages 534–543, 2003.

14 Sariel Har-Peled and Manor Mendel. Fast construction of nets in low-dimensional metrics and their applications. *SIAM Journal on Computing*, 35(5):1148–1184, 2006.

15 Lawrence H Harper. Optimal assignments of numbers to vertices. *Journal of the Society for Industrial and Applied Mathematics*, 12(1):131–135, 1964.

16 Su Jia, Xin Jin, Golnaz Ghasemiesfeh, Jiaxin Ding, and Jie Gao. Competitive analysis for online scheduling in software-defined optical wan. In *Proc. IEEE INFOCOM*, 2017.

17 M. Ghobadi et al. Projector: Agile reconfigurable data center interconnect. In *Proc. ACM SIGCOMM*, pages 216–229, 2016.

18 Kurt Mehlhorn. Nearly optimal binary search trees. *Acta Inf.*, 5:287–295, 1975.

19 Adam Meyerson and Brian Tagiku. Minimizing average shortest path distances via shortcut edge addition. In *Proc. APPROX/RANDOM*, pages 272–285, Berlin, Heidelberg, 2009.

20 Melih Onus and Andréa W Richa. Minimum maximum-degree publish–subscribe overlay network design. *IEEE/ACM Transactions on Networking*, 19(5):1331–1343, 2011.

21 Melih Onus and Andréa W Richa. Parameterized maximum and average degree approximation in topic-based publish-subscribe overlay network design. *Computer Networks*, 94:307–317, 2016.

22 David Peleg and Alejandro A. Schäffer. Graph spanners. *Journal of Graph Theory*, 13(1):99–116, 1989.

23 Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network's (datacenter) network. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 123–137. ACM, 2015.

24 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, June 2016.

25 Ankit Singla. Fat-free topologies. In *Proc. 15th ACM Workshop on Hot Topics in Networks (HotNets)*, pages 64–70, 2016.

26 Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, July 1985.

27 Daniel A Spielman and Shang-Hua Teng. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *Proc. ACM STOC*, pages 81–90, 2004.

Certification of Compact Low-Stretch Routing Schemes*

Alkida Balliu[†1] **and Pierre Fraigniaud**[‡2]

1 **Gran Sasso Science Institute, L'Aquila, Italy**
2 **CNRS and University Paris Diderot, France**

Abstract

On the one hand, the correctness of routing protocols in networks is an issue of utmost importance for guaranteeing the delivery of messages from any source to any target. On the other hand, a large collection of *routing schemes* have been proposed during the last two decades, with the objective of transmitting messages along short routes, while keeping the routing tables small. Regrettably, all these schemes share the property that an adversary may modify the content of the routing tables with the objective of, e.g., blocking the delivery of messages between some pairs of nodes, without being detected by any node.

In this paper, we present a simple *certification* mechanism which enables the nodes to locally detect any alteration of their routing tables. In particular, we show how to locally verify the stretch-3 routing scheme by Thorup and Zwick [SPAA 2001] by adding certificates of $\widetilde{O}(\sqrt{n})$ bits at each node in n-node networks, that is, by keeping the memory size of the same order of magnitude as the original routing tables. We also propose a new *name-independent* routing scheme using routing tables of size $\widetilde{O}(\sqrt{n})$ bits. This new routing scheme can be locally verified using certificates on $\widetilde{O}(\sqrt{n})$ bits. Its stretch is 3 if using handshaking, and 5 otherwise.

1998 ACM Subject Classification C.2.2 Network Protocols, C.2.4 Distributed Systems, G.2.2 Graph Theory

Keywords and phrases Distributed verification, compact routing, local computing

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.6

1 Introduction

Context. A *routing scheme* is a mechanism enabling to deliver messages from any source to any target in a network. The latter is typically modeled as an undirected connected weighted graph $G = (V, E)$ where V models the set of routers and E models the set of communication links between routers. All edges incident to a degree-d node are labeled from 1 to d, in an arbitrary manner, and the label at a node u of an incident edge e is called the *port number* of edge e at u. A routing scheme consists of a way of assigning a routing *table* to every node of the given network. These tables should contain enough information so that, for every target node t, each node is able to compute the port number of the incident edge through which it should forward a message of destination t. The routing tables must collectively guarantee that every message of any source s and any target t will eventually be delivered to t.

* A full version of the paper is available at `https://arxiv.org/abs/1704.06070`.
† Additional support from ANR Project DESCARTES. This work was done during the first author visit to Institut de Recherche en Informatique Fondamentale (IRIF) in 2016-2017.
‡ Additional support from ANR Project DESCARTES and Inria Project GANG.


© Alkida Balliu and Pierre Fraigniaud;
licensed under Creative Commons License CC-BY

Two scenarios are generally considered in the literature. One scenario allows the routing scheme to assign *names* to the nodes, and each target is then identified by its given name. The other, called *name independent*, is assuming that fixed names are given a priori (typically, a name is restricted to be the identity of a node), and the scheme cannot take benefit of naming nodes for facilitating routing.

Among many criteria for evaluating the quality of routing schemes, including, e.g., the time complexity for constructing the routing tables, the two main parameters characterizing a routing scheme are the *size* of its routing tables and names, and the *stretch*. The stretch of a routing scheme is the maximum, taken over all pairs of source-target nodes, of the ratio between the length of the route generated by the scheme from the source to the target, and the length of a shortest path between these two nodes. During the last two decades, there has been an enormous effort to design *compact* routing scheme (i.e., schemes using small tables) of *low* stretch (i.e., with stretch upper bounded by a constant) – see, e.g., [1, 2, 3, 4, 11, 16, 21, 22, 24]. A breakthrough was achieved in [24] where almost tight tradeoffs between size and stretch were explicitly demonstrated. In particular, [24] showed how to design a routing scheme with tables of size $\widetilde{O}(\sqrt{n})$ bits and stretch 3, in any network[1].

All the aforementioned routing schemes share the property that nodes do not have the capability to realize that the routing tables have been modified (either involuntarily or by an attacker). That is, a group of nodes may be provided with routing tables which look consistent with a desired routing scheme, but which do not achieve the desired performances of that scheme (e.g., large stretch, presence of loops, etc.). Indeed, the nodes are not provided with sufficient information to detect such an issue locally, that is, by having each node inspecting only the network structure and the tables assigned to nodes in its vicinity.

Objective. The objective of this paper is, given a routing scheme, to design a mechanism enabling each node to locally detect the presence of falsified routing tables, in the following sense. If some tables are erroneous, then at least one node must be able to detect that error by running a verification algorithm exchanging messages only between neighboring nodes.

Our mechanism for locally verifying the correctness of routing tables is inspired from proof-labeling schemes [20]. It is indeed based on assigning to each node a *certificate*, together with its routing table, and designing a distributed verification algorithm that checks the consistency of these certificates and tables by having each node inspecting only its certificate and its routing table, and the certificate and routing table of each of its neighbors. The set of certificates assigned to the nodes and the verification algorithm running at all nodes in parallel must satisfy that: (1) if all tables are correctly set, then, with some appropriate certificates, all nodes *accept*, and (2) if one or more tables are incorrectly set, then, for every assignment of the certificates, at least one node must *reject*. The second condition guarantees that the verification algorithm cannot be cheated: if the tables are incorrect, there are no ways of assigning the certificates such that all nodes accept.

Rephrasing the objective of the paper, our goal is to assign certificates to nodes, of size not exceeding the size of the routing tables, enabling the nodes to collectively verify the correctness of the routing tables, by having each node interacting with its neighbors only.

Our Results. We show how to locally verify the stretch-3 size-$\widetilde{O}(\sqrt{n})$ routing scheme by Thorup and Zwick [24]. Our certification mechanism uses certificates of $\widetilde{O}(\sqrt{n})$ bits at each node, that is, these certificates have size of the same order of magnitude as the original routing tables. Hence, verifying the scheme in [24] can be done without modifying the scheme,

[1] The notations $\widetilde{O}$ and $\widetilde{\Omega}$ ignore polylogarithmic factors.

Table 1 Summary of our results compared to previous work. All the listed routing schemes have space complexity of $\widetilde{O}(\sqrt{n})$ bits. Our verification algorithms use certificates on $\widetilde{O}(\sqrt{n})$ bits.

scheme	stretch	name independent	verifiable	comment
[24]	3	no	**yes**	–
[3]	5	yes	**?**	–
[2]	3	yes	**?**	–
this paper	**5**	**yes**	**yes**	–
this paper	**3**	**yes**	**yes**	handshaking

and without increasing the memory space consumed by that scheme. We also show that the same holds for the whole hierarchy of routing schemes proposed in [24] for providing a tradeoff between size and stretch.

The situation appears to be radically different for name-independent routing schemes. The stretch-3 name-independent routing scheme by Abraham et al. [2] also uses tables of size $\widetilde{O}(\sqrt{n})$ bits. However, each table includes references to far away nodes, whose validity does not appear to be locally verifiable using certificates of reasonable size. On the other hand, a simplified version of the scheme in [2] can be verified locally with certificates of size $\widetilde{O}(\sqrt{n})$ bits, but its stretch becomes at least 7. Therefore, we propose a new name-independent routing scheme, with tables of size $\widetilde{O}(\sqrt{n})$ bits that can be verified using certificates on $\widetilde{O}(\sqrt{n})$ bits as well. This new routing scheme has stretch at most 5, and the stretch can even be reduced to 3 using handshaking[2]. The routing scheme of Arias et al. [3] has also stretch 5, but it does not appear to be locally verifiable with certificates of reasonable size, and using handshaking does not enable to reduce the stretch.

All our results are summarized in Table 1.

Related Work. The design of *compact* routing tables, and the explicit identification of tradeoffs between the table size and the routes length was initiated thirty years ago, with the seminal work in [22] and [21]. Since then, a large amount of papers were published on this topic, aiming at refining these tradeoffs, and at improving different aspects of the routing schemes, including routing in specific classes of graphs (see [15, 17]). In particular, routing schemes were designed for trees in [11, 24], with space complexity $O(\log^2 n/\log\log n)$ bits[3]. This space complexity was shown to be optimal in [12].

It was proved [18] that, in n-node networks, any shortest path routing scheme requires tables of size $\widetilde{\Omega}(n)$ bits. The aforementioned routing scheme in [24] with stretch 3 and space complexity $\widetilde{O}(\sqrt{n})$ bits was shown to be optimal in [16], in the following sense: no routing scheme with space complexity $o(n)$ bits can achieve a stretch $s < 3$, and, assuming the correctness of a conjecture by Erdős regarding a tradeoff between girth and edge density in graphs, every routing scheme with stretch $s < 5$ has space complexity $\Omega(\sqrt{n})$ bits. On the positive side, [24] tightens the size-stretch tradeoff of [21] by showing that, for every $k \geq 2$, there exists a routing scheme with stretch $4k-5$ and space complexity $\widetilde{O}(n^{1/k})$ bits. (The stretch can be reduced to $2k-1$ using handshaking). Recently, [8] showed that, for $k \geq 4$, a stretch $s = \alpha\, k$ with $\alpha < 4$ can be achieved using routing tables of size $\widetilde{O}(n^{1/k})$.

[2] The handshaking mechanism is similar to DNS lookup in TCP/IP. It allows querying some node(s) for getting additional information about the route to the target.

[3] The space complexity can be reduced to $O(\log n)$ if the designer of the routing scheme is also allowed to assign the port numbers to the nodes.

The distinction between name-independent routing schemes, and routing schemes assigning specific names to the nodes was first made in [4]. Then, [5] presented techniques for designing name-independent routing schemes with constant stretch and space complexity $o(n)$ bits. Almost 15 years after, [3] described a name-independent routing scheme with stretch 5 and space complexity $\widetilde{O}(\sqrt{n})$ bits. This was further improved in [2] thanks to a name-independent routing scheme with stretch 3 and space complexity $\widetilde{O}(\sqrt{n})$ bits. A couple of years later, [1] showed that there are tradeoffs between stretch and space complexity for name-independent routing schemes as well. Specifically, [1] showed that, for any $k \geq 1$, there exists a name-independent routing scheme with space complexity $\widetilde{O}(n^{1/k})$ bits and stretch $O(k)$.

The certification mechanism used in this paper is based on the notion of proof-labeling scheme introduced in [20] in which an oracle, called *prover*, assigns certificates to the nodes, and a distributed algorithm, called *verifier*, checks that this certificates collectively form a proof that the global state of the network is legal with respect to a given boolean network predicate. Proof-labeling schemes have been widely used in literature. For example, [23] uses them to verify spanning trees in networks. This result has been extended in [14], where proof-labeling schemes are used to verify spanning trees in evolving networks that are evolving with time. Variants of proof-labeling schemes have been considered in, e.g., [7, 13], and [19]. More generally, see [10] for a survey of distributed decision.

2 Definitions

Routing Schemes. Let $\mathcal{F}$ be a family of edge-weighted graphs with edges labeled at each node by distinct port numbers from 1 to the degree of the node. The weights are all positive, and the weight of edge e represents its length. It is thus denoted by $\mathrm{length}(e)$. The nodes are given distinct identities. All node-identities and edge-weights are supposed to be stored on $O(\log n)$ bits. For the sake of simplifying notations, we do not make a distinction between a node v and its identity, also denoted by v. Given two nodes u, v, we denote by $\delta(u, v)$ the weighted distance between u and v. Given an edge e of end-point u, the port number of e at u is denoted by $\mathsf{port}_u(e)$.

We follow the usual setting of proof-labeling scheme [20] in which the values of the edge-weights, node identities, and port numbers are fixed, and cannot be corrupted. Only the internal memories of the nodes, storing information about, say, routing, are susceptible to be corrupted.

A *routing scheme* for $\mathcal{F}$ is a mechanism assigning a *name*, $\mathsf{name}(u)$, and a routing *table*, $\mathsf{table}(u)$, to every node u of every graph $G \in \mathcal{F}$ such that, for any pair (s, t) of nodes of any $G \in \mathcal{F}$, there exists a path $u_0, u_1, \ldots, u_k$ from s to t in G with $u_0 = s$, $u_k = t$, and

$$\mathsf{table}(u_i)\big(\mathsf{name}(t)\big) = \mathsf{port}_{u_i}(\{u_i, u_{i+1}\}) \tag{1}$$

for every $i = 0, \ldots, k-1$. That is, every intermediate node u_i, $0 \leq i < k$, can determine on which of its ports the message has to be forwarded, based solely on its routing table, and on the name of the target. In Eq. 1, each table is viewed as a function taking names as arguments, and returning port numbers. The path $u_0, u_1, \ldots, u_k$ is then called the *route* generated by the scheme from s to t. It is worth pointing out the following observations.

Name-independent routing schemes are restricted to use names that are fixed a priori, that is, the name of a node is its identity, i.e., $\mathsf{name}(v) = v$ for every node v. Instead, name-dependent routing schemes allow names to be set for facilitating routing, and names are typically just bounded to be storable on a polylogarithmic number of bits.

The *header* of a message is the part of that message containing all information enabling its routing throughout the network. The header of a message with destination t is typically $\mathsf{name}(t)$. However, some routing schemes ask for message headers that can be modified. This holds for both name-dependent schemes (like, e.g., [24]) and name-independent schemes (like, e.g., [2]). The typical scenario requiring modifiable headers is when a message is routed from source s to target t as follows. From $\mathsf{name}(t)$ and $\mathsf{table}(s)$, node s can derive the existence of some node v containing additional information about how to reach t. Then the message is first routed from s to v, and then from v to t. Distinguishing these two distinct parts of the routes from s to t often requires to use different headers. In case of modifiable headers, Eq. (1) should be tuned accordingly as the argument of routing is not necessarily just a name, but a header.

Some routing schemes may use a mechanism called *handshaking* [24], which is an abstraction of mechanisms such as Domain Name System (DNS) enabling to recover an IP address from its domain name. Let us consider the aforementioned scenario where a routing scheme routes a message from s to t via an intermediate node v identified by s from $\mathsf{name}(t)$ and $\mathsf{table}(s)$. One can then enhance the routing scheme by a handshaking mechanism, enabling s to query v directly, and to recover the information stored at v about t. Then s can route the message directly from s to t, avoiding the detour to v. That is, handshaking enables to distinguish the part of the routing used to get information about the target (like in DNS), from the part of routing used to transfer messages (like in IP). Handshaking is used in [24] to reduce the stretch of routing schemes with space complexity $\widetilde{O}(n^{1/k})$ bits from $4k-5$ to $2k-1$, for every $k>2$.

The *size* of a routing scheme is the maximum, taken over all nodes u of all graphs in $\mathcal{F}$, of the memory space required to encode the function $\mathsf{table}(u)$ at node u. The *stretch* of a routing scheme is the maximum, taken over all pairs (s,t) of nodes in all graphs $G\in\mathcal{F}$, of the ratio of the length of the route from s to t (i.e., the sum of the edge weights along that route) with the weighted distance between s and t in G.

Distributed Verification. Given a graph G, a *certificate function* for G is a function $\mathsf{certificate}: V(G)\to\{0,1\}^*$ assigning a certificate, $\mathsf{certificate}(u)$, to every node $u\in V(G)$. A *verification* algorithm is a distributed algorithm running concurrently at all nodes in parallel. At every node $u\in V(G)$ of every graph $G\in\mathcal{F}$, the algorithm takes as input the identity of node u, the certificate $\mathsf{certificate}(u)$ and routing table $\mathsf{table}(u)$ assigned to node u, as well as the collection of pairs $(\mathsf{table}(v),\mathsf{certificate}(v))$ assigned to all neighbors v of u, with their identities, and outputs *accept* or *reject*.

▶ **Definition 1.** A routing scheme for $\mathcal{F}$ is *verifiable* if there exists a verification algorithm VERIF such that, for every $G\in\mathcal{F}$,

- if the tables given to all nodes of G are the ones specified by the routing scheme, then there exists a certificate function for G such that the verification algorithm VERIF outputs *accept* at all nodes;
- if some tables given to some nodes of G differ from the ones specified by the routing scheme, then, for every certificate function for G, the verification algorithm VERIF outputs reject in at least one node.

The second bullet guarantees that if an adversary modifies some routing tables, or even just a single bit of a single table, then there are no ways it can also modify some, or even all certificates so that to force all nodes to accept: at least one node will detect the change. Of course, this node does not need to be the same for different modifications of the routing tables, or for different certificates.

▶ Remark. The above definition is the classical definition of proof-labeling scheme applied to verifying routing schemes. In particular, it may well be the case that a correct labeling scheme be rejected if the certificates have not been set appropriately, just like a spanning tree T will be rejected by a proof-labeling scheme for spanning trees if the certificates have been set for another spanning tree $T' \neq T$.

3 Name-dependent routing scheme

In this section, we show how to verify the stretch-3 routing scheme by Thorup and Zwick in [24]. This scheme uses tables of $\widetilde{O}(\sqrt{n})$ bits of memory at each node. We show the following:

▶ **Theorem 2.** *The stretch-3 routing scheme by Thorup and Zwick in [24] can be locally verified using certificates of size $\widetilde{O}(\sqrt{n})$ bits.*

Before proving the theorem, let us recall the structure of the routing scheme in [24]. It assigns names and tables to nodes of every $G = (V, E)$ as follows.

Landmarks, Bunch and Clusters. The routing scheme in [24] uses the notion of *landmarks* (a.k.a. *centers*). These landmarks form a subset $L \subseteq V$ of nodes. For $v \in V$, let l_v denote the landmark closest to v in G. For every $v \in V$, the *bunch* of v with respect to the set L is defined as follows: $\mathsf{bunch}(v) = \{u \in V : \delta(v,u) < \delta(l_v, v)\}$. The routing scheme in [24] also uses the notion of *cluster*. For every node $v \in V$, $\mathsf{cluster}(v) = \{u \in V : \delta(v,u) < \delta(l_u, u)\}$. As a consequence, for every $u, v \in V$, we have: $u \in \mathsf{cluster}(v) \iff v \in \mathsf{bunch}(u)$. Note that since, for every $v \in V$, and every $l \in L$, we have $l \notin \mathsf{bunch}(v)$, it follows that $\mathsf{cluster}(l) = \emptyset$ for every $l \in L$. By construction of the bunches and the clusters, it also holds that, for every $v \in V$, $\mathsf{cluster}(v) \cap L = \emptyset$. Also, clusters satisfy the following property.

▶ **Lemma 3** ([24])**.** *If $u \in \mathsf{cluster}(v)$ then, for every node w on a shortest path between u and v, we have $u \in \mathsf{cluster}(w)$.*

In [24], the landmarks are chosen by an algorithm that samples them uniformly at random in V until the following holds: for every node v, $|\mathsf{cluster}(v)| < 4\sqrt{n}$. It is proved that this algorithm returns, w.h.p., a set of landmarks of size at most $2\log(n)\sqrt{n}$.

Names and tables. For every two nodes v and t, let $\mathsf{next}(v,t)$ be the port number of an edge incident to v on a shortest path between v and t. Each node $t \in V$ is assigned a $3\lceil\log(n)\rceil$-bit name as follows: $\mathsf{name}(t) = (t, l_t, \mathsf{next}(l_t, t))$. Each node $v \in V$ then stores the following $\widetilde{O}(\sqrt{n})$-bit information in its routing table, $\mathsf{table}(v)$:

- the identities of all the landmarks $l \in L$;
- the identities of all nodes $t \in \mathsf{cluster}(v)$;
- the set $\{\mathsf{next}(v,t) : t \in L \cup \mathsf{cluster}(v)\}$.

Routing. Note that, by Lemma 3, any message from a node v to a node in $\mathsf{cluster}(v)$ reaches its target along a shortest path. The same holds for landmarks since every node is given information about how to reach every landmark. In general, let us assume that v wants to send a message to some node t with label $(t, l_t, \mathsf{next}(l_t, t))$ that is neither a landmark nor belongs to $\mathsf{cluster}(v)$. In this case, v extracts the landmark l_t nearest to t from t's name (note here the impact of allowing the scheme to assign specific names to nodes), and forwards the message through the port on a shortest path towards l_t using the information v has in its table. Upon reception of the message, l_t forwards the message towards t on a shortest

path using $\mathsf{next}(l_t, t)$ (this information can also be extracted from the name of t), to reach a node $z \in \mathsf{bunch}(t)$. At last, since $z \in \mathsf{bunch}(t)$, we have $t \in \mathsf{cluster}(z)$, which means that z can route to t via a shortest path using the information available in its table. By Lemma 3, this also holds for every node along a shortest path between z and t. Using symmetry, and triangle inequality, [24] shows that this routing scheme guarantees stretch 3.

Proof of Theorem 2. To enable local verification of the stretch-3 routing scheme in [24], a certificate of size $\widetilde{O}(\sqrt{n})$ bits is given to each node. Let $G = (V, E)$ be an undirected graph with positive weights assigned to its edges, and a correct assignment of the routing tables to the nodes according to the specifications of [24] as summarized before. Then each node $v \in V$ is assigned a certificate composed of:

- the distance between v and every landmark in L;
- the distance between v and every node in $\mathsf{cluster}(v)$;
- the set $\{\delta(t, l_t) : t \in \mathsf{cluster}(v)\}$.

As claimed, all these information can be stored using $\widetilde{O}(\sqrt{n})$ bits of memory.

We assume, without loss of generality, that all nodes know n (verifying the value of n is easy using a proof-labeling scheme with $O(\log n)$-bit certificates [20]). The verification of the routing scheme then proceeds as follows. We describe the verification algorithm VERIF running at node $v \in V$. This verification goes in a sequence of steps. At each step, either v outputs *reject* and stops, or it goes to the next step.

We denote by $L(v)$, $C(v)$, and $\{N(v, t) : t \in L(v) \cup C(v)\}$ the content of the routing table of v. These entries are supposed to be the set of landmarks, the cluster of v, and the set of next-pointers given to v, respectively. We also denote by d the distance given in the certificates. That is, node v is given a set $\{d(v, t) : t \in L(v) \cup C(v)\}$ and a set $\{d(t, l_t) : t \in C(v)\}$ where l_t is supposed to be the node in $L(v)$ closest to $t \in C(v)$. Of course, if a node does not have a table and a certificate of these forms, then it outputs *reject*. So, we assume that all tables and certificates have the above formats. The algorithm proceeds as follows at every node v. Node v checks that

1. the information in its table satisfy $|C(v)| \leq 4\sqrt{n}$ and $|L(v)| \leq 2 \log n \sqrt{n}$ bits;
2. it has the same set of landmarks as its neighbors;
3. for every $l \in L(v)$ with $l \neq v$, there exists a neighbor u of v satisfying $d(v, l) = \text{length}(\{v, u\}) + d(u, l)$, and all other neighbors satisfy $d(v, l) \leq \text{length}(\{v, u\}) + d(u, l)$, with $N(v, l)$ pointing to a neighbor u satisfying $d(v, l) = \text{length}(\{v, u\}) + d(u, l)$;
4. if $v \in L(v)$ then $C(v) = \emptyset$;
5. if $v \notin L(v)$ then $v \in C(v)$ and, for every node $t \in C(v)$ with $t \neq v$, there exists a neighbor u of v satisfying $t \in C(u)$ with $d(v, t) = \text{length}(\{v, u\}) + d(u, t)$, and every neighbor u of v with $t \in C(u)$ satisfies $d(v, t) \leq \text{length}(\{v, u\}) + d(u, t)$, with $N(v, t)$ pointing to a neighbor u satisfying $t \in C(u)$ and $d(v, t) = \text{length}(\{v, u\}) + d(u, t)$;
6. for every node $t \in C(v)$, for every neighbor u of v satisfying $t \in C(u)$, the distance $d(t, l_t)$ in the certificate of u is equal to the distance $d(t, l_t)$ in the certificate of v;
7. for every $t \in C(v)$, it holds that $d(v, t) < d(t, l_t)$;
8. for every neighbor u, and every $t \in \mathsf{cluster}(u) \setminus \mathsf{cluster}(v)$, it holds that $\text{length}\{(v, u)\} + d(u, t) \geq d(t, l_t)$.

If v passes all the above tests, then v outputs *accept*, else it outputs *reject*.

We now establish the correctness of this local verification algorithm. First, by construction, if all tables are set according to [24], that is, if, for every node v, $L(v) = L$, $C(v) = \mathsf{cluster}(v)$ and $N(v, t) = \mathsf{next}(v, t)$ for all $t \in L \cup \mathsf{cluster}(v)$, then every node running the verification

algorithm with the appropriate certificate $\{\delta(v,t) : t \in L \cup \mathsf{cluster}(v)\} \cup \{\delta(t,l_t) : t \in \mathsf{cluster}(v)\}$ where l_t is the node in L closest to $t \in \mathsf{cluster}(v)$, will face no inconsistencies with its neighbors, i.e., all the above tests are passed, leading every node to accept, as desired.

So, it remains to show that if some tables are not what they should be according to [24], then, no matter the certificates assigned to the nodes, at least one node will fail one of the tests. If all nodes output *accept*, then, by Step 1, all routing tables are of the appropriate size. Also, by Step 2, the set L of landmarks given to the nodes is the same for all nodes, as otherwise there will be two neighbors that would have different sets. Moreover, by Step 3, we get that, at every node v, the distances of this node to the landmarks, as stored in its certificate, are correct, from which we infer that $N(v,l)$ is appropriately set in the table of v, that is $N(v,l) = \mathsf{next}(v,l)$ for every $l \in L$. Hence, if all the tests in Steps 1-3 are passed, all the data referring to L in both the tables and the certificates are consistent. In particular, every node v knows the landmark l_v which is closest to it, and its distance $\delta(v,l_v)$.

We now show that if all these tests as well as the remaining tests are passed, then the clusters in the tables are correct, w.r.t. L, as well as the next-pointers. We first show that, if all tests are passed, then, for every node $v \in V$, $C(v) \subseteq \mathsf{cluster}(v)$. By Step 4, this latter equality holds for every landmark v. By Step 5, we get that, at every node v, the distance of this node to every node $t \in C(u)$, as stored in its certificate, are correct, from which we infer that $N(v,t)$ is appropriately set in the table of v, that is $N(v,t) = \mathsf{next}(v,t)$ for every $t \in C(v)$. By Step 6, we get that, for every $t \in C(v)$, we do have $d(l_t,t) = \delta(l_t,t)$. Indeed, this equality will be checked by all nodes on a shortest path between v and t (whose existence is guaranteed by Step 5), and t has the right distance $\delta(t,l_t)$ in its certificate by Step 3. Recall that $\mathsf{cluster}(v) = \{t \in V \mid \delta(v,t) < \delta(l_t,t)\}$ where l_t is the landmark closest to t. Step 7 precisely checks that inequality.

It remains to show that there are no nodes in $\mathsf{cluster}(v)$ that are not in $C(v)$. Assume that there exists $t \in \mathsf{cluster}(v) \setminus C(v)$, and let P be a shortest path between v and t. Let v' be the closest node to v on P such that $t \in C(v') \subseteq \mathsf{cluster}(v')$. Note that such a node v' exists as $t \in C(t)$. Let v'' be the node just before v' on P traversed from v to t. By Lemma 3, since $t \in \mathsf{cluster}(v)$, we also have $t \in \mathsf{cluster}(v'')$. We have $t \in \mathsf{cluster}(v'') \setminus C(v'')$. Therefore $\delta(v'',t) < \delta(l_t,t)$. Now, $\delta(v'',t) = \mathrm{length}(\{(v'',v')\}) + \delta(v',t)$ because P is a shortest path between v'' and t passing through v'. So, $\mathrm{length}(\{(v'',v')\}) + \delta(v',t) < \delta(l_t,t)$. Step 8 guarantees that it is not the case. Therefore, there are no nodes in $\mathsf{cluster}(v) \setminus C(v)$. It follows that $\mathsf{cluster}(v) = C(v)$ for all nodes v. This completes the proof of Theorem 2. ◀

4 Name-independent Routing Scheme

The purpose of this section is twofold. First, it serves as recalling basic notions that will be helpful for the design of our new name-independent routing scheme. Second, it is used to show why the known name-independent routing scheme in [2] appears to be difficult, and perhaps even impossible to verify locally.

The Routing Scheme of [2]

The stretch-3 name-independent routing scheme of [2] uses $\widetilde{O}(\sqrt{n})$ space at each node. We provide a high level description of that scheme. Recall that, in name-independent routing, a target node is referred only by its identity. That is, $\mathsf{name}(t)$ is the identity of t, i.e., $\mathsf{name}(t) = t$. Let $G = (V,E)$. For every node $v \in V$, the *vicinity ball* of v, denoted by $\mathsf{ball}(v)$, is the set of the $4\lceil \alpha \log(n) \sqrt{n} \rceil$ closest nodes to v, for a large enough constant $\alpha > 0$, where ties are broken using the order of node identities. By this definition, if $u \in \mathsf{ball}(v)$ and w is on a shortest path between v and u, then $u \in \mathsf{ball}(w)$.

Color-Sets. In [2], the nodes are partitioned into sets $C_1, \ldots C_{\sqrt{n}}$, called *color-sets*, and, for $i = 1, \ldots, \sqrt{n}$, the nodes in color-set C_i are assigned the same color i. For every node $v \in V$, its vicinity ball, $\mathsf{ball}(v)$, contains at least one node from each color-set. To get this, the color of a given node v is determined by a hash function *color* which, given the identity of a node, maps that identity to a color in $\{1, \ldots, \sqrt{n}\}$. This mapping from identities to colors is balanced in the sense that at most $O(\log n\sqrt{n})$ nodes map to the same color. A color is chosen arbitrarily, and all nodes with that color are considered to be the landmarks. Let L be the set of landmarks. It holds that $|L| = O(\log n\sqrt{n})$. Also, each node $v \in V$ has at least one landmark in its vicinity ball. We fix, for each vicinity ball $\mathsf{ball}(v)$, an arbitrary landmark, denoted by l_v.

Routing in trees. This construction in [2] makes use of routing schemes in trees. More precisely, they use the results from [11, 24], which states that there exists a shortest-path (name-dependent) routing scheme for trees using names and tables both on $O(\log^2(n)/\log\log(n))$ bits in n-node trees. For a tree T containing node v, let $\mathsf{table}_T(v)$ and $\mathsf{name}_T(v)$ denote the routing table of node v in T, and the name of v in T, as assigned by the scheme in [11].

The routing tables. For any node v, let $T(v)$ be a shortest-path spanning tree rooted at v. Let $P(v, w, u)$ be a path between v and u composed of a shortest path between v and w and a shortest path between w and u. Such a path is said to be *good* for (v, u) if $v \in \mathsf{ball}(w)$, and there exists an edge $\{x, y\}$ along a shortest path between w and u with $x \in \mathsf{ball}(w)$ and $y \in \mathsf{ball}(u)$; Every node $v \in V$ stores the following information in its routing table $\mathsf{table}(v)$:

- the hash function *color* that maps identities in colors;
- the identity of every node $u \in \mathsf{ball}(v)$, and the port number $\mathsf{next}(v, u)$;
- for every landmark $l \in L$, the routing table $\mathsf{table}_{T(l)}(v)$ for routing in $T(l)$;
- for every node $u \in \mathsf{ball}(v)$, the routing table $\mathsf{table}_{T(u)}(v)$ for routing in $T(u)$;
- the identities of all nodes with same color as v, and, for each such node u, the following additional information:

  Rule 1: if there are no good paths $P(v, w, u)$, v stores $\big(\mathsf{name}_{T(l_u)}(l_u), \mathsf{name}_{T(l_u)}(u)\big)$.

  Rule 2: if there exists a good path $P(v, w, u)$, then let us pick a good path P of minimum length among all good paths; then, let us compare its length $|P|$ with the length $|Q|$ of the path Q composed of a shortest path between v and l_u, and a shortest path between l_u and u; provide v with $\big(\mathsf{name}_{T(l_u)}(l_u), \mathsf{name}_{T(l_u)}(u)\big)$ if $|Q| \leq |P|$, and with $\big(\mathsf{name}_{T(v)}(w), x, \mathsf{port}_x(\{x, y\}), \mathsf{name}_{T(y)}(u)\big)$ otherwise.

Storing the hash function *color* requires $O(\sqrt{n})$ bits. L and $\mathsf{ball}(v)$ are both of size $\widetilde{O}(\sqrt{n})$ bits. Moreover, the number of nodes with identical color is $\widetilde{O}(\sqrt{n})$, for every color. Finally, shortest-path routing in any tree can be achieved using tables and names of size $O(\log^2 n/\log\log n)$ bits [11, 24]. It follows that $|\mathsf{table}(v)| = \widetilde{O}(\sqrt{n})$, as desired.

Routing. Routing from a source s to a target t is achieved in the following way (see [2] for more details). If $t \in \mathsf{ball}(s)$, or $t \in L$, or s and t have the same color, then s routes to t using the information available in its table. More specifically, if $t \in \mathsf{ball}(s)$ or $t \in L$, then s sets the header of the message as just the identity of the target t. Instead, if s and t have the same color but $t \notin \mathsf{ball}(s) \cup L$, the source s sets the header as one of the two possible rules 1 or 2. Otherwise, that is, if $t \notin \mathsf{ball}(s) \cup L$ and $\mathit{color}(s) \neq \mathit{color}(t)$, node s routes the message towards some node $w \in \mathsf{ball}(s)$ sharing the same color as t. (The color of t can be obtained by hashing the identity of t). The header is set to t, and w will change the header upon reception of this message, according to the rules previously specified. It is shown in [2] that this routing guarantees a stretch 3.

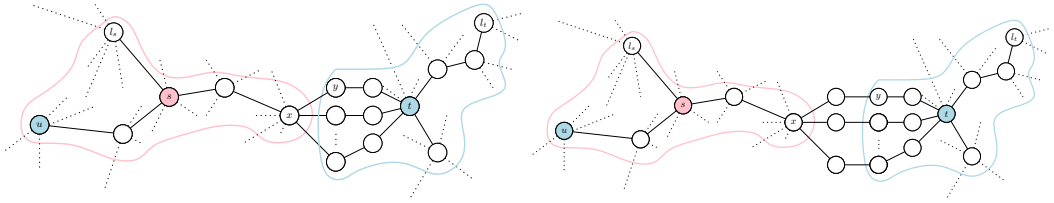


Figure 1 (Left) All the nodes in a shortest path between s and t belong to $\mathsf{ball}(s) \cup \mathsf{ball}(t)$. (Right) There are nodes in a shortest path between s and t that do not belong to $\mathsf{ball}(s) \cup \mathsf{ball}(t)$.

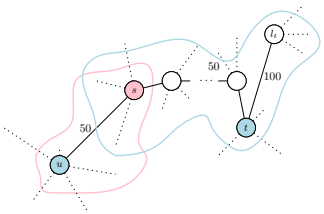


Figure 2 A route of stretch 7.

On the difficulty of locally verifying the scheme in [2]

We note that the routing scheme in [2], as sketched in the previous subsection, has some "*global*" features that makes it plausibly difficult to locally verify, by adding certificates of size $\widetilde{O}(\sqrt{n})$ bits at each node. In this subsection, we mention one of these global features, illustrated in the example depicted on Fig. 1. We are considering routing from a source $s \notin L$ to a target $t \notin L$, of different colors, with $s \notin \mathsf{ball}(t)$ and $t \notin \mathsf{ball}(s)$. Let us assume that node s has color red, while node t has color blue. According to the routing scheme in [2], node s first routes the message towards some blue node $u \in \mathsf{ball}(s)$ to get information about the blue target t. In the example of Fig. 1(left), node u stores $\big(\mathsf{name}_{T(u)}(s), x, port(x,y), \mathsf{name}_{T(y)}(t)\big)$, to guarantee a stretch 3. Instead, in the example of Fig. 1(right), node u stores $\big(\mathsf{name}_{T(l_t)}(l_t), \mathsf{name}_{T(l_t)}(t)\big)$. to guarantee such a small stretch. Verifying locally whether there exists a good path between s and t, which is the condition leading to distinguishing the case where the content of rule 1 must apply, from the case where the content of rule 2 must apply, appears to be a very difficult matter when restricted to certificates of size $\widetilde{O}(\sqrt{n})$.

We point out that systematically applying rule 1 would result in a routing scheme that may be locally verifiable. However, its stretch is at least 7. To see why, let us consider the example displayed in Figure 2 where $s \in \mathsf{ball}(t)$ but $t \notin \mathsf{ball}(s)$. The radius of $\mathsf{ball}(s)$ is 50. Let $u \in \mathsf{ball}(s)$ be one of the farthest nodes to s, i.e., $\delta(s,u) = 50$. The radius of $\mathsf{ball}(t)$ is 100 and $\delta(t, l_t) = 100$. Although $\delta(u,t) = 100$, we assume $u \notin \mathsf{ball}(t)$ due to lexicographical order priorities. Finally, assume that $\delta(s,t) = 50$. The worst case route from s to t would then be $s \rightsquigarrow u \rightsquigarrow s \rightsquigarrow t \rightsquigarrow l_t \rightsquigarrow t$. This path is of length $\delta(s,u) + \delta(u,s) + \delta(s,t) + \delta(t,l_t) + \delta(l_t,t) = 2 \times 50 + 50 + 2 \times 100 = 350$ which is $7\delta(s,t)$.

5 A New Name-Independent Routing Scheme

In this section, we describe and analyze a new name-independent routing scheme, denoted by $\mathcal{R}$. The section is entirely dedicated to the proof of the following theorem. Recall that, given a family of events $(\mathcal{E}_n)_{n\geq 1}$, event $\mathcal{E}_n$ holds with high probability if $\Pr[\mathcal{E}_n] = 1 - O(1/n^c)$ for some $c \geq 1$.

▶ **Theorem 4.** *The name-independent routing scheme $\mathcal{R}$ uses routing tables of size $\widetilde{O}(\sqrt{n})$ bits at each node. It guarantees stretch 5, and, using handshaking, its stretch can be reduced to 3. In both cases, $\mathcal{R}$ can be locally verified using certificates of size $\widetilde{O}(\sqrt{n})$ bits, using a 1-sided error verification scheme which guarantees that incorrect tables are detected with high probability.*

Our verifier used to establish Theorem 4 is actually deterministic. The certificates however store hash functions chosen at random. We assume that these hash functions are not corruptible, and that the adversary is not capable to create collisions (if not accidentally, by chance). Even though this assumption may seem strong, we point out that this is relevant to practical situations in which cryptographic hash functions designed to be collision resistant and hard to invert are used (for more details please refere to Chapter 5 of [9]).

The new routing scheme $\mathcal{R}$. Our new routing scheme $\mathcal{R}$ borrows ingredients from both [2] and [24]. In particular, the landmarks are chosen as for the (name-dependent) routing scheme in [24], i.e., in such a way that every node v has a cluster $\mathsf{cluster}(v)$ of size at most $4\sqrt{n}$, and the landmarks form a set L of size at most $2\log n\sqrt{n}$. However, we reinforce the way the nearest landmark to v is selected, by picking the nearest landmark l_v such that $l_v = \text{argmin}_{l\in L}\delta(v,l)$ where ties are broken by choosing the landmark with smallest identity. Also, we slightly reinforce the definition of next: For every two nodes v and t, we set $\mathsf{next}(v,t)$ as the *smallest* port number at v of an edge incident to v on a shortest path between v and t. These two reinforcements of the definitions of landmarks and next guarantee the following.

▶ **Lemma 5.** *Let $v \in V$, and let l_v be its landmark. Let u be a node on a shortest path between v and l_v. Then $l_u = l_v$.*

▶ **Lemma 6.** *Let $v \in V$, let l_v be its landmark, and let w be the neighbor of l_v such that $\mathsf{next}(l_v, v) = \mathsf{port}_{l_v}(\{l_v, w)\}$. Let u be a node on a shortest path between v and w. We have $\mathsf{next}(l_v, u) = \mathsf{next}(l_v, v)$.*

As in [2], each node that is not a landmark is given a color in $\{1,\ldots,\sqrt{n}\}$ determined by a hash function, *color*, that maps identities to colors, where at most $O(\log n\sqrt{n})$ nodes map to the same color. Also, each node v has a vicinity ball, $\mathsf{ball}(v)$, that contains the $O(\log(n)\sqrt{n})$ nodes closest to v (breaking ties using identities). This guarantees that, with high probability, for every node v, there is at least one node of each color in $\mathsf{ball}(v)$.

For each color c, where $1 \le c \le \sqrt{n}$, we define $\mathsf{Dir}_c = \{(v, l_v, \mathsf{next}(l_v, v)) : color(v) = c\}$ which includes the direction to take at l_v for reaching v of color c along a shortest path.

The routing tables. Every node $v \in V$ then stores the following information in $\mathsf{table}(v)$:

- the hash function *color* that maps identities in colors;
- the identity of every landmark $l \in L$, and the corresponding port $\mathsf{next}(v, l)$;
- the identity of every node $u \in \mathsf{cluster}(v)$, and the corresponding port $\mathsf{next}(v, u)$;
- the identity of every node $u \in \mathsf{ball}(v)$, and the corresponding port $\mathsf{next}(v, u)$;
- the set $\mathsf{Dir}_{\mathrm{color}(v)}$.

Storing the hash function *color* requires $O(\sqrt{n})$ bits, as we use the same function as in [2]. Since L, $\mathsf{cluster}(v)$, $\mathsf{ball}(v)$, and $\mathsf{Dir}_{\mathrm{color}(v)}$ are all of size $\widetilde{O}(\sqrt{n})$, we get that $|\mathsf{table}(v)| = \widetilde{O}(\sqrt{n})$ as desired.

Routing. Let us consider routing towards a target node t, and let v be the current node. If $t \in \mathsf{cluster}(v)$, then routing to t is achieved using $\mathsf{next}(v,t)$. Notice that, by Lemma 3, routing to t will actually be achieved along a shortest path. Similarly, if $t \in \mathsf{ball}(v)$, then routing to t is achieved using $\mathsf{next}(v,t)$ along a shortest path, and this also holds if t is a landmark. In general, i.e., if t is neither a landmark nor a node of $\mathsf{cluster}(v) \cup \mathsf{ball}(v)$, then node v computes $\mathit{color}(t)$ by hashing the identity of t.

If $\mathit{color}(t) = \mathit{color}(v)$, then v forwards the message towards l_t using the information in $\mathsf{Dir}_{\mathrm{color}(v)}$, and including $(l_t, \mathsf{next}(l_t,t))$ in the header of the message so that intermediate nodes carry on routing this message to l_t. At l_t, the message will be routed to t using the information $\mathsf{next}(l_t,t)$ available in the header, reaching a node u_t such that $t \in \mathsf{cluster}(u_t)$. At this point, routing proceeds to t along a shortest path.

If $\mathit{color}(t) \neq \mathit{color}(v)$, then the message is forwarded to an arbitrary node $w \in \mathsf{ball}(v)$ having the same color as t (we know that such a node exists), with w in the header of the message. The message then reach w along a shortest path. At w, we have $\mathit{color}(t) = \mathit{color}(w)$, and thus routing proceeds as in the previous case.

Handshaking. The routing with handshaking to node t proceeds as follows. If $t \in L \cup \mathsf{cluster}(v) \cup \mathsf{ball}(v)$, or $\mathit{color}(v) = \mathit{color}(t)$, then routing proceeds as above. Otherwise, v performs a handshake with a node $w \in \mathsf{ball}(v)$ with $\mathit{color}(w) = \mathit{color}(t)$ to get the identity of l_t as well as $\mathsf{next}(l_t,t)$. Then v routes the message to l_t, where it is forwarded to a node u_t such that $t \in \mathsf{cluster}(u_t)$. At this point, routing proceeds to t along a shortest path.

Stretch of the new routing scheme $\mathcal{R}$. Let $s,t \in V$ be two arbitrary nodes of the graph. We show that the routing scheme $\mathcal{R}$ routes messages from s to t along a route of length at most $5\,\delta(s,t)$ in general, and along a route of length at most $3\,\delta(s,t)$ whenever using handshaking. As we already observed, if $t \in L \cup \mathsf{cluster}(v) \cup \mathsf{ball}(v)$, then the message is routed to t along a shortest path, i.e., with stretch 1. Otherwise, we consider separately whether the color of s is the same as the color of t, or not.

Assume first that $\mathit{color}(t) = \mathit{color}(s)$. Then the message is routed towards l_t along a shortest path, then from l_t to t along a shortest path. The length ℓ of this route satisfies $\ell = \delta(s,l_t) + \delta(l_t,t)$. By the triangle inequality, we get that $\ell \leq \delta(s,t) + 2\delta(l_t,t)$. Since $t \notin \mathsf{cluster}(s)$, we get $\delta(s,t) \geq \delta(t,l_t)$. Therefore $\ell \leq 3\,\delta(s,t)$. We are left with the case where $\mathit{color}(s) \neq \mathit{color}(t)$. Observe first that, with handshaking, the route from s to t will be exactly as the one described in the case $\mathit{color}(s) = \mathit{color}(t)$, resulting in a stretch 3. This completes the proof that $\mathcal{R}$ achieves a stretch 3 with handshaking. Without handshaking, the message is forwarded along a shortest path to an arbitrary node $w \in \mathsf{ball}(v)$ having the same color as t, then from w to l_t along a shortest path, and finally from l_t to t along a shortest path. This route is of length $\delta(s,w) + \delta(w,l_t) + \delta(l_t,t) \leq \delta(s,w) + \delta(w,s) + \delta(s,l_t) + \delta(l_t,t) \leq 2\,\delta(s,w) + 3\,\delta(s,t) \leq 5\,\delta(s,t)$, as desired.

Local Verification of $\mathcal{R}$. We show how to verify $\mathcal{R}$ with a verification algorithm VERIF using certificates on $\widetilde{O}(\sqrt{n})$ bits. Let us define the certificates given to nodes when the routing tables are correctly set as specified by $\mathcal{R}$. For each color c, $1 \leq c \leq \sqrt{n}$, let B_c be the number of bits for encoding the set Dir_c, and let $r = \max_{1 \leq c \leq \sqrt{n}} B_c$. We have $r = \widetilde{O}(\sqrt{n})$. Let $f_1, \ldots, f_k$ be $k = \Theta(\log n)$ hash functions, where each one is mapping sequences of at most r bits onto a single bit. More specifically, each function f_i, $1 \leq i \leq k$, is described as a sequence $f_{i,1}, \ldots, f_{i,r}$ of r bits. Given a sequence $D = (d_1, \ldots, d_\ell)$ of $\ell \leq r$ bits, we set $f_i(D) = (\sum_{j=1}^{\ell} f_{i,j}\, d_j) \bmod 2$. Hence, if the r bits describing f_i are chosen independently

uniformly at random, then, for every two ℓ-bit sets D and D', we have [25]: $D \neq D' \Rightarrow \Pr[f_i(D) = f_i(D')] = 1/2$. Therefore, $D \neq D' \Rightarrow \Pr[(f_i(D) = f_i(D),\ i = 1, \ldots, k)] = 1/2^k$. That is, if $k = \beta\lceil \log_2 n \rceil$ with $\beta > 1$, applying the functions $f_1, \ldots, f_k$ to both sets D and D' enables to detect that they are distinct, with probability $1 - 1/n^\beta$.

Each node $v \in V$ stores the certificate composed of the following fields:

- the distances $\{\delta(v, l) : l \in L\}$;
- the distances $\{\delta(v, u) : u \in \mathsf{ball}(v)\}$;
- the set $\big\{\big(\delta(v, u), l_u, \delta(u, l_u)\big) : u \in \mathsf{cluster}(v)\big\}$;
- the set of k hash functions $f_1, \ldots, f_k$;
- the set $\big\{\big(i, c, f_i(\mathsf{Dir}_c)\big) : 1 \leq i \leq k,\ 1 \leq c \leq \sqrt{n}\big\}$.

The first three entries, as well as the last entry, are clearly on $\widetilde{O}(\sqrt{n})$ bits, because of the sizes of L, $\mathsf{cluster}(v)$, and $\mathsf{ball}(v)$. Regarding the fourth entry, each function f_i is described by a sequence of $\widetilde{O}(\sqrt{n})$ random bits.

The verification algorithm VERIF then proceeds as follows. In a way similar to the proof of Theorem 2, we denote by $H(v)$, $L(v)$, $C(v)$, $B(v)$, $D(v)$, and $\{N(v, t) : t \in L(v) \cup C(v) \cup B(v)\}$ the content of the routing table of v. These entries are supposed to be the hash function *color*, the set of landmarks, the cluster of v, the ball of v, the set $\mathsf{Dir}_{\mathrm{color}(v)}$, and the set of next-pointers given to v, respectively. We also denote by d the distance given in the certificates. That is, node v is given a set $\{d(v, t) : t \in L(v) \cup B(v)\}$, and a set $\{(d(v, t), l_t, d(t, l_t)) : t \in C(v)\}$ where l_t is supposed to be the node in $L(v)$ closest to $t \in C(v)$.

We also denote by $F_1^v, \ldots, F_k^v$ the hash functions given to v in its certificate, and by $F(v) = \{F_{i,c}^v : 1 \leq i \leq k,\ 1 \leq c \leq \sqrt{n}\}$ the set of $O(\sqrt{n} \log n)$ hash values in the certificates. Of course, if a node does not have a table and a certificate of these forms, then it outputs *reject*. So, we assume that all tables and certificates have the above formats.

Clusters, balls and landmarks (including distances, ports, sizes, etc.) are checked exactly as in Section 3 for the routing scheme in [24]. So, in particular, ignoring the colors and ignoring the minimality of the landmarks' identities, we can assume that, for every node v, $L(v) = L$, $C(v) = \mathsf{cluster}(v)$ and $B(v) = \mathsf{ball}(v)$, and, for every node $u \in L \cup \mathsf{cluster}(v) \cup \mathsf{ball}(v)$, we have $d(v, u) = \delta(v, u)$, and $N(v, u) = \mathsf{next}(v, u)$ ignoring the minimality of that port number. To check the remaining entries in the routing tables (as well as the previously ignored colors and minimality criteria for the landmarks and the next-pointers), VERIF performs the following sequence of steps. At every step, if the test is not passed at some node, then VERIF outputs *reject* at this node, and stops. Otherwise, it goes to the next step. If all tests are passed at a node, that node outputs *accept*. Node v checks that:

1. l_v has the smallest identity among all landmarks closest to v;
2. it has the same hash function $H(v)$ as its neighbors, that it has one node of each color in $B(v)$, that v appears in $D(v)$, and that all nodes appearing in $D(v)$ have the same color as v;
3. if $l_v \neq v$ then there exists at least one neighbor u on a shortest path between v and l_v with $N(l_v, v) = N(l_v, u)$; for every neighbor u on the shortest path between v and l_v (implying $l_u = l_v$) with $u \neq l_v$, $N(l_v, v) \leq N(l_u, u)$; and if l_v is a neighbor of v on a shortest path between l_v and v, $N(l_v, v) = \mathsf{port}_{l_v}(\{l_v, v\})$;
4. for every $u \in L \cup B(v) \cup C(v)$, the port number $N(v, u)$ is the smallest among all the ports of edges incident to v on a shortest path between v and u;
5. it has the same hash functions $F_1^v, \ldots, F_k^v$, as its neighbors;
6. $F_i^v(D(v)) = F_{i,H(v)}^v$ for every $i = 1, \ldots, k$;
7. the hash value $F_{i,c}^v$, $1 \leq i \leq k$, $1 \leq c \leq \sqrt{n}$, are identical to the ones of their neighbors.

If v passes all the above tests, then v outputs *accept*, else it outputs *reject*.

We now establish the correctness of this local verification algorithm. First, by construction, if all tables are set according to the specification of $\mathcal{R}$, then every node running the verification algorithm with the appropriate certificate will face no inconsistencies with its neighbors, i.e., all the above tests are passed, leading every node to accept, as desired. So, it remains to show that if some tables are not what they should be according to $\mathcal{R}$, then, no matter the certificates assigned to the nodes, at least one node will fail one of the tests with high probability.

If all nodes pass the test of Step 1, then we are guaranteed that not only $L(v) = L$, but also that l_v is indeed the appropriate landmark of v. If all nodes pass the test of Step 2, then it must be the case that $H(v) = \mathit{color}(v)$, that $B(v) = \mathsf{ball}(v)$ with the desired coloring property, and that $D(v) \subseteq \mathsf{Dir}_{\mathrm{color}(v)}$. By Lemma 6, we get that if all nodes pass Steps 3 and 4, then $N(v,u) = \mathsf{next}(v,u)$ where the minimality condition is satisfied. Let us assume that there exists a pair of nodes (u,v) with same color such that $(u, l_u, \mathsf{next}(l_u,u)) \in \mathsf{Dir}_{\mathrm{color}(v)} \setminus D(v)$. By the previous steps, we know that $(u, l_u, \mathsf{next}(l_u,u)) \in D(u)$. Hence, $D(u) \neq D(v)$. On the other hand, if Step 5 is passed by all nodes, then all nodes agree on a set of $k = \Theta(\log n)$ hash function $f_1, \ldots, f_k$. Therefore, assuming that these functions are set at random, we get that, with high probability, there exists at least one function f_i, $1 \leq i \leq k$, such that $f_i(D(v)) \neq f_i(D(u))$. If all nodes pass Step 6, then in particular $F_i^v(D(v)) = F_{i,c}^v$, and $F_i^u(D(u)) = F_{i,c}^u$ where $c = \mathit{color}(v) = \mathit{color}(u)$. We know that $F_{i,c}^v = f_i(D(v))$ and $F_{i,c}^u = f_i(D(u))$, which implies that $F_{i,c}^v \neq F_{i,c}^u$ with high probability, which will be detected at Step 7 by two neighboring nodes in the network. This completes the proof of Theorem 4. ◀

▶ Remark. We have seen that the stretch-3 name-independent routing scheme from [2] does not seem to be locally checkable with certificates of $\widetilde{O}(\sqrt{n})$ bits. Our new name-independent routing scheme $\mathcal{R}$ is locally checkable with certificates of $\widetilde{O}(\sqrt{n})$ bits, but it has stretch 5. We can show that, as for the scheme in [2], the routing scheme in [3] do not appear to be locally checkable with certificates of $\widetilde{O}(\sqrt{n})$ bits. Moreover, handshaking, which may allow that scheme to become locally verifiable with small certificates, does not reduce its stretch (for details, please refer to the full version of this paper [6]).

6 Conclusion and Further Work

We have shown that it is possible to verify routing schemes based on tables of size $\widetilde{O}(\sqrt{n})$ bits using certificates with sizes of the same order of magnitude as the space consumed by the routing tables. The stretch factor is preserved, but to the cost of using handshaking mechanisms for name-independent routing. We do not know whether there exists a stretch-3 name-independent routing scheme, with tables of size $\widetilde{O}(\sqrt{n})$ bits, that can be verified using certificates on $\widetilde{O}(\sqrt{n})$ bits. Our new routing scheme, which is verifiable with certificates on $\widetilde{O}(\sqrt{n})$, has stretch 3 only if using handshaking (otherwise, it has stretch 5). Moreover, the certification of our routing scheme is probabilistic, and it would be of interest to figure out whether deterministic certification exists for some stretch-3 name-independent routing scheme, with tables and certificates on $\widetilde{O}(\sqrt{n})$ bits. It could also be of interest to figure out whether there exists a verification scheme using certificates of size $\widetilde{O}(n^c)$ bits, with $c < \frac{1}{2}$.

Interestingly, our result for stretch-3 name-dependent routing can be extended to larger stretches. Namely, by using the same techniques as for stretch 3, we can show that the family of routing schemes in [24] using, for every $k \geq 1$, tables of size $\widetilde{O}(n^{1/k})$ with stretch at most $4k-5$ (or $2k-1$ using handshaking) are verifiable with certificates of size $\widetilde{O}(n^{1/k})$ (see [6] for more details). However, we do not know whether such a tradeoff between table sizes and

stretches can be established for verifiable *name-independent* routing schemes. Specifically, are the existing families of name-independent routing schemes using, for every $k \geq 1$, tables of size $\widetilde{O}(n^{1/k})$ with stretch at most $O(k)$, verifiable with certificates of size $\widetilde{O}(n^{1/k})$? If not, is it possible to design a new family of verifiable name-independent routing schemes satisfying the same size-stretch tradeoff?

Acknowledgements. The authors are thankful to Cyril Gavoille and Laurent Viennot for discussions about the content of this paper.

References

1. Ittai Abraham, Cyril Gavoille, and Dahlia Malkhi. On space-stretch trade-offs: upper bounds. In *SPAA 2006: Proceedings of the 18th Annual ACM Symposium on Parallelism in Algorithms and Architectures, Cambridge, Massachusetts, USA, July 30 - August 2, 2006*, pages 217–224, 2006.
2. Ittai Abraham, Cyril Gavoille, Dahlia Malkhi, Noam Nisan, and Mikkel Thorup. Compact name-independent routing with minimum stretch. In *SPAA 2004: Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, June 27-30, 2004, Barcelona, Spain*, pages 20–24, 2004.
3. Marta Arias, Lenore Cowen, Kofi A. Laing, Rajmohan Rajaraman, and Orjeta Taka. Compact routing with name independence. In *SPAA 2003: Proceedings of the Fifteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, June 7-9, 2003, San Diego, California, USA (part of FCRC 2003)*, pages 184–192, 2003.
4. Baruch Awerbuch, Amotz Bar-Noy, Nathan Linial, and David Peleg. Compact distributed data structures for adaptive routing (extended abstract). In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washigton, USA*, pages 479–489, 1989.
5. Baruch Awerbuch and David Peleg. Sparse partitions (extended abstract). In *31st Annual Symposium on Foundations of Computer Science, St. Louis, Missouri, USA, October 22-24, 1990, Volume II*, pages 503–513, 1990.
6. Alkida Balliu and Pierre Fraigniaud. Certification of compact low-stretch routing schemes. In *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, 2017.
7. Mor Baruch, Pierre Fraigniaud, and Boaz Patt-Shamir. Randomized proof-labeling schemes. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*, pages 315–324, 2015.
8. Shiri Chechik. Compact routing schemes with improved stretch. In *ACM Symposium on Principles of Distributed Computing, PODC'13, Montreal, QC, Canada, July 22-24, 2013*, pages 33–41, 2013.
9. Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. *Cryptography Engineering: Design Principles and Practical Applications*. Wiley Publishing, 2010.
10. Laurent Feuilloley and Pierre Fraigniaud. Survey of distributed decision. *Bulletin of the EATCS*, 119, 2016.
11. Pierre Fraigniaud and Cyril Gavoille. Routing in trees. In *Automata, Languages and Programming, 28th International Colloquium, ICALP 2001, Crete, Greece, July 8-12, 2001, Proceedings*, pages 757–772, 2001.
12. Pierre Fraigniaud and Cyril Gavoille. A space lower bound for routing in trees. In *STACS 2002, 19th Annual Symposium on Theoretical Aspects of Computer Science, Antibes - Juan les Pins, France, March 14-16, 2002, Proceedings*, pages 65–75, 2002.

13 Pierre Fraigniaud, Amos Korman, and David Peleg. Towards a complexity theory for local distributed computing. *J. ACM*, 60(5):35, 2013.

14 Klaus-Tycho Förster, Thomas Lüdi, Jochen Seidel, and Roger Wattenhofer. Local Checkability, No Strings Attached: (A)cyclicity, Reachability, Loop Free Updates in SDNs. In *Theoretical Computer Science (TCS)*, November 2016.

15 Cyril Gavoille. Routing in distributed networks: overview and open problems. *SIGACT News*, 32(1):36–52, 2001.

16 Cyril Gavoille and Marc Gengler. Space-efficiency for routing schemes of stretch factor three. *J. Parallel Distrib. Comput.*, 61(5):679–687, 2001.

17 Cyril Gavoille and David Peleg. Compact and localized distributed data structures. *Distributed Computing*, 16(2-3):111–120, 2003.

18 Cyril Gavoille and Stephane Perennes. Memory requirements for routing in distributed networks (extended abstract). In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, USA, May 23-26, 1996*, pages 125–133, 1996.

19 Mika Göös and Jukka Suomela. Locally checkable proofs in distributed computing. *Theory of Computing*, 12(1):1–33, 2016.

20 Amos Korman, Shay Kutten, and David Peleg. Proof labeling schemes. *Distributed Computing*, 22(4):215–233, 2010.

21 David Peleg and Eli Upfal. A tradeoff between space and efficiency for routing tables (extended abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 43–52, 1988.

22 Nicola Santoro and Ramez Khatib. Labelling and implicit routing in networks. *Comput. J.*, 28(1):5–8, 1985.

23 Stefan Schmid and Jukka Suomela. Exploiting locality in distributed SDN control. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, HotSDN 2013, The Chinese University of Hong Kong, Hong Kong, China, Friday, August 16, 2013*, pages 121–126, 2013.

24 Mikkel Thorup and Uri Zwick. Compact routing schemes. In *SPAA*, pages 1–10, 2001.

25 Andrew Chi-Chih Yao. Some complexity questions related to distributive computing (preliminary report). In *Proceedings of the 11h Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1979, Atlanta, Georgia, USA*, pages 209–213, 1979.

Near-Optimal Approximate Shortest Paths and Transshipment in Distributed and Streaming Models*

Ruben Becker[1], Andreas Karrenbauer[2], Sebastian Krinninger[3], and Christoph Lenzen[4]

1 **MPI for Informatics, Saarland Informatics Campus, Saarbrücken, Germany**
ruben.becker@mpi-inf.mpg.de
2 **MPI for Informatics, Saarland Informatics Campus, Saarbrücken, Germany**
andreas.karrenbauer@mpi-inf.mpg.de
3 **Faculty of Computer Science, University of Vienna, Austria**
sebastian.krinninger@univie.ac.at
4 **MPI for Informatics, Saarland Informatics Campus, Saarbrücken, Germany**
christoph.lenzen@mpi-inf.mpg.de

Abstract

We present a method for solving the *shortest transshipment* problem—also known as uncapacitated minimum cost flow—up to a multiplicative error of $1+\varepsilon$ in undirected graphs with non-negative integer edge weights using a tailored gradient descent algorithm. Our gradient descent algorithm takes $\varepsilon^{-3}\operatorname{polylog} n$ iterations, and in each iteration it needs to solve an instance of the transshipment problem up to a multiplicative error of $\operatorname{polylog} n$, where n is the number of nodes. In particular, this allows us to perform a single iteration by computing a solution on a sparse spanner of logarithmic stretch. Using a careful white-box analysis, we can further extend the method to finding approximate solutions for the single-source shortest paths (SSSP) problem. As a consequence, we improve prior work by obtaining the following results:

1. *Broadcast CONGEST model:* $(1+\varepsilon)$-approximate SSSP using $\tilde{O}((\sqrt{n}+D)\cdot\varepsilon^{-O(1)})$ rounds,[1] where D is the (hop) diameter of the network.
2. *Broadcast congested clique model:* $(1+\varepsilon)$-approximate shortest transshipment and SSSP using $\tilde{O}(\varepsilon^{-O(1)})$ rounds.
3. *Multipass streaming model:* $(1+\varepsilon)$-approximate shortest transshipment and SSSP using $\tilde{O}(n)$ space and $\tilde{O}(\varepsilon^{-O(1)})$ passes.

The previously fastest SSSP algorithms for these models leverage sparse hop sets. We bypass the hop set construction; computing a spanner is sufficient with our method. The above bounds assume non-negative integer edge weights that are polynomially bounded in n; for general non-negative weights, running times scale with the logarithm of the maximum ratio between non-zero weights. In case of asymmetric costs for traversing an edge in opposite directions, running times scale with the maximum ratio between the costs of both directions over all edges.

1998 ACM Subject Classification I.1.2 Algorithms, G.1.6 Optimization

Keywords and phrases Shortest Paths, Shortest Transshipment, Undirected Min-cost Flow, Gradient Descent, Spanner

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.7

* Full version available at https://arxiv.org/abs/1607.05127.
[1] We use $\tilde{O}(\cdot)$ to hide polylogarithmic factors in n.

© Ruben Becker, Andreas Karrenbauer, Sebastian Krinninger, and Christoph Lenzen;
licensed under Creative Commons License CC-BY

1 Introduction

Single-source shortest paths (SSSP) is a fundamental and well-studied problem in computer science. Thanks to sophisticated algorithms and data structures [20, 23, 41], it has been known for a long time how to obtain (near-)optimal running time in the RAM model. This is not the case in non-centralized models of computation, which become more and more relevant in a big-data world. Despite certain progress for *exact* SSSP algorithms [6, 7, 9, 15, 28, 30, 39, 40], there remain large gaps to the strongest known lower bounds. Close-to-optimal running times have so far only been achieved by efficient approximation schemes [10, 17, 25, 32]. For instance, in the CONGEST model of distributed computing, the state of the art is as follows: Exact SSSP on weighted graphs can be computed in $O(D^{1/3}(n \log n)^{2/3})$ rounds [15], where D is the (hop) diameter of the graph, and $(1+\varepsilon)$-approximate SSSP can be computed in $(\sqrt{n}+D) \cdot 2^{O(\sqrt{\log n \log(\varepsilon^{-1} \log n)})}$ rounds [25].[2] Even for constant ε, the latter exceeds the strongest known lower bound of $\Omega(\sqrt{n}/\log n + D)$ rounds [13] by a super-polylogarithmic factor. As a consequence of the techniques developed in this paper, we make a qualitative algorithmic improvement for $(1+\varepsilon)$-approximate SSSP in this model: we solve the problem in $(\sqrt{n}+D) \cdot \varepsilon^{-O(1)} \operatorname{polylog} n$ rounds. We thus narrow the gap between upper and lower bound significantly and additionally improve the dependence on ε. Our new approach achieves its superior running time by leveraging techniques from continuous optimization.

It is inherent to our approach that we actually tackle a problem that seems more general than SSSP. In the *shortest transshipment* problem, we seek to find a cheapest routing for sending units of a single good from sources to sinks along the edges of a graph meeting the nodes' demands. Equivalently, we want to find the minimum-cost flow in a graph where edges have unlimited capacity. The special case of SSSP can be modeled as a shortest transshipment problem by setting the demand of the source to $-n+1$ (thus supplying $-n+1$ units) and the demand of every other node to 1. Unfortunately, this relation breaks when we consider approximation schemes: A $(1+\varepsilon)$-approximate solution to the transshipment problem merely yields $(1+\varepsilon)$-approximations to the distances *on average.* In the special case of SSSP, however, one is interested in obtaining a $(1+\varepsilon)$-approximation to the distance for *each single node* and we show how to extend our algorithm to provide such a guarantee as well.

Techniques from continuous optimization have been key to recent breakthroughs in the combinatorial realm of graph algorithms [8, 11, 12, 27, 31, 33, 37]. In this paper, we apply this paradigm to computing primal and dual $(1+\varepsilon)$-approximate solutions to the shortest transshipment problem in undirected graphs with non-negative edge weights. Accordingly, we perform projected gradient descent for a suitable norm-minimization formulation of the problem, where we approximate the infinity norm by a differentiable soft-max function. To make this general approach work in our setting, we need to add significant problem-specific tweaks. In particular, we develop a gradient descent algorithm that reduces the problem of computing a $(1+\varepsilon)$-approximation to the more relaxed problem of computing, e.g., an $O(\log n)$-approximation. We then exploit that an $O(\log n)$-approximation can be computed very efficiently by solving the problem on a sparse spanner, and that it is well-known how to compute sparse spanners efficiently. To obtain the aforementioned per-node guarantee in the approximate SSSP problem, we additionally exploit specific properties of our gradient descent algorithm. Further effort is required to extract an approximate shortest-path tree (i.e., a primal solution) from the dual solution (i.e., estimated distances to the source).

[2] Note that these running times refer to weighted graphs. In unweighted graphs, the SSSP problem can easily be solved in $O(D)$ rounds by performing a BFS tree computation.

Our method is widely applicable among a plurality of non-centralized models of computation in a rather straightforward way. We obtain the first non-trivial algorithms for approximate undirected shortest transshipment in the broadcast CONGEST,[3] broadcast congested clique, and multipass streaming models. As a further, arguably more important, consequence, we improve upon prior results for computing approximate SSSP in these models. Our approximate SSSP algorithms are the first to be provably optimal up to polylogarithmic factors.

Our Contributions and Results. We summarize our technical and conceptual contributions as follows:

(C1) We give a problem-specific gradient descent algorithm for approximating the shortest transshipment, which requires access to an oracle computing an α-approximate dual solution for any given demand vector.[4] To compute a $(1+\varepsilon)$-approximation, the algorithm performs $\tilde{O}(\varepsilon^{-3}\alpha^2)$ oracle calls. If the oracle returns primal solutions, so does our algorithm.

(C2) We provide an additional analysis of the gradient descent algorithm that allows us to extend the method to solving SSSP in order to achieve a per-node approximation guarantee.

(C3) We observe that spanners can be used to obtain an efficient shortest transshipment oracle with approximation guarantee $\alpha \in O(\log n)$.

By implementing our method in specific models of computation, we obtain the following concrete algorithmic results in graphs with non-negative polynomially bounded[5] integer edge weights:

(R1) We give faster algorithms for computing $(1+\varepsilon)$-approximate SSSP:

1. *Broadcast CONGEST model:* We obtain a deterministic algorithm for computing $(1+\varepsilon)$-approximate SSSP using $\tilde{O}((\sqrt{n}+D)\cdot\varepsilon^{-O(1)})$ rounds. This improves upon the previous best upper bound of $(\sqrt{n}+D)\cdot 2^{O(\sqrt{\log n \log(\varepsilon^{-1}\log n)})}$ rounds [25]. For $\varepsilon^{-1} \in O(\operatorname{polylog} n)$, we match the lower bound of $\Omega(\sqrt{n}/\log n + D)$ [13] (applying to any (randomized) $(\operatorname{poly} n)$-approximation of the distance between two fixed nodes in a weighted undirected graph) up to polylogarithmic factors in n.
2. *Broadcast congested clique model:* We obtain a deterministic algorithm for computing $(1+\varepsilon)$-approximate SSSP using $\tilde{O}(\varepsilon^{-O(1)})$ rounds. This improves upon the previous best upper bound of $2^{O(\sqrt{\log n \log(\varepsilon^{-1}\log n)})}$ rounds [25].
3. *Multipass streaming model:* We obtain a deterministic algorithm for computing $(1+\varepsilon)$-approximate SSSP using $\tilde{O}(\varepsilon^{-O(1)})$ passes and $O(n \log n)$ space. This improves upon the previous best upper bound of $(2+1/\varepsilon)^{O(\sqrt{\log n \log\log n})}$ passes and $O(n\log^2 n)$ space [17]. By setting ε small enough, we can compute distances up to the value $\log n$ exactly in integer-weighted graphs using $\operatorname{polylog} n$ passes and $O(n \log n)$ space. Thus, up to polylogarithmic factors in n, our result matches a lower bound of $n^{1+\Omega(1/p)}/\operatorname{poly} p$ space for all algorithms that decide in p passes if the distance between two fixed nodes in an unweighted undirected graph is at most $2(p+1)$ for any $p = O(\log n/\log\log n)$ [22].

[3] Also known as the node-CONGEST model.

[4] Note that dual feasibility is crucial here. In particular, this rules out an oracle based on tree embeddings [2, 16], as such trees might have stretch $\Omega(n)$ on individual edges.

[5] For general non-negative weights, running times scale by a multiplicative factor of $\log R$, where R is the maximum ratio between non-zero edge weights.

(R2) We give fast algorithms for computing $(1+\varepsilon)$-approximate shortest transshipment:

1. *Broadcast CONGEST model:* A deterministic algorithm using $\tilde{O}(\varepsilon^{-3}n)$ rounds.
2. *Broadcast congested clique model:* A deterministic algorithm using $\tilde{O}(\varepsilon^{-3})$ rounds.
3. *Multipass streaming model:* A deterministic algorithm using $\tilde{O}(\varepsilon^{-3})$ passes and $O(n \log n)$ space.

No non-trivial upper bounds were known before in these three models.

In the case of SSSP, we can deterministically compute a $(1+\varepsilon)$-approximation to the distance from the source for every node. Using a *randomized* procedure, we can additionally compute (with high probability within the same asymptotic running times) a tree on which every node has a path to the source that is within a factor of $(1+\varepsilon)$ of its true distance.

In the case of shortest transshipment, we can (deterministically) return $(1+\varepsilon)$-approximate primal and dual solutions. We can further extend the results to asymmetric weights on undirected edges, where each edge can be used in either direction at potentially different costs. Denoting by $\lambda \geq 1$ the maximum over all edges of the cost ratio between traversing the edge in different directions, our algorithms give the same guarantees if the number of rounds or passes, respectively, is increased by a factor of $\lambda^4 \log \lambda$.

Related Work on Shortest Transshipment. Shortest transshipment is a classic problem in combinatorial optimization [29, 36]. The classic algorithms for directed graphs with non-negative edge weights in the RAM model run in time $O(n(m + n \log n) \log n)$ [35] and $O((m + n \log n)B)$ [14], respectively, where B is the sum of the nodes' demands (when they are given as integers) and the term $m + n \log n$ comes from SSSP computations. If the graph contains negative edge weights, then these algorithms require an additional preprocessing step to compute SSSP in presence of negative edge weights, for example in time $O(mn)$ using the Bellman-Ford algorithm [4, 19] or in time $O(m\sqrt{n} \log N)$ using Goldberg's algorithm [21].[6] The weakly polynomial running time was first improved to $\tilde{O}(m^{3/2} \operatorname{polylog} R)$ [12] and then to $\tilde{O}(m\sqrt{n} \operatorname{polylog} R)$ in a recent breakthrough for minimum-cost flow [31], where R is the ratio between the largest and the smallest edge weight. Independent of our work, Sherman [38] obtained a randomized algorithm for computing a $(1+\varepsilon)$-approximate shortest transshipment in weighted undirected graphs in time $O(\varepsilon^{-2}m^{1+o(1)})$ using a generalized-preconditioning approach. We refer the reader to the full paper for a detailed comparison of Sherman's and our approach. We are not aware of any non-trivial algorithms for computing (approximate) shortest transshipment in non-centralized models of computation, such as distributed and streaming models.

Comparison to Hop Set Based SSSP Algorithms. The state-of-the art SSSP algorithms in the distributed CONGEST model follow the framework developed in [34], where (1) the problem of computing SSSP is reduced to an overlay network of size $N = \tilde{O}(\sqrt{n})$ and (2) a sparse hop set is constructed to speed up computing SSSP on the overlay network. An (h, ε)-hop set is a set of weighted edges that, when added to the original graph, provides sufficient shortcuts to approximate all pairwise distances using paths with only h edges ("hops"). In the algorithm by Nanongkai et al. [25], the upper bound of $(\sqrt{n} + D) \cdot 2^{O(\sqrt{\log n \log(\varepsilon^{-1} \log n)})}$ on the number of rounds is achieved by constructing an (h, ε)-hop set of size $O(N\rho)$ where $h \leq 2^{O(\sqrt{\log n \log(\varepsilon^{-1} \log n)})}$ and $\rho \leq 2^{O(\sqrt{\log n \log(\varepsilon^{-1} \log n)})}$. Elkin's algorithm [15], which takes $O(D^{1/3}(n \log n)^{2/3})$ rounds, uses an exact $(N/\rho, 0)$ hop set of size $O(N\rho)$ similar to

[6] Goldberg's running time bound holds for integer-weighted graphs with most negative weight $-N$.

the one developed by Shi and Spencer in a PRAM algorithm [40]. Elkin's main technical contribution lies in showing how to compute this hop set without constructing the overlay network explicitly. Roughly speaking, in these algorithms, *both* h and ρ enter the running time of the corresponding SSSP algorithms, in addition to the time needed to construct the hop set.

The concept of hop sets has been introduced by Cohen in the context of PRAM algorithms for approximate SSSP [10]. The increased interest in hop sets and their applications in the last years [5, 17, 24, 25, 32] has culminated in the construction of (h,ε)-hop sets of size $O(n^{1+\frac{1}{2^{k+1}}-1})$ for $h = O\left((\frac{k}{\varepsilon})^k\right)$ [18, 26]. Recent lower bounds by Abboud et al. [1] show that this trade-off is essentially tight: any construction of (h,ε)-hop sets of size $\leq n^{1+\frac{1}{2^k-1}-\delta}$ must have $h = \Omega_k\left((\frac{1}{\varepsilon})^k\right)$ (where $k \geq 1$ is an integer and $\delta > 0$). This implies that the hop set based algorithms, as long as the factor ρ has to be paid in the running time for construction hop sets of size $n\rho$, will never be able to achieve a running time comparable to our SSSP algorithm exclusively by finding better hop sets.

Spanners. In our approach we use a spanner to obtain an efficient shortest transshipment oracle.

▶ **Definition 1** (Spanner)**.** Given $G = (V, E, w)$ and $\alpha \geq 1$, an *α-spanner* of G is a subgraph $(V, E', w|_{E'})$, $E' \subseteq E$, in which distances are at most by factor α larger than in G.

In other words, a spanner removes edges from G while approximately preserving distances. It is well-known that for every undirected graph we can efficiently compute an α-spanner of size $O(n \log n)$ with $\alpha = O(\log n)$ [3].

Structure of this paper. In the following section, we will first describe the gradient descent algorithm for the case of symmetric weights. More precisely, we will describe how to obtain a primal/dual solution pair of approximation ratio $(1+\varepsilon)$ for an oracle yielding both primal and dual solutions; if the oracle provides dual solutions only, so do our algorithms. In Section 2.2, we describe how to obtain $(1+\varepsilon)$-approximate distances for *every* node in the SSSP case. In Section 2.3, we show how to obtain a $(1+\varepsilon)$-approximate primal *tree* solution. In Section 3, we briefly describe how the above framework can be implemented in various distributed and streaming models of computation. Due to space limitations, we refer to the full paper for further details.

The full version also discusses how our techniques can be generalized to asymmetric edge weights. The key observation is that, essentially, the gradient descent algorithm can be guided by basing the oracle on solving the symmetrized variant problem on an (undirected) spanner. The additional inaccuracy of the approximation slows down the progress of the algorithm by a factor of $\lambda^4 \log \lambda$. However, while this generalization does not affect our approach structurally, some technical obstacles need to be overcome. For the sake of a streamlined presentation, we thus confine the discussion to the symmetric problem.

2 General Approach for Solving Shortest Transshipment and SSSP

Let $G = (V, E)$ be a (w.l.o.g. connected) undirected graph with n nodes, m edges, and positive[7] integral edge weights $w \in \mathbb{Z}^m_{\geq 1}$. Furthermore, let $b \in \mathbb{Z}^n$ be a vector of demands.

[7] Note that excluding 0 as an edge weight is a only a mild restriction, because we can always generate new weights w' with $w'_e = 1 + \lceil n/\varepsilon \rceil \cdot w_e$ while preserving at least one of the shortest paths between

W.l.o.g., we restrict to feasible and non-trivial instances, i.e., $b^T\mathbb{1} = 0$ and $b \neq 0$.[8] A common approach to model the *undirected shortest transshipment* problem as a linear program considers the node-arc-incidence matrix $A \in \{-1, 0, 1\}^{n\times 2m}$ of the corresponding bidirected graph,[9] where we substitute each edge e by a forward and a backward arc with the same weight w_e in both directions. While this may seem redundant, it is convenient in terms of notation and generalizes to the case of asymmetric edge weights considered in the full paper. With W being the $2m \times 2m$ diagonal matrix containing the weights, we obtain the following primal/dual pair of linear programs:

$$\min\{\|Wx\|_1 : Ax = b\} = \max\{b^T y : \|W^{-1}A^T y\|_\infty \leq 1\}, \tag{1}$$

where for $z \in \mathbb{R}^d$ we write $\|z\|_1 = \sum_{i=1}^d |z_i|$ and $\|z\|_\infty = \max_{i\in[d]}\{|z_i|\}$. The primal (left) program asks to "ship" the flow given by b from sources (negative demand) to sinks (positive demand) along the edges of the graph, minimizing the cost of the flow, i.e., $\sum_{e\in E} w_e|x_e|$. Note that, without changing that $Ax = b$ or affecting the objective, we can remove "negative" flow $x_{vw} < 0$ by increasing x_{wv} by $|x_{vw}|$ and setting $x_{vw} = 0$. Thus, w.l.o.g., we may assume that $x \geq 0$; in particular, an optimal solution sends flow only in one direction over any edge.

The dual (right) program asks for potentials y such that for each edge $e = (v, w) \in E$, $|y_v - y_w| \leq w_e$, maximizing $b^T y$. Note that, because $b^T\mathbb{1} = 0$, shifting the potential by $r \times \mathbb{1}$ for any $r \in \mathbb{R}$ does neither change $b^T y$ nor $y_v - y_w$ for any $v, w \in V$. The goal of the dual is thus to maximize the differences in potential of sources and sinks (weighted according to b), subject to the constraint that the potentials of neighbors must not differ by more than the weight of their connecting edge.

In the special case of SSSP with source $s \in V$, we have that (i) $b_s = -n + 1$ and $b_v = 1$ for all $v \neq s$, (ii) an optimal primal solution x^* is given by routing, for each $s \neq v \in V$, one unit of flow along a shortest path from s to v, and (iii) optimal potentials y^* are given by setting y^*_v to the distance from s to v.

2.1 Gradient Descent

We now describe a gradient descent method that, given an oracle that computes α-*approximate* primal and dual solutions to the undirected shortest transshipment problem for any specified demand vector $\tilde{b}$, returns primal and dual feasible solutions x and y to the undirected shortest transshipment problem that are $(1 + \varepsilon)$-close to optimal, i.e., fulfill $\|Wx\|_1 \leq (1 + \varepsilon)b^T y$, using $O(\varepsilon^{-3}\alpha^2 \log \alpha \log n)$ calls to the oracle. We then provide an oracle with $\alpha \in \text{polylog}(n)$. For ease of notation, we assume that $\log \alpha \in \text{polylog}\, n$ throughout this paper.

As our first step, we relate the dual of the shortest transshipment problem to its "reciprocal" linear program that normalizes the objective to 1 and seeks to minimize $\|W^{-1}A^T y\|_\infty$:

$$\min\{\|W^{-1}A^T\pi\|_\infty : b^T\pi = 1\}. \tag{2}$$

We denote by π^* an optimal solution to this problem, whereas y^* denotes an optimal solution to the dual of the original problem (1). It is easy to see that feasible solutions π of (2) that satisfy $\|W^{-1}A^T\pi\|_\infty > 0$ are mapped to feasible solutions of the dual program in (1)

each pair of nodes as well as $(1 + \varepsilon)$-approximations. As we assume edge weights to be integer we can assume that $\varepsilon \geq 1/(n\|w\|_\infty)$ (as otherwise it is required to compute an exact solution) and thus our asymptotic running time bounds are not affected by this modification.

[8] Here $\mathbb{1}$ denotes the all-ones vector and thus $b^T\mathbb{1} = 0$ simply means that the positive demands equal the negative demands (i.e., the supplies).

[9] The incidence matrix A of a directed graph contains a row for every node and a column for every arc and $A_{i,j}$ is -1 if the j-th arc leaves the i-th vertex, 1 if it enters the vertex, and 0 otherwise.

via $f(\pi) := \pi / \|W^{-1}A^T\pi\|_\infty$. Similarly, feasible solutions y of the dual program in (1) that satisfy $b^T y > 0$ are mapped to feasible solutions of (2) via $g(y) := y/b^T y$. Moreover, the map $f(\cdot)$ preserves the approximation ratio. Namely, for any $\varepsilon > 0$, if π is a solution of (2) within factor $1+\varepsilon$ of the optimum, then $f(\pi)$ is feasible for (1) and within factor $1+\varepsilon$ of the optimum. In particular, $f(\pi^*)$ is an optimal solution of (1).

We would like to apply gradient descent to (2). However, this is not readily possible, since the objective is not differentiable. Hence, we will change the problem another time by using the so-called soft-max function (a.k.a. log-sum-exp or lse for short), which is a suitable approximation for the maximum entry $(v)_{\max} := \max\{v_i : i \in [d]\}$ of a vector $v \in \mathbb{R}^d$.[10] It is defined as $\mathrm{lse}_\beta(v) := \frac{1}{\beta} \ln\left(\sum_{i\in[d]} e^{\beta v_i}\right)$, where $\beta > 0$ is a parameter that controls the accuracy of the approximation of the maximum at the expense of smoothness. We note that $\mathrm{lse}_\beta(\cdot)$ is a convex function for any $\beta > 0$ and provides the following additive approximation of the maximum:

$$(x)_{\max} = \frac{1}{\beta} \ln e^{\beta\cdot(x)_{\max}} \leq \mathrm{lse}_\beta(x) \leq \frac{1}{\beta} \ln \sum_{i\in[d]} e^{\beta\cdot(x)_{\max}} = \frac{\ln(d)}{\beta} + (x)_{\max}. \tag{3}$$

A trade-off in the choice of β arises because β also controls the smoothness of the lse-function. Formally, lse_β is β-Lipschitz smooth (i.e., its gradient is β-Lipschitz continuous) w.r.t. to the pair 1-norm/∞-norm:

$$\|\Phi_\beta(x) - \Phi_\beta(y)\|_1 \leq \beta\|x-y\|_\infty. \tag{4}$$

Using the soft-max function, we define the *potential function*

$$\Phi_\beta(\pi) := \mathrm{lse}_\beta\left(W^{-1}A^T\pi\right).$$

Recalling that A was defined to represent each edge of the graph by a forward and backward arc, we see that $(W^{-1}A^T\pi)_{\max} = \|W^{-1}A^T\pi\|_\infty$, i.e., $\Phi_\beta(\pi)$ is indeed a smooth approximation of the objective of (2). In order to control the approximation error, β is adapted in the course of the algorithm such that the additive error $\ln(2m)/\beta$ is always at most $\frac{\varepsilon}{4}\Phi_\beta(\pi)$. Thus, we maintain a multiplicative approximation of the dual objective function of (2), i.e.,

$$\|W^{-1}A^T\pi\|_\infty \leq \Phi_\beta(\pi) \leq \frac{\|W^{-1}A^T\pi\|_\infty}{1-\varepsilon/4}. \tag{5}$$

Our gradient descent algorithm, see Algorithm 1 for a pseudo-code implementation, first computes a starting solution π that is an α-approximate (dual) solution to (2) and an initial β that is appropriate for π as discussed above. This can be done, e.g., by solving the problem on an α-spanner and scaling down (by at most a factor of α) to obtain a feasible solution for the original graph. In each iteration, it updates the potentials π using an α-approximate solution to a shortest transshipment problem with a modified demand vector $\tilde{b}$ that depends on the gradient. Depending on the objective value of this approximation, the algorithm either performs an update to π or terminates, see the check for the value of δ in the algorithm.

The intuition behind the algorithm is the following. As the potential function is differentiable, its gradient exists everywhere and it points in the opposite direction of the steepest descent. However, our update steps must maintain the constraint $b^T\pi = 1$, i.e., they must lie in the orthogonal complement of b. To this end, we consider the projection of the gradient

[10] Note the difference to the ∞-norm, which is defined as the maximum of the *absolute values* of the entries of a vector.

Algorithm 1: `gradient_transship` (G, b, ε)

Compute α-approximation π to $\min\{\|W^{-1}A^T\pi\|_\infty : b^T\pi = 1\}$. *// use oracle*
Determine β so that $4\ln(2m) \leq \varepsilon\beta\Phi_\beta(\pi) \leq 5\ln(2m)$.
repeat
  Set $\tilde{b} := P^T\nabla\Phi_\beta(\pi)$, where $P := I - \pi b^T$. *// project to maintain $b^T\pi = 1$*
  if $\tilde{b} = 0$ **then return** π *// Special case: optimal solution found*
  Determine $\tilde{h}$ with $\|W^{-1}A^T\tilde{h}\|_\infty = 1$ and $\tilde{b}^T\tilde{h} \geq \frac{1}{\alpha}\max\{\tilde{b}^T h : \|W^{-1}A^T h\|_\infty \leq 1\}$.
  // $\tilde{h}$ can be obtained from the oracle with demand vector $\tilde{b} = P^T\nabla\Phi_\beta(\pi)$
  Set $\delta := \frac{\tilde{b}^T\tilde{h}}{\|W^{-1}A^TP\tilde{h}\|_\infty}$. *// δ measures closeness to optimality*
  if $\delta > \frac{\varepsilon}{8\alpha}$ **then** $\pi \leftarrow \pi - \frac{\delta}{2\beta\|W^{-1}A^TP\tilde{h}\|_\infty}P\tilde{h}$. *// project to maintain $b^T\pi = 1$*
  while $4\ln(2m) \geq \varepsilon\beta\Phi_\beta(\pi)$ **do** $\beta \leftarrow \frac{5}{4}\beta$. *// find appropriate β*
until $\delta \leq \frac{\varepsilon}{8\alpha}$
return π

$P^T\nabla\Phi_\beta(\pi)$.[11] Because the gradient, and hence the direction of the steepest descent, changes when we move away from our current solution, we use an adaptive step width restricting the update to a region for which we know that the gradient does not vary too much.

If we had a sufficiently good guarantee on the Lipschitz smoothness of $\Phi_\beta(\cdot)$, using the gradient itself (resp. its projection) as the update direction h (i.e., performing the update $\pi \leftarrow \pi - \eta h$ for an appropriate step width η) would decrease the objective $\Phi_\beta(\pi)$ fast enough. However, we only have such a guarantee on the Lipschitz smoothness of $\mathrm{lse}_\beta(\cdot)$. By the convexity of the objective $\Phi_\beta(\pi)$, we can argue that the (normalized) progress of an update direction h is the ratio $P^T\nabla\Phi_\beta(\pi)/\|W^{-1}A^Th\|_\infty$, which suggests finding h by $\max\{\nabla\Phi_\beta(\pi)^TPh : \|W^{-1}A^Th\|_\infty \leq 1\}$. Note that this linear program is precisely of the form (1), with demand vector $\tilde{b} := P^T\nabla\Phi_\beta(\pi)$, and is thus not easier to solve as the original problem. However, finding an approximately optimal update direction only mildly affects the number of iterations, i.e., querying the oracle for an α-approximate dual solution with demand $\tilde{b}$ yields the desired guarantee.

We then use the projection P to derive a feasible update and rescale so that the gradient does not change too much, enabling us to prove a sufficiently strong progress guarantee – unless the current solution is already close to the optimum. This is captured by δ, which is guaranteed to be large in case significant progress still can be made. Conversely, a small δ implies that we are close to the optimum. Accordingly, at termination π is a near-optimal solution of (2), and rescaling to $y = \pi/\|W^{-1}A^T\pi\|_\infty$ yields a near-optimal dual solution of (1). Here, scaling up β as the potential decreases ensures that the incurred approximation error is sufficiently small. On the other hand, using large β and having the guarantee that δ is large as long as we are not close to the optimum guarantees that the potential function decreases rapidly and only a small number of iterations is required.

We proceed by formalizing this intuition. First, we show that a primal-dual pair that is $(1+\varepsilon)$-close to optimal in (1) can be constructed from the output potentials π and α-approximate primal and dual solutions, say $\tilde{f}$ and $\tilde{h}$, to the transshipment problem that was solved in the last iteration of the algorithm. If one is only interested in a dual solution to (1), then the α-approximate dual solution $\tilde{h}$ is enough and thus only an oracle providing

[11] As $b^T\pi = 1$, we have that $b^TPh = b^T(I - \pi b^T)h = b^Th - b^T\pi b^Th = 0$ for all h.

a dual solution is required, as done in Algorithm 1. A primal solution can be obtained from $\tilde{f}$ and the vector $\tilde{x} := W^{-1}\nabla \operatorname{lse}_\beta\left(W^{-1}A^T\pi\right)$. This choice of $\tilde{x}$ is obtained by applying the chain rule of differentiation to $\nabla\Phi_\beta(\pi)$, i.e., $\nabla\Phi_\beta(\pi) = AW^{-1}\nabla \operatorname{lse}_\beta\left(W^{-1}A^T\pi\right)$. In the correctness proof we allow a more general choice of $\tilde{x}$, which we will exploit later on for finding a tree solution for approximate SSSP.

▶ **Lemma 2** (Correctness). *Let $0 < \varepsilon \leq 1/2$,*

- *$\pi \in \mathbb{R}^n$ and $\beta \in \mathbb{R}$ denote the return values of Algorithm* `gradient_transship`*,*
- *$\tilde{f} \in \mathbb{R}^{2m}$ and $\tilde{h} \in \mathbb{R}^n$ be the α-approximate pair returned by the oracle in the last iteration of Algorithm* `gradient_transship`*, and*
- *$\tilde{x} \in \mathbb{R}^{2m}$ be such that $A\tilde{x} = \nabla\Phi_\beta(\pi)$ and $\|W\tilde{x}\|_1 \leq 1 + \varepsilon/8$.*

Then $x := \frac{\tilde{x}-\tilde{f}}{\pi^T\nabla\Phi_\beta(\pi)}, y := \frac{\pi}{\|W^{-1}A^T\pi\|_\infty}$ is a $(1+\varepsilon)$-approximate pair, i.e., it holds that $Ax = b$, $\|W^{-1}A^Ty\|_\infty \leq 1$, and $\|Wx\|_1 \leq (1+\varepsilon)b^Ty$.

Proof. First note that $A\tilde{f} = \tilde{b}$ and $\tilde{b} = P^T\nabla\Phi_\beta(\pi) = \nabla\Phi_\beta(\pi) - b\pi^T\nabla\Phi_\beta(\pi)$. Thus $Ax = \frac{\nabla\Phi_\beta(\pi)-\tilde{b}}{\pi^T\nabla\Phi_\beta(\pi)} = b$. Moreover, $\|W^{-1}A^Ty\|_\infty = 1$ follows directly from the definition of y.

It remains to show that $\|Wx\|_1 \leq (1+\varepsilon)b^Ty$.[12] It can be shown (see full paper for details) that convexity of $\Phi_\beta(\cdot)$ and the guarantee on β yield

$$\pi^T\nabla\Phi_\beta(\pi) \geq \left(1 - \frac{\varepsilon}{4}\right)\Phi_\beta(\pi) \geq \left(1 - \frac{\varepsilon}{4}\right)\|W^{-1}A^T\pi\|_\infty > 0. \tag{6}$$

Hence, $|\pi^T\nabla\Phi_\beta(\pi)| = \pi^T\nabla\Phi_\beta(\pi)$. Moreover, $\|W\tilde{x}\|_1 \leq 1 + \varepsilon/8$ by assumption and thus

$$\|Wx\|_1 \overset{\Delta\text{-ineq.}}{\leq} \frac{1 + \frac{\varepsilon}{8} + \|W\tilde{f}\|_1}{\pi^T\nabla\Phi_\beta(\pi)} \overset{\alpha\text{-approx.}}{\leq} \frac{1 + \frac{\varepsilon}{8} + \alpha\tilde{b}^T\tilde{h}}{\pi^T\nabla\Phi_\beta(\pi)} = \frac{1 + \frac{\varepsilon}{8} + \alpha\delta\|W^{-1}A^TP\tilde{h}\|_\infty}{\pi^T\nabla\Phi_\beta(\pi)},$$

where $\delta = \frac{\tilde{b}^T\tilde{h}}{\|W^{-1}A^TP\tilde{h}\|_\infty}$ as in Algorithm `gradient_transship`. By the definition of $P = I - \pi b^T$ and the triangle inequality for the infinity norm, we obtain $\|W^{-1}A^TP\tilde{h}\|_\infty \leq \|W^{-1}A^T\tilde{h}\|_\infty + |b^T\tilde{h}|\|W^{-1}A^T\pi\|_\infty$. Using the upper bound $|b^T\tilde{h}| \leq b^Ty^*$ from the optimality of y^* and $\|W^{-1}A^T\tilde{h}\|_\infty \leq 1$, we obtain $\|W^{-1}A^TP\tilde{h}\|_\infty \leq 1 + \|W^{-1}A^T\pi\|_\infty b^Ty^*$. Using (6) for the denominator, this yields

$$\|Wx\|_1 \leq \frac{1 + \frac{\varepsilon}{8} + \alpha\delta(1 + \|W^{-1}A^T\pi\|_\infty b^Ty^*)}{(1 - \frac{\varepsilon}{4})\|W^{-1}A^T\pi\|_\infty} \leq \frac{1 + \frac{\varepsilon}{8} + \frac{\varepsilon}{8}(1 + \frac{\|Wx\|_1}{b^Ty})}{(1 - \frac{\varepsilon}{4})}b^Ty,$$

since $b^Ty^* \leq \|Wx\|_1$ by weak duality, $\|W^{-1}A^T\pi\|_\infty = 1/b^Ty$, and $\delta \leq \frac{\varepsilon}{8\alpha}$ at termination of the algorithm. Thus $(1 + \frac{\varepsilon}{4})/(1 - \frac{3\varepsilon}{8}) \leq (1 + \frac{\varepsilon}{4})/(1 - \frac{\varepsilon}{2}) \leq (1 + \varepsilon)$ yields the result. ◀

It remains to show a bound on the number of iterations until termination. To this end, we establish that the potential function decreases by a multiplicative factor in each iteration.

▶ **Lemma 3** (Multiplicative Decrement of Φ_β). *Let $\pi \in \mathbb{R}^n$, let β satisfy $\varepsilon\beta\Phi_\beta(\pi) \leq 5\ln(2m)$, and let $\tilde{h}$ satisfy $\|W^{-1}A^TP\tilde{h}\|_\infty > 0$, where $P = I - \pi b^T$. Then, for $\delta := \frac{\tilde{b}^T\tilde{h}}{\|W^{-1}A^TP\tilde{h}\|_\infty}$, where $\tilde{b} = P^T\nabla\Phi_\beta(\pi)$, it holds that*

$$\Phi_\beta\left(\pi - \frac{\delta}{2\beta\|W^{-1}A^TP\tilde{h}\|_\infty}P\tilde{h}\right) \leq \left(1 - \frac{\varepsilon\delta^2}{20\ln(2m)}\right)\Phi_\beta(\pi).$$

[12] Here, we omit the special case $\tilde{b} = 0$, which guarantees optimality. See full version for details.

Proof. Let us denote $h := \frac{\delta}{2\beta\|W^{-1}A^T P\tilde{h}\|_\infty}\tilde{h}$. Recall that $\Phi_\beta(\cdot)$ is convex, thus

$$\begin{aligned}\Phi_\beta(\pi - Ph) - \Phi_\beta(\pi) &\overset{\text{Convexity}}{\leq} -\nabla\Phi_\beta\big(\pi - Ph\big)^T Ph + \nabla\Phi_\beta(\pi)^T Ph - \nabla\Phi_\beta(\pi)^T Ph \\ &= \left[\nabla \operatorname{lse}_\beta\left(W^{-1}A^T\pi\right) - \nabla \operatorname{lse}_\beta\left(W^{-1}A^T(\pi - Ph)\right)\right]^T W^{-1}A^T Ph - \tilde{b}^T h \\ &\overset{\text{Hölder}}{\leq} \|\nabla \operatorname{lse}_\beta\left(W^{-1}A^T\pi\right) - \nabla \operatorname{lse}_\beta\left(W^{-1}A^T(\pi - Ph)\right)\|_1 \|W^{-1}A^T Ph\|_\infty - \tilde{b}^T h \\ &\overset{\beta\text{-Lipschitz}}{\leq} \beta\|W^{-1}A^T Ph\|_\infty^2 - \tilde{b}^T h,\end{aligned}$$

where we used Hölder's inequality[13] and then the fact that the lse_β-function is β-Lipschitz smooth (see (4)). Using the definitions of h and δ yields $\Phi_\beta(\pi - Ph) - \Phi_\beta(\pi) \leq \frac{\delta^2}{4\beta} - \frac{\delta^2}{2\beta} = -\frac{\delta^2}{4\beta}$. Using the upper bound on β yields the result. ◀

This progress guarantee is sufficient to show the following bound on the number of iterations.

▶ **Lemma 4** (Number of Iterations). *Suppose that $0 < \varepsilon \leq 1/2$. Then, it holds that Algorithm* `gradient_transship` *terminates within $O(\varepsilon^{-3}\alpha^2 \log\alpha \log n)$ iterations.*

Proof. Note that for all $x \in \mathbb{R}^n$, $\nabla_\beta \operatorname{lse}_\beta(x) \leq 0$, i.e., lse_β is decreasing as a function of β and thus the while-loop that scales β up does not increase $\Phi_\beta(\pi)$. Denote by β_0 and π_0 the initial values of β and π, respectively, and by β and π the values at termination. By Lemma 3 and the fact that the algorithm ensures $\delta > \varepsilon/(8\alpha)$ as long as it does not terminate, the potential decreases by a factor of $1 - \frac{\varepsilon\delta^2}{20\ln(2m)} \leq 1 - \frac{\varepsilon^3}{1280\alpha^2\ln(2m)}$.[14] Hence, the number of iterations k can be bounded by

$$k \leq \log\left(\frac{\Phi_\beta(\pi)}{\Phi_{\beta_0}(\pi_0)}\right)\left(\log\left(1 - \frac{\varepsilon^3}{1280\alpha^2\ln(2m)}\right)\right)^{-1} \leq \log\left(\frac{\Phi_{\beta_0}(\pi_0)}{\Phi_\beta(\pi)}\right)\frac{1280\alpha^2\ln(2m)}{\varepsilon^3}.$$

As $\ln(2m) \in O(\log n)$, it remains show that $\frac{\Phi_{\beta_0}(\pi_0)}{\Phi_\beta(\pi)} \in O(\alpha)$. Using that π_0 is an α-approximate solution and that β_0 is such that $4\ln(2m) \leq \varepsilon\beta_0\Phi_{\beta_0}(\pi^0)$, we obtain that

$$\Phi_{\beta_0}(\pi^0) = \operatorname{lse}_{\beta_0}\left(W^{-1}A^T\pi^0\right) \overset{(3)}{\leq} \|W^{-1}A^T\pi^0\|_\infty + \frac{\ln(2m)}{\beta_0} \leq \alpha\|W^{-1}A^T\pi^*\|_\infty + \frac{\varepsilon\Phi_{\beta_0}(\pi^0)}{4}$$

and thus $\Phi_{\beta_0}(\pi^0) \leq \alpha\|W^{-1}A^T\pi^*\|_\infty/(1-\varepsilon/4)$. On the other hand, $\Phi_\beta(\pi) \geq \|W^{-1}A^T\pi\|_\infty \geq \|W^{-1}A^T\pi^*\|_\infty$ and thus $\frac{\Phi_{\beta_0}(\pi^0)}{\Phi_\beta(\pi)} \leq \frac{\alpha}{1-\varepsilon/4} = O(\alpha)$ and the bound follows. ◀

We remark that one can first run the gradient descent algorithm with $\varepsilon = 1/2$ and then switch to the desired accuracy. Using this trick, the above bound slightly improves to $O((\varepsilon^{-3} + \log\alpha)\alpha^2 \log n)$. From the discussion so far, we obtain the following result.

▶ **Theorem 5.** *Given an oracle that computes α-approximate solutions to the undirected transshipment problem, using Algorithm* `gradient_transship`*, we can compute primal and dual solutions x, y to the shortest transshipment problem satisfying $\|Wx\|_1 \leq (1+\varepsilon)b^T y$ with $\tilde{O}(\varepsilon^{-3}\alpha^2)$ oracle calls. If the oracle only returns α-approximate dual solutions, then Algorithm* `gradient_transship` *computes a $(1+\varepsilon)$-approximate dual solution.*

---

[13] Hölder's inequality states that $x^T y \leq \|x\|_p\|y\|_q$ for p, q satisfying $\frac{1}{p} + \frac{1}{q} = 1$, assuming $\frac{1}{\infty} = 0$.

[14] Here, we omit the technical argument that the condition $\|W^{-1}A^T P\tilde{h}\|_\infty > 0$ of Lemma 3 is always fulfilled when we apply the lemma. See full version for details.

Algorithm 2: `sssp` (G, s, ε)

Let $\hat{y} = 0$, $b = \mathbb{1} - n\mathbb{1}_s$, and $\varepsilon' = \frac{\varepsilon^3}{3840\alpha^2 \ln(2m)}$.
while $b_s < 0$ **do**
  Set $\pi = \texttt{gradient\_transship}(G, b, \varepsilon')$ and $y = \frac{\pi}{\|W^{-1}A^T\pi\|_\infty}$.
  Determine β so that $4\ln(2m) < \varepsilon'\beta\Phi_\beta(y) \leq 5\ln(2m)$ and compute $\nabla\Phi_\beta(y)$.
  for *each* $v \in V$ *with* $b_v = 1$ **do**
    Set $\tilde{b} := P^T\nabla\Phi_\beta(y)$, where $P := [I - \frac{y}{(\mathbb{1}_v - \mathbb{1}_s)^T y}(\mathbb{1}_v - \mathbb{1}_s)^T]$.
    Compute $\tilde{h}$ with $\|W^{-1}A^T\tilde{h}\|_\infty = 1$ and $\tilde{b}^T\tilde{h} \geq \frac{1}{\alpha}\max\{\tilde{b}^T h : \|W^{-1}A^T h\|_\infty \leq 1\}$. *// $\tilde{h}$ can be obtained from the oracle with demand vector $\tilde{b} = P^T\nabla\Phi_\beta(\pi)$*
    Set $\delta := \frac{\tilde{b}^T\tilde{h}}{\|W^{-1}A^T P\tilde{h}\|_\infty}$.
    if $\delta \leq \frac{\varepsilon}{8\alpha}$ **then** set $b_v = 0$, $\hat{y}_v = y_v - y_s$ and $b_s \leftarrow b_s + 1$
return $\hat{y}$

2.2 Single-Source Shortest Paths

In the special case of SSSP, we have $b_v = 1$ for all $v \in V \setminus \{s\}$ and $b_s = 1 - n$ for the source s. In fact, it is the combination of $n-1$ shortest s-t-path problems. Let π be the potentials returned by Algorithm `gradient_transship` and let us assume, w.l.o.g., that $\pi_s = 0$ (otherwise shift $\pi \leftarrow \pi - \pi_s\mathbb{1}$). Recall that in an optimal solution π^* with $\pi^*_s = 0$ the value of π^*_v for any v denotes the distance from s to v. Thus the approximation guarantee from Theorem 5 yields that for the potentials π, it holds that $\sum_{v\neq s}\pi_v \leq (1+\varepsilon)\sum_{v\neq s}\pi^*_v$, i.e., the distances merely approximate the optimal distances *on average* over all sink-nodes, which is unsatisfactory. However, we can obtain potentials π such that for *every* v, it holds that $\pi_v \leq (1+\varepsilon)\pi^*_v$ and equivalently $y^* \geq y_v \geq y^*_v/(1+\varepsilon)$ for the s-v-distances.

Using the tools proposed above, we can show that when running the gradient descent algorithm with higher precision, we can determine "good" nodes for which we know the distance with sufficient accuracy by checking, for every node v, whether the gradient would allow further progress for the s-v shortest path problem. We then argue that a constant fraction of the nodes will be "good" when the algorithm is finished. We then concentrate on the other nodes by adapting the demand vector b accordingly, i.e., setting $b_v = 0$ for all good nodes v. We iterate until all nodes are good. The pseudocode is given in Algorithm `sssp`.

▶ **Theorem 6.** *Let $y^* \in \mathbb{R}^n$ denote the distances of all nodes from the source node s. Algorithm `sssp` computes a vector $y \in \mathbb{R}^n$ with $\|W^{-1}A^T y\|_\infty \leq 1$ such that $y^*_v/(1+\varepsilon) \leq y_v \leq y^*_v$ holds for each $v \in V$, using $\mathrm{polylog}(n, \|w\|_\infty)$ calls to Algorithm `gradient_transship`.*

2.3 Finding a Primal Tree Solution

In the following, we explain how to obtain primal tree solutions, for a specific implementation of the transshipment oracle from Section 3, where we solve the subproblem on spanner.

Recall that, as shown in Lemma 2, $x := \frac{\tilde{x}-\tilde{f}}{\pi^T\nabla\Phi_\beta(\pi)}$ is a $(1+\epsilon)$-approximate primal solution, where $\tilde{f}$ is the primal solution computed by the oracle in the last iteration of the algorithm and $\tilde{x} := W^{-1}\nabla\,\mathrm{lse}_\beta(W^{-1}A^T\pi)$. To also obtain a $(1+\epsilon)$-approximate primal *tree* solution, we first sample a tree, say T_1, from $\tilde{x}$ by sampling for each node among its incident edges a parent edge with probabilities proportional to the values in $\tilde{x}$. Then we compute an optimal

tree solution x_T in the graph $G' = (V, T_1 \cup S)$ consisting of the tree T_1 and the edges of the spanner S. As $\mathrm{E}[\|Wx(T_1)\|_1] = 1$, using Markov's inequality we get that, with probability $\Omega(\varepsilon)$, the tree solution $x(T_1)$ corresponding to T_1 satisfies $\|Wx(T_1)\|_1 \leq 1 + \varepsilon/8$. Repeating the sampling $O(\varepsilon \log n)$ times and taking the best result, we obtain such a solution with high probability. Thus, using Lemma 2, we can conclude that the optimal tree solution x_T in G' is $(1+\varepsilon)$-approximate for the problem. To obtain an approximate tree solution in the case of SSSP, we repeat this sampling after every call of the gradient descent algorithm, obtaining a tree T_i in the i-th call. We can then find the approximate shortest path tree in the graph $G' = (V, \bigcup_i T_i \cup S)$ combining the sampled edges of each iteration and the initial spanner. Note that, since the number of calls to the gradient descent algorithm is $\mathrm{polylog}(n, \|w\|_\infty)$, the resulting graph is still of size $O(n \,\mathrm{polylog}(n, \|w\|_\infty))$ for a spanner of size $O(n \log n)$.

3 Implementation in Various Models of Computation

Common to all our implementations is the use of sparse spanners. An optimal solution of an instance of the shortest transshipment problem on an α-spanner of the input graph is an α-approximate solution to the original problem. Thus, whenever our gradient descent algorithm asks the oracle for an α-approximate solution to a subproblem, we solve the subproblem on a spanner to get an approximation with $\alpha = O(\log n)$.

Broadcast Congested Clique. In the *broadcast congested clique* model, the system consists of n fully connected nodes labeled by unique $O(\log n)$-bit identifiers. Computation proceeds in synchronous rounds, where in each round, nodes may perform arbitrary local computations, broadcast (send) an $O(\log n)$-bit message to the other nodes, and receive the messages from other nodes. The input is distributed among the nodes. The first part of the input of every node consists of its incident edges (given by their endpoints' identifiers) and their weights. The second part of the input is problem specific: for the transshipment problem, every node v knows its demand b_v and for SSSP v knows whether or not it is the source s. In both cases, every node knows $0 < \varepsilon \leq 1/2$ as well. Each node needs to compute its part of the output. For shortest transshipment, every node in the end needs to know a $(1+\varepsilon)$-approximation of the optimum value, and for SSSP every node needs to know a $(1+\varepsilon)$-approximation of its distance to the source. The complexity of the algorithm is measured in the worst-case number of rounds until the computation is complete.

Implementing our approach in this model is straightforward. The key observations are:

- Every node can locally aggregate information about its incident edges (e.g. concerning the "stretches" under the potential of the current solution π) and make it known to all other nodes in a single communication round. Thus, given $\beta > 0$ and $\pi \in \mathbb{R}^n$, it is rather straightforward to evaluate $\Phi_\beta(\pi)$ and $\nabla\Phi_\beta(\pi)$ in a constant number of rounds.
- An $O(\log n)$-spanner of the input graph can be computed and made known to all nodes quickly, following the algorithm of Baswana and Sen [3] (see full paper for details).
- Local computation then suffices to solve (sub)problems on the spanner optimally. In particular, $O(\log n)$-approximations to transshipment problems can be computed easily. It suffices to communicate the demand vector; in cases where the demand vector is known a priori (e.g. when strengthening the approximation guarantee from average to worst-case for each node in the SSSP problem), even this is not necessary.

▶ **Theorem 7.** *For any $0 < \varepsilon \leq 1/2$, in the broadcast congested clique model a deterministic $(1+\varepsilon)$-approximation to the shortest transshipment problem in undirected graphs with non-negative edge weights can be computed in $\varepsilon^{-3}\,\mathrm{polylog}\, n$ rounds.*

▶ **Theorem 8.** *For any $0 < \varepsilon \leq 1$, in the broadcast congested clique model a deterministic $(1+\varepsilon)$-approximation to single-source shortest paths in undirected graphs with non-negative edge weights can be computed in $\varepsilon^{-9}\operatorname{polylog} n$ rounds.*

To compute a tree solution, the main observation is that the sampling of the tree can be performed locally at every node.

Broadcast CONGEST Model. The *broadcast CONGEST* model differs from the broadcast congested clique in that communication is restricted to edges that are present in the input graph. That is, node v receives the messages sent by node w if and only if $\{v, w\} \in E$. All other aspects of the model are identical to the broadcast congested clique. We stress that this restriction has significant impact, however: Denoting the hop diameter of the input graph (i.e., the diameter of the unweighted graph $G = (V, E)$) by D, it is straightforward to show that $\Omega(D)$ rounds are necessary to solve the SSSP problem. Moreover, it has been established that $\Omega(\sqrt{n}/\log n)$ rounds are required even on graphs with $D \in O(\log n)$ [13]. Both of these bounds apply to randomized approximation algorithms.

Our main result for this model is that we can nearly match the above lower bounds for approximate SSSP computation. The solution is based on combining a known reduction to an overlay network on $\tilde{\Theta}(\varepsilon^{-1}\sqrt{n})$ nodes, simulating the broadcast congested clique on this overlay, and applying Theorem 8. Simulating a round of the broadcast congested clique for k nodes is done by pipelining each of the k messages over a breadth-first search tree of the underlying graph, taking $O(D + k)$ rounds.

▶ **Corollary 9.** *For any $0 < \varepsilon \leq 1$, in the broadcast CONGEST model a deterministic $(1+\varepsilon)$-approximation to single-source shortest paths in undirected graphs with non-negative weights can be computed in $\tilde{O}((\sqrt{n} + D) \cdot \varepsilon^{-9})$ rounds.*

Multipass Streaming Model. In the *streaming* model the input graph is presented to the algorithm edge by edge as a "stream" without repetitions. The goal is to design algorithms that use as little space as possible. Space is counted in memory words, where we assume that an edge weight or a node identifier fits into a word. In the *multipass streaming* model, the algorithm may make several such passes over the input stream and the goal is to keep the number of passes small (again using little space). For graph algorithms, the usual assumption is that the edges of the graph are presented to the algorithm in arbitrary order.

The main observation is that we can apply the same approach as before with $O(n \log n)$ space: this enables us to store a spanner throughout the entire computation, and we can keep track of intermediate (node) state vectors. Computations on the spanner are thus "free," while $\Phi_\beta(\pi)$ and $\nabla\Phi_\beta(\pi)$ can be evaluated in a single pass by straightforward aggregation. It follows that $\varepsilon^{-O(1)}\operatorname{polylog} n$ passes suffice for completing the computation.

▶ **Theorem 10.** *For any $0 < \varepsilon \leq 1/2$, in the multipass streaming model a deterministic $(1+\varepsilon)$-approximation to the shortest transshipment problem in undirected graphs with non-negative weights can be computed in $\varepsilon^{-3}\operatorname{polylog} n$ passes with $O(n \log n)$ space.*

▶ **Theorem 11.** *For any $0 < \varepsilon \leq 1$, in the multipass streaming model, a deterministic $(1+\varepsilon)$-approximation to single-source shortest paths in undirected graphs with non-negative weights can be computed in $\varepsilon^{-9}\log(\|w\|_\infty)\operatorname{polylog} n$ passes with $O(n \log n)$ space.*

References

1 Amir Abboud, Greg Bodwin, and Seth Pettie. A hierarchy of lower bounds for sublinear additive spanners. In *Symposium on Discrete Algorithms (SODA)*, pages 568–576, 2017. doi:10.1137/1.9781611974782.36.

2 Yair Bartal. On approximating arbitrary metrices by tree metrics. In *Symposium on the Theory of Computing (STOC)*, pages 161–168, 1998. doi:10.1145/276698.276725.

3 Surender Baswana and Sandeep Sen. A simple and linear time randomized algorithm for computing sparse spanners in weighted graphs. *Random Structures & Algorithms*, 30(4):532–563, 2007. Announced at ICALP'03. doi:10.1002/rsa.20130.

4 Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

5 Aaron Bernstein. Fully dynamic $(2 + \epsilon)$ approximate all-pairs shortest paths with fast query and close to linear update time. In *Symposium on Foundations of Computer Science (FOCS)*, pages 693–702, 2009. doi:10.1109/FOCS.2009.16.

6 Gerth Stølting Brodal, Jesper Larsson Träff, and Christos D. Zaroliagis. A parallel priority queue with constant time operations. *Journal of Parallel and Distributed Computing*, 49(1):4–21, 1998. Announced at IPPS'97. doi:10.1006/jpdc.1998.1425.

7 Keren Censor-Hillel, Petteri Kaski, Janne H. Korhonen, Christoph Lenzen, Ami Paz, and Jukka Suomela. Algebraic methods in the congested clique. In *Symposium on Principles of Distributed Computing (PODC)*, pages 143–152, 2015. doi:10.1145/2767386.2767414.

8 Paul Christiano, Jonathan A. Kelner, Aleksander Mądry, Daniel A. Spielman, and Shang-Hua Teng. Electrical flows, laplacian systems, and faster approximation of maximum flow in undirected graphs. In *Symp. on Theory of Computing (STOC)*, pages 273–282, 2011. doi:10.1145/1993636.1993674.

9 Edith Cohen. Using selective path-doubling for parallel shortest-path computations. *Journal of Algorithms*, 22(1):30–56, 1997. Announced at ISTCS'93. doi:10.1006/jagm.1996.0813.

10 Edith Cohen. Polylog-time and near-linear work approximation scheme for undirected shortest paths. *Journal of the ACM*, 47(1):132–166, 2000. Announced at STOC'94. doi:10.1145/331605.331610.

11 Michael B. Cohen, Aleksander Madry, Piotr Sankowski, and Adrian Vladu. Negative-weight shortest paths and unit capacity minimum cost flow in $\tilde{O}(m^{10/7} \log W)$ time. In *Symposium on Discrete Algorithms (SODA)*, 2017. doi:10.1137/1.9781611974782.48.

12 Samuel I. Daitch and Daniel A. Spielman. Faster approximate lossy generalized flow via interior point algorithms. In *Symposium on Theory of Computing (STOC)*, pages 451–460, 2008. doi:10.1145/1374376.1374441.

13 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. Announced at STOC'11. doi:10.1137/11085178X.

14 Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, 1972. doi:10.1145/321694.321699.

15 Michael Elkin. Distributed exact shortest paths in sublinear time. In *Symposium on the Theory of Computing (STOC)*, pages 757–770, 2017. doi:10.1145/3055399.3055452.

16 Michael Elkin, Yuval Emek, Daniel A. Spielman, and Shang-Hua Teng. Lower-stretch spanning trees. *SIAM Journel on Computing*, 38(2):608–628, 2008. Announced at STOC'05. doi:10.1137/050641661.

17 Michael Elkin and Ofer Neiman. Hopsets with constant hopbound, and applications to approximate shortest paths. In *Symposium on Foundations of Computer Science (FOCS)*, pages 128–137, 2016. doi:10.1109/FOCS.2016.22.

18 Michael Elkin and Ofer Neiman. Linear-size hopsets with small hopbound, and distributed routing with low memory. *CoRR*, abs/1704.08468, 2017. https://arxiv.org/abs/1704.08468. URL: https://arxiv.org/abs/1704.08468.

19 Lester R. Ford. Network flow theory. Technical Report P-923, The RAND Corporation, 1956.

20 Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987. Announced at FOCS'84. doi:10.1145/28869.28874.

21 Andrew V. Goldberg. Scaling algorithms for the shortest paths problem. *SIAM Journal on Computing*, 24(3):494–504, 1995. Announced at SODA'93. doi:10.1137/S0097539792231179.

22 Venkatesan Guruswami and Krzysztof Onak. Superlinear lower bounds for multipass graph processing. In *Conference on Computational Complexity (CCC)*, pages 287–298, 2013. doi:10.1109/CCC.2013.37.

23 Thomas Dueholm Hansen, Haim Kaplan, Robert Endre Tarjan, and Uri Zwick. Hollow heaps. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 689–700, 2015. doi:10.1007/978-3-662-47672-7_56.

24 Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Decremental single-source shortest paths on undirected graphs in near-linear total update time. In *Symposium on Foundations of Computer Science (FOCS)*, pages 146–155, 2014. doi:10.1109/FOCS.2014.24.

25 Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. A deterministic almost-tight distributed algorithm for approximating single-source shortest paths. In *Symposium on Theory of Computing (STOC)*, pages 489–498, 2016. doi:10.1145/2897518.2897638.

26 Shang-En Huang and Seth Pettie. Thorup-Zwick emulators are universally optimal hopsets. *CoRR*, abs/1705.00327, 2017. https://arxiv.org/abs/1705.00327. URL: https://arxiv.org/abs/1705.00327.

27 Jonathan A. Kelner, Yin Tat Lee, Lorenzo Orecchia, and Aaron Sidford. An almost-linear-time algorithm for approximate max flow in undirected graphs, and its multicommodity generalizations. In *Symposium on Discrete Algorithms (SODA)*, pages 217–226, 2014. doi:10.1137/1.9781611973402.16.

28 Philip N. Klein and Sairam Subramanian. A randomized parallel algorithm for single-source shortest paths. *Journal of Algorithms*, 25(2):205–220, 1997. Announced at STOC'92. doi:10.1006/jagm.1997.0888.

29 Bernhard Korte and Jens Vygen. *Combinatorial Optimization*. Springer, 2000.

30 François Le Gall. Further algebraic algorithms in the congested clique model and applications to graph-theoretic problems. In *International Symposium on Distributed Computing (DISC)*, pages 57–70, 2016. doi:10.1007/978-3-662-53426-7_5.

31 Yin Tat Lee and Aaron Sidford. Path finding methods for linear programming: Solving linear programs in $\tilde{O}(\sqrt{rank})$ iterations and faster algorithms for maximum flow. In *Symposium on Foundations of Computer Science (FOCS)*, pages 424–433, 2014. doi:10.1109/FOCS.2014.52.

32 Gary L. Miller, Richard Peng, Adrian Vladu, and Shen Chen Xu. Improved parallel algorithms for spanners and hopsets. In *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 192–201, 2015. doi:10.1145/2755573.2755574.

33 Aleksander Mądry. Navigating central path with electrical flows: From flows to matchings, and back. In *Symposium on Foundations of Computer Science (FOCS)*, pages 253–262, 2013. doi:10.1109/FOCS.2013.35.

34 Danupon Nanongkai. Distributed approximation algorithms for weighted shortest paths. In *Symposium on Theory of Computing (STOC)*, pages 565–573, 2014. doi:10.1145/2591796.2591850.

35 James B. Orlin. A faster strongly polynomial minimum cost flow algorithm. *Operations Research*, 41(2):338–350, 1993. Announced at STOC'88. doi:10.1287/opre.41.2.338.

36 Alexander Schrijver. *Combinatorial Optimization*. Springer, 2003.

37 Jonah Sherman. Nearly maximum flows in nearly linear time. In *Symposium on Foundations of Computer Science (FOCS)*, pages 263–269, 2013. doi:10.1109/FOCS.2013.36.

38 Jonah Sherman. Generalized preconditioning and network flow problems. In *Symposium on Discrete Algorithms (SODA)*, pages 772–780, 2017. doi:10.1137/1.9781611974782.49.

39 Hanmao Shi and Thomas H. Spencer. Time-work tradeoffs of the single-source shortest paths problem. *Journal of Algorithms*, 30(1):19–32, 1999. doi:10.1006/jagm.1998.0968.

40 Thomas H. Spencer. Time-work tradeoffs for parallel algorithms. *Journal of the ACM*, 44(5):742–778, 1997. Announced at SODA'91 and SPAA'91. doi:10.1145/265910.265923.

41 Mikkel Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46(3):362–394, 1999. Announced at FOCS'97. doi:10.1145/316542.316548.

Asynchronous Approach in the Plane: A Deterministic Polynomial Algorithm*

Sébastien Bouchard[1], Marjorie Bournat[2], Yoann Dieudonné†[3], Swan Dubois[4], and Franck Petit[5]

1 **Sorbonne Universités, UPMC Univ Paris 06, CNRS, INRIA, LIP6 UMR 7606, Paris, France**
sebastien.bouchard@lip6.fr
2 **Sorbonne Universités, UPMC Univ Paris 06, CNRS, INRIA, LIP6 UMR 7606, Paris, France**
marjorie.bournat@lip6.fr
3 **Laboratoire MIS, Université de Picardie Jules Verne, Amiens, France**
yoann.dieudonne@u-picardie.fr
4 **Sorbonne Universités, UPMC Univ Paris 06, CNRS, INRIA, LIP6 UMR 7606, Paris, France**
swan.dubois@lip6.fr
5 **Sorbonne Universités, UPMC Univ Paris 06, CNRS, INRIA, LIP6 UMR 7606, Paris, France**
franck.petit@lip6.fr

Abstract

In this paper we study the task of *approach* of two mobile agents having the same limited range of vision and moving asynchronously in the plane. This task consists in getting them in finite time within each other's range of vision. The agents execute the same deterministic algorithm and are assumed to have a compass showing the cardinal directions as well as a unit measure. On the other hand, they do not share any global coordinates system (like GPS), cannot communicate and have distinct labels. Each agent knows its label but does not know the label of the other agent or the initial position of the other agent relative to its own. The route of an agent is a sequence of segments that are subsequently traversed in order to achieve approach. For each agent, the computation of its route depends only on its algorithm and its label. An adversary chooses the initial positions of both agents in the plane and controls the way each of them moves along every segment of the routes, in particular by arbitrarily varying the speeds of the agents. Roughly speaking, the goal of the adversary is to prevent the agents from solving the task, or at least to ensure that the agents have covered as much distance as possible before seeing each other. A deterministic approach algorithm is a deterministic algorithm that always allows two agents with any distinct labels to solve the task of approach regardless of the choices and the behavior of the adversary. The cost of a complete execution of an approach algorithm is the length of both parts of route travelled by the agents until approach is completed.

Let Δ and l be the initial distance separating the agents and the length of (the binary representation of) the shortest label, respectively. *Assuming that Δ and l are unknown to both agents, does there exist a deterministic approach algorithm whose cost is polynomial in Δ and l?*

Actually the problem of approach in the plane reduces to the network problem of rendezvous in an infinite oriented grid, which consists in ensuring that both agents end up meeting at the same time at a node or on an edge of the grid. By designing such a rendezvous algorithm with appropriate properties, as we do in this paper, we provide a positive answer to the above question.

* This work was performed within Project ESTATE (Ref. ANR-16-CE25-0009-03), supported by French state funds managed by the ANR (Agence Nationale de la Recherche).
† Partially supported by the European Regional Development Fund (ERDF) and the Picardy region under Project TOREDY.

© Sébastien Bouchard, Marjorie Bournat, Yoann Dieudonné, Swan Dubois, and Franck Petit;
licensed under Creative Commons License CC-BY
31st International Symposium on Distributed Computing (DISC 2017).
Editor: Andréa W. Richa; Article No. 8; pp. 8:1–8:16

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

Our result turns out to be an important step forward from a computational point of view, as the other algorithms allowing to solve the same problem either have an exponential cost in the initial separating distance and in the labels of the agents, or require each agent to know its starting position in a global system of coordinates, or only work under a much less powerful adversary.

1998 ACM Subject Classification G.2.2 Graph Theory, C.2.4 Distributed Systems

Keywords and phrases mobile agents, asynchronous rendezvous, plane, infinite grid, deterministic algorithm, polynomial cost

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.8

1 Introduction

1.1 Model and Problem

The distributed system considered in this paper consists of two *mobile agents* that are initially placed by an adversary at arbitrary but distinct positions in the plane. Both agents have a *limited sensory radius* (in the sequel also referred to as *radius of vision*), the value of which is denoted by ϵ, allowing them to sense (or, to see) all their surroundings at distance at most ϵ from their respective current locations. We assume that the agents know the value of ϵ. As stated in [11], when $\epsilon = 0$, if agents start from arbitrary positions of the plane and can freely move on it, making them occupy the same location at the same time is impossible in a deterministic way. So, we assume that $\epsilon > 0$ and we consider the task of *approach* which consists in bringing them at distance at most ϵ so that they can see each other. In other words, the agents completed their approach once they mutually sense each other and they can even get closer. Without loss of generality, we assume in the rest of this paper that $\epsilon = 1$.

The initial positions of the agents, arbitrarily chosen by the adversary, are separated by a distance Δ that is initially unknown to both agents and that is greater than $\epsilon = 1$. In addition to the initial positions, the adversary also assigns a different non-negative integer (called label) to each agent. The label of an agent is the only input of the deterministic algorithm executed by the agent. While the labels are distinct, the algorithm is the same for both agents. Each agent is equipped with a compass showing the cardinal directions and with a unit of length. The cardinal directions and the unit of length are the same for both agents.

To describe how and where each agent moves, we need to introduce two important notions that are borrowed from [11]: The *route* and the *walk* of an agent. The *route* of an agent is a sequence $(S_1, S_2, S_3 \ldots)$ of segments $S_i = [a_i, a_{i+1}]$ traversed in stages as follows. The route starts from a_1, the initial position of the agent. For every $i \geq 1$, starting from the position a_i, the agent initiates Stage i by choosing a direction α (using its compass) as well as a distance x. Stage i ends as soon as the agent either sees the other agent or reaches a_{i+1} corresponding to the point at distance x from a_i in direction α. Stages are repeated indefinitely (until the approach is completed).

Since both agents never know their positions in a global coordinate system, the directions they choose at each stage can only depend on their (deterministic) algorithm and their labels. So, the route (the actual sequence of segments) followed by an agent depends on its algorithm and its label, but also on its initial position. By contrast, the *walk* of each agent along every segment of its route is controlled by the adversary. More precisely, within each stage S_i and

while the approach is not achieved, the adversary can arbitrarily vary the speed of the agent, stop it and even move it back and forth as long as the walk of the agent is continuous, does not leave S_i, and ends at a_{i+1}. Roughly speaking, the goal of the adversary is to prevent the agents from solving the task, or at least to ensure that the agents have covered as much distance as possible before seeing each other. We assume that at any time an agent can remember the route it has followed since the beginning.

A *deterministic approach algorithm* is a deterministic algorithm that always allows two agents to solve the task of approach regardless of the choices and the behavior of the adversary. The *cost* of an accomplished approach is the length of both parts of route travelled by the agents until they see each other. An approach algorithm is said to be *polynomial* in Δ and in the length of the binary representation of the shortest label between both agents if it always permits to solve the problem of approach at a cost that is polynomial in the two aforementioned parameters, no matter what the adversary does.

It is worth mentioning that the use of distinct labels is not fortuitous. In the absence of a way of distinguishing the agents, the task of approach would have no deterministic solution. This is especially the case if the adversary handles the agents in a perfect synchronous manner. Indeed, if the agents act synchronously and have the same label, they will always follow the same deterministic rules leading to a situation in which the agents will always be exactly at distance Δ from each other.

1.2 Our Results

In this paper, we prove that the task of approach can be solved deterministically in the above asynchronous model, at a cost that is polynomial in the unknown initial distance separating the agents and in the length of the binary representation of the shortest label. To obtain this result, we go through the design of a deterministic algorithm for a very close problem, that of rendezvous in an infinite oriented grid which consists in ensuring that both agents end up meeting either at a node or on an edge of the grid. The tasks of approach and rendezvous are very close as the former can be reduced to the latter.

It should be noticed that our result turns out to be an important advance, from a computational point of view, in resolving the task of approach. Indeed, the other existing algorithms allowing to solve the same problem either have an exponential cost in the initial separating distance and in the labels of the agents [11], or require each agent to know its starting position in a global system of coordinates [9], or only work under a much less powerful adversary [17] which initially assigns a possibly different speed to each agent but cannot vary it afterwards.

1.3 Related Work

The task of approach is closely linked to the task of rendezvous. Historically, the first mention of the rendezvous problem appeared in [32]. From this publication until now, the problem has been extensively studied so that there is henceforth a huge literature about this subject. This is mainly due to the fact that there is a lot of alternatives for the combinations we can make when addressing the problem, *e.g.,* playing on the environment in which the agents are supposed to evolve, the way of applying the sequences of instructions (*i.e.,* deterministic or randomized) or the ability to leave some traces in the visited locations, etc. Naturally, in this paper we focus on work that are related to deterministic rendezvous. This is why we will mostly dwell on this scenario in the rest of this subsection.

However, for the curious reader wishing to consider the matter in greater depth, regarding randomized rendezvous, a good starting point is to go through [2, 27]. Concerning deterministic rendezvous, the literature is divided according to the way of modeling the environnement: Agents can either move in a graph representing a network, or in the plane.

For the problem of rendezvous in networks, a lot of papers considered synchronous settings, *i.e.,* a context where the agents move in the graph in synchronous rounds. This is particularly the case of [16] in which the authors presented a deterministic protocol for solving the rendezvous problem, which guarantees a meeting of the two involved agents after a number of rounds that is polynomial in the size n of the graph, the length l of the shortest of the two labels and the time interval τ between their wake-up times. As an open problem, the authors asked whether it was possible to obtain a polynomial solution to this problem which would be independent of τ. A positive answer to this question was given, independently of each other, in [26] and [34]. While these algorithms ensure rendezvous in polynomial time (*i.e.,* a polynomial number of rounds), they also ensure it at polynomial cost because the cost of a rendezvous protocol in a graph is the number of edges traversed by the agents until they meet – each agent can make at most one edge traversal per round. Note that despite the fact a polynomial time implies a polynomial cost in this context, the reciprocal is not always true as the agents can have very long waiting periods, sometimes interrupted by a movement. Thus these parameters of cost and time are not always linked to each other. This was highlighted in [30] where the authors studied the tradeoffs between cost and time for the deterministic rendezvous problem. More recently, some efforts have been dedicated to analyse the impact on time complexity of rendezvous when in every round the agents are brought with some pieces of information by making a query to some device or some oracle [13, 29]. Along with the work aiming at optimizing the parameters of time and/or cost of rendezvous, some other work have examined the amount of required memory to solve the problem, *e.g.,* [23, 24] for tree networks and in [10] for general networks. In [5], the problem is approached in a fault-prone framework, in which the adversary can delay an agent for a finite number of rounds, each time it wants to traverse an edge of the network.

Rendezvous is the term that is usually used when the task of meeting is restricted to a team of exactly two agents. When considering a team of two agents or more, the term of gathering is commonly used. Still in the context of synchronous networks, we can cite some work about gathering two or more agents. In [18], the task of gathering is studied for anonymous agents while in [4, 14, 19] the same task is studied in presence of byzantine agents that are, roughly speaking, malicious agents with an arbitrary behavior.

Some studies have been also dedicated to the scenario in which the agents move asynchronously in a network [11, 20, 28], *i.e.,* assuming that the agent speed may vary, controlled by the adversary. In [28], the authors investigated the cost of rendezvous for both infinite and finite graphs. In the former case, the graph is reduced to the (infinite) line and bounds are given depending on whether the agents know the initial distance between them or not. In the latter case (finite graphs), similar bounds are given for ring shaped networks. They also proposed a rendezvous algorithm for an arbitrary graph provided the agents initially know an upper bound on the size of the graph. This assumption was subsequently removed in [11]. However, in both [28] and [11], the cost of rendezvous was exponential in the size of the graph. The first rendezvous algorithm working for arbitrary finite connected graphs at cost polynomial in the size of the graph and in the length of the shortest label was presented in [20]. (It should be stressed that the algorithm from [20] cannot be used to obtain the solution described in the present paper: this point is fully explained in the end of this subsection). In all the aforementioned studies, the agents can remember all the actions they have made

since the beginning. A different asynchronous scenario for networks was studied in [12]. In this paper, the authors assumed that agents are oblivious, but they can observe the whole graph and make navigation decisions based on these observations.

Concerning rendezvous or gathering in the plane, we also found the same dichotomy of synchronicity *vs.* asynchronicity. The synchronous case was introduced in [33] and studied from a fault-tolerance point of view in [1, 15, 21]. In [25], rendezvous in the plane is studied for oblivious agents equipped with unreliable compasses under synchronous and asynchronous models. Asynchronous gathering of many agents in the plane has been studied in various settings in [6, 7, 8, 22, 31]. However, the common feature of all these papers related to rendezvous or gathering in the plane – which is not present in our model – is that the agents can observe all the positions of the other agents or at least the global graph of visibility is always connected (*i.e.,* the team cannot be split into two groups so that no agent of the first group can detect at least one agent of the second group).

Finally, the closest works to ours allowing to solve the problem of approach under an asynchronous framework are [9, 3, 11, 17]. In [9, 11, 17], the task of approach is solved by reducing it to the task of rendezvous in an infinite oriented grid. In [3], the authors present a solution to solve the task of approach in a multidimensional space by reducing it to the task of rendezvous in an infinite multidimensional grid. Let us give some more details concerning these four works to highlight the contrasts with our present contribution. The result from [11] leads to a solution to the problem of approach in the plane but has the disadvantage of having an exponential cost. The result from [9] and [3] also implies a solution to the problem of approach in the plane at cost polynomial in the initial distance of the agents. However, in both these works, the authors use the powerful assumption that each agent knows its starting position in a global system of coordinates (while in our paper, the agents are completely ignorant of where they are). Lastly, the result from [17] provides a solution at cost polynomial in the initial distance between agents and in the length of the shortest label. However, the authors of this study also used a powerful assumption: The adversary initially assigns a possibly different and arbitrary speed to each agent but cannot vary it afterwards. Hence, each agent moves at constant speed and uses clock to achieve approach. By contrast, in our paper, we assume basic asynchronous settings, *i.e.,* the adversary arbitrarily and permanently controls the speed of each agent.

To close this subsection, it is worth mentioning that it is unlikely that the algorithm from [20] that we referred to above, which is especially designed for asynchronous rendez-vous in arbitrary finite graphs, could be used to obtain our present result. First, in [20] the algorithm has not a cost polynomial in the initial distance separating the agents and in the length of the smaller label. Actually, ensuring rendezvous at this cost is even impossible in arbitrary graph, as witnessed by the case of the clique with two agents labeled 0 and 1: the adversary can hold one agent at a node and make the other agent traverse $\Theta(n)$ edges before rendezvous, in spite of the initial distance 1. Moreover, the validity of the algorithm given in [20] closely relies on the fact that both agents must evolve in the same finite graph, which is clearly not the case in our present scenario. In particular even when considering the task of rendezvous in an infinite oriented grid, the natural attempt consisting in making each agent apply the algorithm from [20] within bounded grids of increasing size and centered in its initial position, does not permit to claim that rendezvous ends up occurring. Indeed, the bounded grid considered by an agent is never exactly the same than the bounded grid considered by the other one (although they may partly overlap), and thus the agents never evolve in the same finite graph which is a necessary condition to ensure the validity of the solution of [20] and by extension of this natural attempt.

1.4 Roadmap

The next section (Section 2) is dedicated to the computational model and basic definitions. We sketch our solution in Section 3, more formally described in Sections 4. Finally, we make some concluding remarks in Section 5. Due to the lack of space, details on the algorithm, the proofs of correction and cost analysis are omitted but will appear in the journal version of the paper.

2 Preliminaries

We know from [11, 17] that the problem of approach in the plane can be reduced to that of rendezvous in an infinite grid specified in the next paragraph.

Consider an *infinite square grid* in which every node u is adjacent to 4 nodes located North, East, South, and West from node u. We call such a grid a *basic grid*. Two agents with distinct labels (corresponding to non-negative integers) starting from arbitrary and distinct nodes of a basic grid G have to meet either at some node or inside some edge of G. As for the problem of approach (in the plane), each agent is equipped with a compass showing the cardinal directions. The agents can see each other and communicate only when they share the same location in G. In other words, in the basic grid G we assume that the sensory radius (or, radius of vision) of the agents is equal to zero. In such settings, the only initial input that is given to a rendezvous algorithm is the label of the executing agent. When occupying a node u, an agent decides (according to its algorithm) to move to an adjacent node v via one of the four cardinal directions: the movement of the agent along the edge $\{u, v\}$ is controlled by the adversary in the same way as in a section of a route (refer to Subsection 1.1), *i.e.,* the adversary can arbitrarily vary the speed of the agent, stop it and even move it back and forth as long as the walk of the agent is continuous, does not leave the edge, and ends at v.

The *cost* of a rendezvous algorithm in a basic grid is the total number of edge traversals by both agents until their meeting.

From the reduction described in [17], we have the following theorem.

▶ **Theorem 1.** *If there exists a deterministic algorithm solving the problem of rendezvous between any two agents in a basic grid at cost polynomial in D and in the length of the binary representation of the shortest of their labels where D is the distance (in the Manhattan metric) between the two starting nodes occupied by the agents, then there exists a deterministic algorithm solving the problem of approach in the plane between any two agents at cost polynomial in Δ and in the length of the binary representation of the shortest of their labels where Δ is the initial Euclidean distance separating the agents.*

Hence in the rest of the paper we will consider rendezvous in a basic grid, instead of the task of approach. We use N (resp. E, S, W) to denote the cardinal direction North (resp. East, South, West) and an instruction like "Perform NS" means that the agent traverses one edge to the North and then traverses one edge to the South (by the way, coming back to its initial position). We denote by D the initial (Manhattan) distance separating two agents in a basic grid. A route followed by an agent in a basic grid corresponds to a path in the grid (*i.e.,* a sequence of edges $e_1, e_2, e_3, e_4, \ldots$) that are consecutively traversed by the agent until rendezvous is done. For any integer k, we define the *reverse path* to the path $e_1, \ldots, e_k$ as the path $e_k, e_{k-1}, \ldots, e_1 = \overline{e_1}, \ldots, \overline{e_{k-1}}, \overline{e_k}$. We denote by $C(p)$ the number of edge traversals performed by an agent during the execution of a procedure p.

Consider two distinct nodes u and v. We define a specific path from u to v, denoted $P(u,v)$, as follows. If there exists a unique shortest path from u to v, this shorthest path is $P(u,v)$. Otherwise, consider the smallest rectangle $R_{(u,v)}$ such that u and v are two of its corners. $P(u,v)$ is the unique path among the shortest path from u to v that traverses all the edges on the northern side of $R_{(u,v)}$. Note that $P(u,v) = \overline{P(v,u)}$.

3 Idea of the algorithm

3.1 Informal Description in a Nutshell ...

We aim at achieving rendezvous of two asynchronous mobile agents in an infinite grid and in a deterministic way. It is well known that solving rendezvous deterministically is impossible in some symmetric graphs (like a basic grid) unless both agents are given distinct identifiers called labels. We use them to break the symmetry, *i.e.,* in our context, to make the agents follow different routes. The idea is to make each agent "read" its label binary representation, a bit after another from the most to the least significant bits, and for each bit it reads, follow a route depending on the read bit. Our algorithm ensures rendezvous during some of the periods when they follow different routes *i.e.,* when the two agents process two different bits.

Furthermore, to design the routes that both agents will follow, our approach would require to know an upper bound on two parameters, namely the initial distance between the agents and the length (of the binary representation) of the shortest label. As we suppose that the agents have no knowledge of these parameters, they both perform successive "assumptions", in the sequel called *phases*, in order to find out such an upper bound. Roughly speaking, each agent attempts to estimate such an upper bound by successive tests, and for each of these tests, acts as if the upper bound estimation was correct. Both agents first perform Phase 0. When Phase i does not lead to rendezvous, they perform Phase $i+1$, and so on. More precisely, within Phase i, the route of each agent is built in such a way that it ensures rendezvous if 2^i is a good upper bound on the parameters of the problem. Hence, in our approach two requirements are needed: both agents are assumed (1) to process two different bits (*i.e.,* 0 and 1) almost concurrently and (2) to perform Phase $i = \alpha$ almost at the same time – where α is the smallest integer such that the two aforementioned parameters are upper bounded by 2^α.

However, to meet these requirements, we have to face two major issues. First, since the adversary can vary both agent speeds, the idea described above does not prevent the adversary from making the agents always process the same type of bit at the same time. Besides, the route cost depends on the phase number, and thus, if an agent were performing some Phase i with i exponential in the initial distance and in the length of the binary representation of the smallest label, then our algorithm would not be polynomial. To tackle these two issues, we use a mechanism that prevents the adversary from making an agent execute the algorithm arbitrarily faster than the other without meeting. Each of both these issues is circumvent via a specific "synchronization mechanism". Roughly speaking, the first one makes the agents read and process the bits of the binary representation of their labels at quite the same speed, while the second ensures that they start Phase α at almost the same time. This is particularly where our feat of strength is: orchestrating in a subtle manner these synchronizations in a fully asynchronous context while ensuring a polynomial cost. Now that we have described the very high level idea of our algorithm, let us give more details.

3.2 Under the hood

The approach described above allows us to solve rendezvous when there exists an index for which the binary representations of both labels differ. However, this is not always the case especially when a binary representation is a prefix of the other one (e.g., 100 and 1000). Hence, instead of considering its own label, each agent will consider a transformed label: The transformation borrowed from [16] will guarantee the existence of the desired difference over the new labels. In the rest of this description, we assume for convenience that the initial Manhattan distance D separating the agents is at least the length of the shortest binary representation of the two transformed labels (the complementary case adds an unnecessary level of complexity to understand the intuition).

As mentioned previously, our solution (cf. Algorithm 1 in Section 4) works in phases numbered $0, 1, 2, 3, 4, \ldots$ During Phase i (cf. Procedure *Assumption* called at line 3 in Algorithm 1), the agent supposes that the initial distance D is at most 2^i and processes one by one the first 2^i bits of its transformed label: In the case where 2^i is greater than the binary representation of its transformed label, the agent will consider that each of the last "missing" bits is 0. When processing a bit, the agent executes a particular route which depends on the bit value and the phase number. The route related to bit 0 (relying in particular on Procedure *Berry* called at line 9 in Algorithm 2) and the route related to bit 1 (relying in particular on on Procedure *Cloudberry* called at line 11 in Algorithm 2) are obviously different and designed in such a way that if both these routes are executed almost simultaneously by two agents within a phase corresponding to a correct upper bound, then rendezvous occurs by the time any of them has been completed. In the light of this, if we denote by α the smallest integer such that $2^\alpha \geq D$, it turns out that an ideal situation would be that the agents concurrently start phase α and process the bits at quite the same rate within this phase. Indeed, we would then obtain the occurrence of rendezvous by the time the agents complete the process of the jth bit of their transformed label in phase α, where j is the smallest index for which the binary representations of their transformed labels differ. However, getting such an ideal situation in presence of a fully asynchronous adversary appears to be really challenging. This is where the two synchronization mechanisms briefly mentioned above come into the picture.

If the agents start Phase α approximately at the same time, the first synchronization mechanism (cf. Procedure *RepeatSeed* called at line 15 in Algorithm 2) permits to force the adversary to make the agents process their respective bits at similar speed within Phase α, as otherwise rendezvous would occur prematurely during this phase before the process by any agent of the jth bit. This constraint is imposed on the adversary by dividing each bit process into some predefined steps and by ensuring that after each step s of the kth bit process, for any $k \leq 2^\alpha$, an agent follows a specific route that forces the other agent to complete the step s of its kth bit process. This route, on which the first synchronization is based, is constructed by relying on the following simple principle: If an agent performs a given route X included in a given area $\mathcal{S}$ of the basic grid, then the other agent can "*push* it" over X. In other words, unless rendezvous occurs, the agent forces the other to complete its route X by covering $\mathcal{S}$ a number of times at least equal to the number of edge traversals involved in route X (each covering of $\mathcal{S}$ allows to traverse all the edges of $\mathcal{S}$ at least once). Hence, one of the major difficulties we have to face lies in the setting up of the second synchronization mechanism guaranteeing that the agents start Phase α around the same time. At first glance, it might be tempting to use an analogous principle to the one used for dealing with the first synchronization. Indeed, if an agent a_1 follows a route covering r times an area $\mathcal{Y}$ of the grid, such that $\mathcal{Y}$ is where the first $\alpha - 1$ phases of an agent a_2 take place and r is the

maximal number of edge traversals an agent can make during these phases, then agent a_1 pushes agent a_2 to complete its first $\alpha - 1$ phases and to start Phase α. Nevertheless, a strict application of this principle to the case of the second synchronization directly leads to an algorithm having a cost that is superpolynomial in D and the length of the smallest label, due to a cumulative effect that does not appear for the case of the first synchronization. As a consequence, to force an agent to start its Phase α, the second synchronization mechanism does not depend on the kind of route described above, but on a much more complicated route that permits an agent to "*push*" the second one. This works by considering the "pattern" that is drawn on the grid by the second agent rather than just the number of edges that are traversed (cf. Procedure *Harvest* called at line 1 in Algorithm 2). This is the most tricky part of our algorithm, one of the main idea of which relies in particular on the fact that some routes made of an arbitrarily large sequence of edge traversals can be pushed at a relative low cost by some other routes that are of comparatively small length, provided they are judiciously chosen. Let us illustrate this point through the following example. Consider an agent a_1 following from a node v_1 an arbitrarily large sequence of X_i, in which each X_i corresponds either to $A\overline{A}$ or $B\overline{B}$ where A and B are any routes ($\overline{A}$ and $\overline{B}$ corresponding to their respective backtrack *i.e.,* the sequence of edge traversals followed in the reverse order). An agent a_2 starting from an initial node v_2 located at a distance at most d from v_1 can force agent a_1 to finish its sequence of X_i (or otherwise rendezvous occurs), regardless of the number of X_i, simply by executing $A\overline{A}B\overline{B}$ from each node at distance at most d from v_2. To support this claim, let us suppose by contradiction that it does not hold. At some point, agent a_2 necessarily follows $A\overline{A}B\overline{B}$ from v_1. However, note that if either agent starts following $A\overline{A}$ (resp. $B\overline{B}$) from node v_1 while the other is following $A\overline{A}$ (resp. $B\overline{B}$) from node v_1, then the agents meet. Indeed, this implies that the more ahead agent eventually follows $\overline{A}$ (resp. $\overline{B}$) from a node v_3 to v_1 while the other is following A (resp. B) from v_1 to v_3, which leads to rendezvous. Hence, when agent a_2 starts following $B\overline{B}$ from node v_1, agent a_1 is following $A\overline{A}$, and is not in v_1, so that it has at least started the first edge traversal of $A\overline{A}$. This means that when agent a_2 finishes following $A\overline{A}$ from v_1, a_1 is following $A\overline{A}$, which implies, using the same arguments as before, that they meet before either of them completes this route. Hence, in this example, agent a_2 can force a_1 to complete an arbitrarily large sequence of edge traversals with a single and simple route. Actually, our second synchronization mechanism implements this idea (this point is refined in Section 4). This was way the most complicated to set up, as each part of each route in every phase had to be orchestrated very carefully to permit *in fine* this low cost synchronization while still ensuring rendezvous. However, it is through this original and novel way of moving that we finally get the polynomial cost.

4 Formal description of our algorithm and its analysis

The purpose of this section is to give the formal description of our solution and the involved subroutines along with their main objectives and how they work at a high level. The main algorithm that solves the rendezvous in a basic grid is Algorithm RV (shown in Algorithm 1). As mentioned in Section 3, we use the label of an agent only when it has been transformed. Let us describe this transformation that is borrowed from [16]. Let $(b_0 b_1 \dots b_{n-1})$ be the binary representation of the label of an agent. We define its transformed label as the binary sequence $(b_0 b_0 b_1 b_1 \dots b_{n-1} b_{n-1} 01)$. This transformation permits to obtain the following feature: Given two distinct labels, their transformed labels are never prefixes of each other. As explained in the previous section, we need such a feature because our solution requires

Algorithm 1 RV

```
1: d ← 1
2: while agents have not met yet do
3:    Execute Assumption(d)
4:    d ← 2d
5: end while
```

that at some point both agents follow different routes by processing different bit values.

Algorithm RV makes use of a subroutine, *i.e.,* Procedure *Assumption*. When an agent executes this procedure with a parameter α that is a "good" assumption *i.e.,* that upperbounds the initial distance D and the value j of the smallest bit position for which both transformed labels differ, we have the guarantee that rendezvous occurs by the end of this execution. In the rest of this section, we assume that α is the smallest good assumption that upperbounds D and j. The code of Procedure *Assumption* is given in Algorithm 2. It makes use, for technical reasons, of the sequence r that is defined below.

$\forall$ power of two i, $\rho(i) = 2i^4$ and $r(i) = \rho(i) + 3i$

Procedure *Assumption* can be divided into two parts. The first part consists of the execution of Procedure *Harvest* (line 1 of Algorithm 2) and corresponds to the second synchronization mechanism mentioned in Section 3. The main feature of this procedure is the following: when the earlier agent finishes the execution of $Harvest(\alpha)$ within the execution of $Assumption(\alpha)$, we have the guarantee that the later agent has at least started to execute *Assumption* with parameter α (actually, as explained below, we have even the guarantee that most of $Harvest(\alpha)$ has been executed by the later agent). Procedure *Harvest* is presented below. The second part of Procedure *Assumption* (cf. lines $2-19$ of Algorithm 2) consists in processing the bits of the transformed label one by one. More precisely when processing a given bit in a call to Procedure $Assumption(d)$, the agent acts in steps $0, 1, \ldots, 2d(d+1)$: After each of these steps, the agent executes Pattern *RepeatSeed* whose role is described below. In each of these steps, the agent executes *Berry* (resp. *Cloudberry*) if the bit it is processing is 0 (resp. 1). These patterns of moves (cf. Algorithms 5 and 6) are made in such a way that rendezvous occurs by the time any agent finishes the process of its jth bit in $Assumption(\alpha)$ if we have the following synchronization property. Each time any of both agents starts executing a step s during the process of its jth bit in $Assumption(\alpha)$, the other agent has finished the execution of either step $s-1$ in the jth bit process of $Assumption(\alpha)$ if $s > 0$, or the last step of the $(j-1)$th bit process of $Assumption(\alpha)$ if $s = 0$ ($j > 0$ in view of the label transformation given above). To obtain such a synchronization, an agent executes what we called the first synchronization mechanism in the previous section (cf. line 15 in Algorithm 2) after each step of a bit process. Actually, this mechanism relies on procedure *RepeatSeed*, the code of which is given in Algorithm 8. Note that the total number of steps, and thus of executions of *RepeatSeed*, in $Assumption(\alpha)$ is $2\alpha^2(\alpha+1)+\alpha$. For every $0 \leq i \leq 2\alpha^2(\alpha+1)+\alpha$, the ith execution of *RepeatSeed* in $Assumption(\alpha)$ by an agent permits to force the other agent to finish the execution of its ith step in $Assumption(\alpha)$ by repeating a pattern *Seed* (its main purpose is described just above its code given by Algorithm 7): With the appropriate parameters, this pattern *Seed* covers any pattern (*Berry* or *Cloudberry*) made in the ith step of $Assumption(\alpha)$ and the number of times it is repeated is at least the maximal number of edge traversals we can make in the ith step of $Assumption(\alpha)$.

Algorithm 2 *Assumption*(d)

```
1: Execute Harvest(d)
2: radius ← r(d)
3: i ← 1
4: while i ≤ d do
5:    j ← 0
6:    while j ≤ 2d(d + 1) do
7:       // Begin of step j
8:       if the length of the transformed label is strictly greater than i, or its ith bit is 0
         then
9:          Execute Berry(radius, d)
10:      else
11:         Execute Cloudberry(radius, d, d, j)
12:      end if
13:      // End of step j
14:      radius ← radius + 3d
15:      Execute RepeatSeed(radius, C(Cloudberry(radius − 3d, d, d, j)))
16:      j ← j + 1
17:   end while
18:   i ← i + 1
19: end while
```

Algorithm 3 gives the code of Procedure *Harvest*. As in Procedure *Assumption*, it makes use, for technical reasons, of two sequences ρ and r that are defined above. Procedure *Harvest* is made of two parts: the executions of Procedure *PushPattern* (lines $1-3$ of Algorithm 3), and the calls to the patterns *Cloudberry* and *RepeatSeed* (lines $4-5$ of Algorithm 3). When *Harvest* is executed with parameter α (which is a good assumption), the first part ensures that the later agent has at least completed every execution of *Assumption* with a parameter that is smaller than α, while the second part ensures that the later agent has completed almost the entire execution of *Harvest*(α) (more precisely, when the earlier agent finishes the second part, we have the guarantee that it remains for the later agent to execute at most the last line before completing its own execution of *Harvest*(α)).

To give further details on Procedure *Harvest*, let us first describe Procedure *PushPattern* (its code is given in Algorithm 4). When the earlier agent completes the execution of *PushPattern*$(2i, d)$ with i some power of two, assuming that the later agent had already completed *Assumption*(i), we have the guarantee that the later agent has completed its execution of *Assumption*$(2i)$. To ensure this, we regard the execution of *Assumption*$(2i)$ as a sequence of calls to basic patterns (namely *RepeatSeed*, *Berry* and *Cloudberry*), which is formally defined in Definition 2. This sequence is what we meant when talking about "the pattern drawn on the grid" in Section 3. For each basic pattern p_1 in the sequence, the earlier agent executes another pattern p_2 at the end of which we ensure that the later agent has completed p_1. If p_1 is either Pattern *Berry* or Pattern *Cloudberry*, then p_2 is Pattern *RepeatSeed*: we use the same idea here as for the first synchronization mechanism. If p_1 is Pattern *RepeatSeed*, then p_2 is Pattern *Berry*, relying on a property of the route $X\overline{X}$ (with X any non-empty route) introduced in the last paragraph of Subsection 3.2: if both agents follow this route concurrenly from the same node, then they meet. Pattern *Seed* can be seen as such a route, and Procedure *Berry* (whose code is shown in Algorithm 5) consists

Algorithm 3 $Harvest(d)$

1: **for** $i \leftarrow 1$; $i < d$; $i \leftarrow 2i$ **do**
2: Execute $PushPattern(i, d)$
3: **end for**
4: Execute $Cloudberry(\rho(d), d, d, 0)$
5: Execute $RepeatSeed(r(d), C(Cloudberry(\rho(d), d, d, 0)))$

in executing Pattern *Seed* from each node at distance at most α. Hence, unless they meet, the later agent completes its execution of Pattern *RepeatSeed* before the earlier one starts executing *Seed* from the same node. Note that *PushPattern* uses as many patterns as the number of basic patterns in the sequence it is supposed to push: this and the fact of doubling the value of the input parameter of Procedure *Assumption* in Algorithm 1 contribute in particular to keep the polynomiality of our solution.

Thus, once the earlier agent completes the first part of $Harvest(\alpha)$, the later one has at least started the execution of $Assumption(\alpha)$ (and thus of the first part of $Harvest(\alpha)$). At this point, we might think at first glance that we just shifted the problem. Indeed, the number of edge traversals that have to be made to complete all the executions of *Assumption* prior to $Assumption(\alpha)$ is quite the same, if not higher, than the number of edge traversals that have to be made when executing the first part of $Harvest(\alpha)$. Hence the difference between both agents in terms of edge traversals has not been improved here. However, a crucial and decisive progress has nonetheless been done: contrary *a priori* to the series of *Assumption* executed before $Assumption(d)$, the first part of $Harvest(\alpha)$ can be pushed at low cost via the execution of Pattern *Cloudberry* (line 4 of Algorithm 3) by the earlier agent. Actually this pattern corresponds to the kind of route, described at the end of Subsection 3.2 for the second synchronization mechanism, which is of small length compared to the sequence of patterns it can push. Indeed, the first part of $Harvest(\alpha)$ can be viewed as a "large" sequence of Patterns *Seed* and *Berry*: however *Seed* and *Berry* can be seen (by analogy with Subsection 3.2) as routes of the form $A\overline{A}$ and $B\overline{B}$ respectively, while Pattern *Cloudberry* executes *Seed* and *Berry* (i.e., $A\overline{A}B\overline{B}$) once from at least each node at distance at most α.

Note that when the earlier agent has completed the execution of Pattern *Cloudberry* in $Harvest(\alpha)$, the later agent has at least started the execution of Pattern *Cloudberry* in $Harvest(\alpha)$. Hence, there is still a difference between both agents, but it has been considerably reduced: it is now relatively small so that we can handle it pretty easily afterwards.

▶ **Definition 2** (Basic and Perfect Decomposition)**.** Given a call P to an algorithm, we say that the basic decomposition of P, denoted by $\mathcal{BD}(P)$, is P itself if P corresponds to a basic pattern, the type of which belongs to $\{RepeatSeed; Berry; Cloudberry\}$. Otherwise, if during its execution P makes no call then $\mathcal{BD}(P) = \perp$, else $\mathcal{BD}(P) = \mathcal{BD}(x_1), \mathcal{BD}(x_2), \ldots, \mathcal{BD}(x_n)$ where $x_1, x_2, \ldots, x_n$ is the sequence (in the order of execution) of all the calls in P that are children of P. We say that $\mathcal{BD}(P)$ is a perfect decomposition if it does not contain any $\perp$.

▶ Remark. The basic decomposition of every call to Procedure *Assumption* is perfect.

Starting from a node v, the main purpose of $Seed(x)$ is to visit all nodes of the grid at distance at most x from v and to traverse all edges of the grid linking two nodes at distance at most x from v (informally, the procedure permits to cover an area of radius x).

The following two theorems state the validity and the polynomial cost of Algorithm RV (their proofs will appear in the journal version of the paper).

Algorithm 4 $PushPattern(i, d)$

for each p in $\mathcal{BD}(Assumption(i))$ **do**
  if p is a call to pattern $RepeatSeed$ with value x as first parameter **then**
    Execute $Berry(x, d)$
  else
    /* pattern p is either a call to pattern $Berry$ or a call to pattern $Cloudberry$ (in view of the above remark) and has at least two parameters */
    Let x (resp. y) be the first (resp. the second) parameter of p
    Execute $RepeatSeed(d + x + 2y, C(Cloudberry(x, y, y, 0)))$
  end if
end for

Algorithm 5 Pattern $Berry(x, y)$

/* First period */
Let u be the current node
for $i \leftarrow 1$; $i \leq x + y$; $i \leftarrow i + 1$ **do**
  for $j \leftarrow 0$; $j \leq i$; $j \leftarrow j + 1$ **do**
    for $k \leftarrow 0$; $k \leq j$; $k \leftarrow k + 1$ **do**
      for each node v at distance k from u ordered clockwise from the North **do**
        Follow $P(u, v)$
        Execute $Seed(i - j)$
        Follow $P(v, u)$
      end for
    end for
  end for
end for
/* Second period */
$L \leftarrow$ the path followed by the agent during the first period
Backtrack by following the reverse path $\overline{L}$

Algorithm 6 Pattern $Cloudberry(x, y, z, h)$

/* First period */
Let u be the current node
Let U be the list of nodes at distance at most z from u ordered in the order of the first visit when applying $Seed(z)$ from node u
for $i \leftarrow 0$; $i \leq 2z(z + 1)$; $i \leftarrow i + 1$ **do**
  Let v be the node with index $h + i \pmod{2z(z + 1) + 1}$ in U
  Follow $P(u, v)$
  Execute $Seed(x)$
  Execute $Berry(x, y)$
  Follow $P(v, u)$
end for
/* Second period */
$L \leftarrow$ the path followed by the agent during the first period
Backtrack by following the reverse path $\overline{L}$

Algorithm 7 Pattern $RepeatSeed(x, n)$

Execute n times Pattern $Seed(x)$

Algorithm 8 Pattern $Seed(x)$

/* First period */
for $i \leftarrow 1$; $i \leq x$; $i \leftarrow i + 1$ **do**
  /* Phase i */
  Perform $(N(SE)^i(WS)^i(NW)^i(EN)^i)$
end for
/* Second period */
$L \leftarrow$ the path followed by the agent during the first period
Backtrack by following the reverse path $\overline{L}$

▶ **Theorem 3.** *Algorithm RV solves the problem of rendezvous in the basic grid.*

▶ **Theorem 4.** *The cost of Algorithm RV is polynomial in D and l, where D is the initial (Manhattan) distance separating both agents and l is the length of the shortest label.*

5 Conclusion

From Theorems 1, 3 and 4, we obtain the following result concerning the task of approach.

▶ **Theorem 5.** *The task of approach can be solved at cost polynomial in the unknown initial distance Δ separating the agents and in the length of (the binary representation) of the shortest of their labels.*

Throughout the paper, we made no attempt at optimizing the cost. Actually, as the acute reader will have noticed, our main concern was only to prove the polynomiality. Hence, a natural open problem is to find out the optimal cost to solve the task of approach. This would be all the more important as in turn we could compare this optimal cost with the cost of solving the same task with agents that can position themselves in a global system of coordinates (the almost optimal cost for this case is given in [9]) in order to determine whether the use of such a system is finally relevant to minimize the travelled distance.

References

1 Noa Agmon and David Peleg. Fault-tolerant gathering algorithms for autonomous mobile robots. *SIAM J. Comput.*, 36(1):56–82, 2006.

2 Steve Alpern and Shmuel Gal. *The theory of search games and rendezvous.* Int. Series in Operations Research and Management Science, Kluwer Academic Publishers, 2003.

3 Evangelos Bampas, Jurek Czyzowicz, Leszek Gasieniec, David Ilcinkas, and Arnaud Labourel. Almost optimal asynchronous rendezvous in infinite multidimensional grids. In *Distributed Computing, 24th International Symposium, DISC 2010, Proceedings*, pages 297–311, 2010.

4 Sébastien Bouchard, Yoann Dieudonné, and Bertrand Ducourthial. Byzantine gathering in networks. *Distributed Computing*, 29(6):435–457, 2016.

5 Jérémie Chalopin, Yoann Dieudonné, Arnaud Labourel, and Andrzej Pelc. Rendezvous in networks in spite of delay faults. *Distributed Computing*, 29(3):187–205, 2016.

6 Mark Cieliebak, Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro. Distributed computing by mobile robots: Gathering. *SIAM J. Comput.*, 41(4):829–879, 2012.

7 Reuven Cohen and David Peleg. Convergence properties of the gravitational algorithm in asynchronous robot systems. *SIAM J. Comput.*, 34(6):1516–1528, 2005.

8 Reuven Cohen and David Peleg. Convergence of autonomous mobile robots with inaccurate sensors and movements. *SIAM J. Comput.*, 38(1):276–302, 2008.

9 Andrew Collins, Jurek Czyzowicz, Leszek Gasieniec, and Arnaud Labourel. Tell me where I am so I can meet you sooner. In *Automata, Languages and Programming, 37th International Colloquium, ICALP 2010, Proceedings, Part II*, pages 502–514, 2010.

10 Jurek Czyzowicz, Adrian Kosowski, and Andrzej Pelc. How to meet when you forget: log-space rendezvous in arbitrary graphs. *Distributed Computing*, 25(2):165–178, 2012.

11 Jurek Czyzowicz, Andrzej Pelc, and Arnaud Labourel. How to meet asynchronously (almost) everywhere. *ACM Transactions on Algorithms*, 8(4):37, 2012.

12 Gianlorenzo D'Angelo, Gabriele Di Stefano, and Alfredo Navarra. Gathering on rings under the look-compute-move model. *Distributed Computing*, 27(4):255–285, 2014.

13 Shantanu Das, Dariusz Dereniowski, Adrian Kosowski, and Przemyslaw Uznanski. Rendezvous of distance-aware mobile agents in unknown graphs. In *Structural Information and Communication Complexity - 21st International Colloquium, SIROCCO 2014, Proceedings*, pages 295–310, 2014.

14 Shantanu Das, Flaminia L. Luccio, and Euripides Markou. Mobile agents rendezvous in spite of a malicious agent. In *Algorithms for Sensor Systems - 11th International Symposium on Algorithms and Experiments for Wireless Sensor Networks, ALGOSENSORS 2015, Revised Selected Papers*, pages 211–224, 2015.

15 Xavier Défago, Maria Gradinariu, Stéphane Messika, and Philippe Raipin Parvédy. Fault-tolerant and self-stabilizing mobile robots gathering. In *Distributed Computing, 20th International Symposium, DISC 2006, Proceedings*, pages 46–60, 2006.

16 Anders Dessmark, Pierre Fraigniaud, Dariusz R. Kowalski, and Andrzej Pelc. Deterministic rendezvous in graphs. *Algorithmica*, 46(1):69–96, 2006.

17 Yoann Dieudonné and Andrzej Pelc. Deterministic polynomial approach in the plane. *Distributed Computing*, 28(2):111–129, 2015.

18 Yoann Dieudonné and Andrzej Pelc. Anonymous meeting in networks. *Algorithmica*, 74(2):908–946, 2016.

19 Yoann Dieudonné, Andrzej Pelc, and David Peleg. Gathering despite mischief. *ACM Transactions on Algorithms*, 11(1):1, 2014.

20 Yoann Dieudonné, Andrzej Pelc, and Vincent Villain. How to meet asynchronously at polynomial cost. *SIAM J. Comput.*, 44(3):844–867, 2015.

21 Yoann Dieudonné and Franck Petit. Self-stabilizing gathering with strong multiplicity detection. *Theor. Comput. Sci.*, 428:47–57, 2012.

22 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.

23 Pierre Fraigniaud and Andrzej Pelc. Deterministic rendezvous in trees with little memory. In *Distributed Computing, 22nd International Symposium, DISC 2008, Proceedings*, pages 242–256, 2008.

24 Pierre Fraigniaud and Andrzej Pelc. Delays induce an exponential memory gap for rendezvous in trees. *ACM Transactions on Algorithms*, 9(2):17, 2013.

25 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.

26 Dariusz R. Kowalski and Adam Malinowski. How to meet in anonymous network. *Theor. Comput. Sci.*, 399(1-2):141–156, 2008.

27 Evangelos Kranakis, Danny Krizanc, and Sergio Rajsbaum. Mobile agent rendezvous: A survey. In *Structural Information and Communication Complexity, 13th International Colloquium, SIROCCO 2006, Proceedings*, pages 1–9, 2006.

28 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.

29 Avery Miller and Andrzej Pelc. Fast rendezvous with advice. *Theor. Comput. Sci.*, 608:190–198, 2015.

30 Avery Miller and Andrzej Pelc. Time versus cost tradeoffs for deterministic rendezvous in networks. *Distributed Computing*, 29(1):51–64, 2016.

31 Linda Pagli, Giuseppe Prencipe, and Giovanni Viglietta. Getting close without touching: near-gathering for autonomous mobile robots. *Distributed Computing*, 28(5):333–349, 2015.

32 Thomas Schelling. *The Strategy of Conflict*. Oxford University Press, Oxford, 1960.

33 Ichiro Suzuki and Masafumi Yamashita. Distributed anonymous mobile robots: Formation of geometric patterns. *SIAM J. Comput.*, 28(4):1347–1363, 1999.

34 Amnon Ta-Shma and Uri Zwick. Deterministic rendezvous, treasure hunts, and strongly universal exploration sequences. *ACM Transactions on Algorithms*, 10(3):12, 2014.

Cost of Concurrency in Hybrid Transactional Memory

Trevor Brown[*1] **and Srivatsan Ravi**[2]

1 **Technion, Israel Institute of Technology, Haifa, Israel**
me@tbrown.pro
2 **University of Southern California, Los Angeles, California, USA**
srivatsr@usc.edu

Abstract

State-of-the-art *software transactional memory (STM)* implementations achieve good performance by carefully avoiding the overhead of *incremental validation* (i.e., re-reading previously read data items to avoid inconsistency) while still providing *progressiveness* (allowing transactional aborts only due to *data conflicts*). Hardware transactional memory (HTM) implementations promise even better performance, but offer no progress guarantees. Thus, they must be combined with STMs, leading to *hybrid* TMs (HyTMs) in which hardware transactions must be *instrumented* (i.e., access metadata) to detect contention with software transactions.

We show that, unlike in progressive STMs, software transactions in progressive HyTMs cannot avoid incremental validation. In fact, this result holds even if hardware transactions can *read* metadata *non-speculatively*. We then present *opaque* HyTM algorithms providing *progressiveness for a subset of transactions* that are optimal in terms of hardware instrumentation. We explore the concurrency vs. hardware instrumentation vs. software validation trade-offs for these algorithms. Our experiments with Intel and IBM POWER8 HTMs seem to suggest that (i) the *cost of concurrency* also exists in practice, (ii) it is important to implement HyTMs that provide progressiveness for a maximal set of transactions without incurring high hardware instrumentation overhead or using global contending bottlenecks and (iii) there is no easy way to derive more efficient HyTMs by taking advantage of non-speculative accesses within hardware.

1998 ACM Subject Classification D.1.3 Concurrent Programming – parallel programming

Keywords and phrases Transactional memory, Lower bounds, Opacity

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.9

1 Introduction

The *Transactional Memory (TM)* abstraction is a synchronization mechanism that allows the programmer to *optimistically* execute sequences of shared-memory operations as *atomic transactions*. Several software TM designs [8, 24, 13, 11] have been introduced subsequent to the original TM proposal based in hardware [14]. The original dynamic STM implementation DSTM [13] ensures that a transaction aborts only if there is a read-write *data conflict* with a concurrent transaction (à la *progressiveness* [12]). However, to satisfy *opacity* [12], read operations in DSTM must *incrementally* validate the responses of all previous read operations to avoid inconsistent executions. This results in quadratic (in the size of the transaction's read set) step-complexity for transactions. Subsequent STM implementations like NOrec [8]

* Trevor Brown received funding for this work from the Natural Sciences and Engineering Research Council of Canada.


© Trevor Brown and Srivatsan Ravi;
licensed under Creative Commons License CC-BY

and TL2 [10] minimize the impact on performance due to incremental validation. NOrec uses a global sequence lock that is read at the start of a transaction and performs *value-based* validation during read operations only if the value of the global lock has been changed (by an updating transaction) since reading it. TL2, on the other hand, eliminates incremental validation completely. Like NOrec, it uses a global sequence lock, but each data item also has an associated sequence lock value that is updated alongside the data item. When a data item is read, if its associated sequence lock value is different from the value that was read from the sequence lock at the start of the transaction, then the transaction aborts.

In fact, STMs like TL2 and NOrec ensure progress in the absence of data conflicts with O(1) step complexity read operations and *invisible reads* (read operations which do not modify shared memory). Nonetheless, TM designs that are implemented entirely in software still incur significant performance overhead. Thus, current CPUs have included instructions to mark a block of memory accesses as transactional [1, 17], allowing them to be executed *atomically* in hardware. Hardware transactions promise better performance than STMs, but they offer no progress guarantees since they may experience *spurious* aborts. This motivates the need for *hybrid* TMs in which the *fast* hardware transactions are complemented with *slower* software transactions that do not have spurious aborts.

To allow hardware transactions in a HyTM to detect conflicts with software transactions, hardware transactions must be *instrumented* to perform additional metadata accesses, which introduces overhead. Hardware transactions typically provide automatic conflict detection at cacheline granularity, thus ensuring that a transaction will be aborted if it experiences memory contention on a cacheline. This is at least the case with Intel's Transactional Synchronization Extensions [25]. The IBM POWER8 architecture additionally allows hardware transactions to access metadata *non-speculatively*, thus bypassing automatic conflict detection. While this has the advantage of potentially reducing contention aborts in hardware, this makes the design of HyTM implementations potentially harder to prove correct.

In [3], it was shown that hardware transactions in opaque progressive HyTMs must perform at least one metadata access per transactional read and write. In this paper, we show that in opaque progressive HyTMs with invisible reads, software transactions *cannot* avoid incremental validation. Specifically, we prove that *each read operation* of a software transaction in a progressive HyTM must necessarily incur a validation cost that is *linear* in the size of the transaction's read set. This is in contrast to TL2 which is progressive and has constant complexity read operations. Thus, in addition to the linear instrumentation cost in hardware transactions, there is a quadratic step complexity cost in software transactions.

We then present opaque HyTM algorithms providing *progressiveness for a subset of transactions* that are optimal in terms of hardware instrumentation. Algorithm 1 is progressive for all transactions, but it incurs high instrumentation overhead in practice. Algorithm 2 avoids all instrumentation in fast-path read operations, but is progressive only for slow-path reading transactions. We also sketch how *some* hardware instrumentation can be performed *non-speculatively* without violating opacity.

Extensive experiments were performed to characterize the *cost of concurrency* in practice. We studied the instrumentation-optimal algorithms, as well as TL2, Transactional Lock Elision (TLE) [22] and Hybrid NOrec [23] on both Intel and IBM POWER architectures. Each of the algorithms we studied contributes to an improved understanding of the concurrency vs. hardware instrumentation vs. software validation trade-offs for HyTMs. Comparing results between the very different Intel and IBM POWER architectures also led to new insights. Collectively, our results suggest the following.

(i) The *cost of concurrency* is significant in practice; high hardware instrumentation impacts performance negatively on Intel and much more so on POWER8 due to its limited transactional cache capacity.

(ii) It is important to implement HyTMs that provide progressiveness for a maximal set of transactions without incurring high hardware instrumentation overhead or using global contending bottlenecks.

(iii) There is no easy way to derive more efficient HyTMs by taking advantage of non-speculative accesses supported within the fast-path in POWER8 processors.

2 Hybrid transactional memory (HyTM)

Transactional memory (TM). A *transaction* is a sequence of *transactional operations* (or *t-operations*), reads and writes, performed on a set of *transactional objects* (*t-objects*). A TM *implementation* provides a set of concurrent *processes* with deterministic algorithms that implement reads and writes on t-objects using a set of *base objects*.

Configurations and executions. A *configuration* of a TM implementation specifies the state of each base object and each process. In the *initial* configuration, each base object has its initial value and each process is in its initial state. An *event* (or *step*) of a transaction invoked by some process is an invocation of a t-operation, a response of a t-operation, or an atomic *primitive* operation applied to base object along with its response. An *execution fragment* is a (finite or infinite) sequence of events $E = e_1, e_2, \ldots$. An *execution* of a TM implementation $\mathcal{M}$ is an execution fragment where, informally, each event respects the specification of base objects and the algorithms specified by $\mathcal{M}$.

For any finite execution E and execution fragment E', $E \cdot E'$ denotes the concatenation of E and E', and we say that $E \cdot E'$ is an *extension* of E. For every transaction identifier k, $E|k$ denotes the subsequence of E restricted to events of transaction T_k. If $E|k$ is non-empty, we say that T_k *participates* in E, Let $txns(E)$ denote the set of transactions that participate in E. Two executions E and E' are *indistinguishable* to a set $\mathcal{T}$ of transactions, if for each transaction $T_k \in \mathcal{T}$, $E|k = E'|k$. A transaction $T_k \in txns(E)$ is *complete in* E if $E|k$ ends with a response event. The execution E is *complete* if all transactions in $txns(E)$ are complete in E. A transaction $T_k \in txns(E)$ is *t-complete* if $E|k$ ends with A_k or C_k; otherwise, T_k is *t-incomplete*. We consider the dynamic programming model: the *read set* (resp., the *write set*) of a transaction T_k in an execution E, denoted $Rset_E(T_k)$ (resp., $Wset_E(T_k)$), is the set of t-objects that T_k attempts to read (and resp. write) by issuing a t-read (resp., t-write) invocation in E (for brevity, we sometimes omit the subscript E).

We assume that base objects are accessed with *read-modify-write* (rmw) primitives. A rmw primitive event on a base object is *trivial* if, in any configuration, its application does not change the state of the object. Otherwise, it is called *nontrivial*. Events e and e' of an execution E *contend* on a base object b if they are both primitives on b in E and at least one of them is nontrivial.

Hybrid transactional memory executions. We now describe the execution model of a *Hybrid transactional memory (HyTM)* implementation. In our model, shared memory configurations may be modified by accessing base objects via two kinds of primitives: *direct* and *cached*.

(i) In a direct (also called non-speculative) access, the rmw primitive operates on the memory state: the direct-access event atomically reads the value of the object in the shared memory and, if necessary, modifies it.

(ii) In a cached access performed by a process i, the rmw primitive operates on the *cached* state recorded in process i's *tracking set* τ_i.

More precisely, τ_i is a set of triples (b, v, m) where b is a base object identifier, v is a value, and $m \in \{shared, exclusive\}$ is an access *mode*. The triple (b, v, m) is added to the tracking set when i performs a cached rmw access of b, where m is set to *exclusive* if the access is nontrivial, and to *shared* otherwise. We assume that there exists some constant TS such that the condition $|\tau_i| \leq TS$ must always hold; this condition will be enforced by our model. A base object b is *present* in τ_i with mode m if $\exists v, (b, v, m) \in \tau_i$.

Hardware aborts. A tracking set can be *invalidated* by a concurrent process: if, in a configuration C where $(b, v, exclusive) \in \tau_i$ (resp., $(b, v, shared) \in \tau_i$), a process $j \neq i$ applies any primitive (resp., any *nontrivial* primitive) to b, then τ_i becomes *invalid* and any subsequent event invoked by i sets τ_i to $\emptyset$ and returns $\perp$. We refer to this event as a *tracking set abort*.

Any transaction executed by a *correct* process that performs at least one cached access must necessarily perform a *cache-commit* primitive that determines the terminal response of the transaction. A cache-commit primitive issued by process i with a valid τ_i does the following: for each base object b such that $(b, v, exclusive) \in \tau_i$, the value of b in C is updated to v. Finally, τ_i is set to $\emptyset$ and the operation returns *commit*. We assume that a fast-path transaction T_k returns A_k as soon a cached primitive or *cache-commit* returns $\perp$.

Slow-path and fast-path transactions. We partition HyTM transactions into *fast-path transactions* and *slow-path transactions*. A slow-path transaction models a regular software transaction. An event of a slow-path transaction is either an invocation or response of a t-operation, or a direct rmw primitive on a base object. A fast-path transaction essentially encapsulates a hardware transaction. Specifically, in any execution E, we say that a transaction $T_k \in txns(E)$ is a fast-path transaction if $E|k$ contains at least one cached event. An event of a *hardware transaction* is either an invocation or response of a t-operation, or a direct trivial access or a cached access, or a cache-commit primitive.

▶ Remark (Tracking set aborts). Let $T_k \in txns(E)$ be any t-incomplete fast-path transaction executed by process i, where $(b, v, exclusive) \in \tau_i$ (resp., $(b, v, shared) \in \tau_i$) after execution E, and e be any event (resp., nontrivial event) that some process $j \neq i$ is poised to apply after E. The next event of T_k in any extension of $E \cdot e$ is A_k.

▶ Remark (Capacity aborts). Any cached access performed by a process i executing a fast-path transaction T_k; $|Dset(T_k)| > 1$ first checks the condition $|\tau_i| = TS$, where TS is a pre-defined constant, and if so, it sets $\tau_i = \emptyset$ and immediately returns $\perp$.

Direct reads within fast-path. Note that we specifically allow hardware transactions to perform reads without adding the corresponding base object to the process' tracking set, thus modeling the *suspend/resume* instructions supported by IBM POWER8 architectures. Note that Intel's HTM does not support this feature: an event of a fast-path transaction does not include any direct access to base objects.

HyTM properties. We consider the TM-correctness property of *opacity* [12]: an execution E is opaque if there exists a *legal* (every t-read of a t-object returns the value of its latest committed t-write) sequential execution S equivalent to some t-completion of E that respects the *real-time ordering* of transactions in E. We also assume a weak *TM-liveness* property for t-operations: every t-operation returns a matching response within a finite number of its own steps if running step-contention free from a configuration in which every other transaction is

t-complete. Moreover, we focus on HyTMs that provide *invisible reads*: t-read operations do not perform nontrivial primitives in any execution.

3 Progressive HyTM must perform incremental validation

In this section, we show that it is impossible to implement opaque *progressive* HyTMs with *invisible reads* with $O(1)$ step-complexity read operations for slow-path transactions. This result holds even if fast-path transactions may perform direct trivial accesses.

Formally, we say that a HyTM implementation $\mathcal{M}$ is progressive for a set $\mathcal{T}$ of transactions if in any execution E of $\mathcal{M}$; $\mathcal{T} \subseteq txns(E)$, if any transaction $T_k \in \mathcal{T}$ returns A_k in E, there exists another concurrent transaction T_m that *conflicts* (both access the same t-object and at least one writes) with T_k in E [12].

We construct an execution of a progressive opaque HyTM in which every t-read performed by a read-only slow-path transaction must access linear (in the size of the read set) number of distinct base objects.

▶ **Theorem 1.** *Let $\mathcal{M}$ be any progressive opaque HyTM implementation providing invisible reads. There exists an execution E of $\mathcal{M}$ and some slow-path read-only transaction $T_k \in txns(E)$ that incurs a time complexity of $\Omega(m^2)$; $m = |Rset(T_k)|$.*

Proof sketch. We construct an execution of a read-only slow-path transaction T_ϕ that performs $m \in \mathbb{N}$ distinct t-reads of t-objects $X_1, \ldots, X_m$. We show inductively that for each $i \in \{1, \ldots, m\}$; $m \in \mathbb{N}$, the i^{th} t-read must access $i-1$ distinct base objects during its execution. The (partial) steps in our execution are depicted in Figure 1.

For each $i \in \{1, \ldots, m\}$, $\mathcal{M}$ has an execution of the form depicted in Figure 1b. Start with the complete step contention-free execution of slow-path read-only transaction T_ϕ that performs $(i-1)$ t-reads: $read_\phi(X_1) \cdots read_\phi(X_{i-1})$, followed by the t-complete step contention-free execution of a fast-path transaction T_i that writes $nv_i \neq v_i$ to X_i and commits and then the complete step contention-free execution fragment of T_ϕ that performs its i^{th} t-read: $read_\phi(X_i) \to nv_i$. Indeed, by progressiveness, T_i cannot incur tracking set aborts and since it accesses only a single t-object, it cannot incur capacity aborts. Moreover, in this execution, the t-read of X_i by slow-path transaction T_ϕ must return the value nv written by fast-path transaction T_i since this execution is indistinguishable to T_ϕ from the execution in Figure 1a.

We now construct $(i-1)$ different executions of the form depicted in Figure 1c: for each $\ell \leq (i-1)$, a fast-path transaction T_ℓ (preceding T_i in real-time ordering, but invoked following the $(i-1)$ t-reads by T_ϕ) writes $nv_\ell \neq v$ to X_ℓ and commits, followed by the t-read of X_i by T_ϕ. Observe that, T_ℓ and T_i which access mutually disjoint data sets cannot contend on each other since if they did, they would concurrently contend on some base object and incur a tracking set abort, thus violating progressiveness. Indeed, by the TM-liveness property we assumed (cf. Section 2) and invisible reads for T_ϕ, each of these $(i-1)$ executions exist.

In each of these $(i-1)$ executions, the final t-read of X_i cannot return the new value nv: the only possible serialization for transactions is T_ℓ, T_i, T_ϕ; but the $read_\phi(X_\ell)$ performed by T_k that returns the initial value v is not legal in this serialization—contradiction to the assumption of opacity. In other words, slow-path transaction T_ϕ is forced to verify the validity of t-objects in $Rset(T_\phi)$. Finally, we note that, for all $\ell, \ell' \leq (i-1); \ell' \neq \ell$, fast-path transactions T_ℓ and $T_{\ell'}$ access mutually disjoint sets of base objects thus forcing the t-read of X_i to access least $i-1$ different base objects in the worst case. Consequently, for all

Table 1 Table summarizing complexities of HyTM implementations.

	Algorithm 1	Algorithm 2	TLE	HybridNOrec
Instrumentation in fast-path reads	per-read	constant	constant	constant
Instrumentation in fast-path writes	per-write	per-write	constant	constant
Validation in slow-path reads	$\Theta(\|Rset\|)$	$O(\|Rset\|)$	none	$O(\|Rset\|)$, but validation only if concurrency
h/w-s/f concurrency	prog.	prog. for slow-path readers	zero	not prog., but small contention window
Direct accesses inside fast-path	yes	no	no	yes
opacity	yes	yes	yes	yes

$i \in \{2, \ldots, m\}$, slow-path transaction T_ϕ must perform at least $i-1$ steps while executing the i^{th} t-read in such an execution. ◀

3.1 How STM implementations mitigate the quadratic lower bound step complexity

NOrec [8] is a progressive opaque STM that minimizes the average step-complexity resulting from incremental validation of t-reads. Transactions read a global versioned lock at the start, and perform value-based validation during t-read operations *iff* the global version has changed. TL2 [10] improves over NOrec by circumventing the lower bound of Theorem 1. Concretely, TL2 associates a global version with each t-object updated during a transaction and performs validation with O(1) complexity during t-reads by simply verifying if the version of the t-object is greater than the global version read at the start of the transaction. Technically, NOrec and algorithms in this paper provide a stronger definition of progressiveness: a transaction may abort only if there is a prefix in which it conflicts with another transaction and both are t-incomplete. TL2 on the other hand allows a transaction to abort due to a concurrent conflicting transaction.

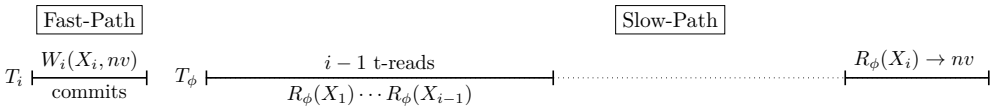
3.2 Implications for disjoint-access parallelism in HyTM

The property of disjoint-access parallelism (DAP), in its *weakest* form, ensures that two transactions concurrently contend on the same base object only if their data sets are connected in the *conflict graph*, capturing data-set overlaps among all concurrent transactions [4]. It is well known that weak DAP STMs with invisible reads must perform incremental validation even if the required TM-progress condition requires transactions to commit only in the absence of any concurrent transaction [12, 16]. For example, DSTM [13] is a weak DAP STM that is progressive and consequently incurs the validation complexity. On the other hand, TL2 and NOrec are not weak DAP since they employ a global versioned lock that mitigates the cost of incremental validation, but this allows two transactions accessing disjoint data sets to concurrently contend on the same memory location. Indeed, this inspires the proof of Theorem 1.

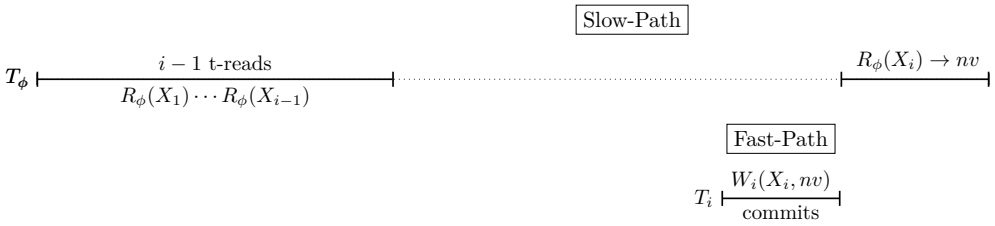
4 Hybrid transactional memory algorithms

4.1 Instrumentation-optimal progressive HyTM

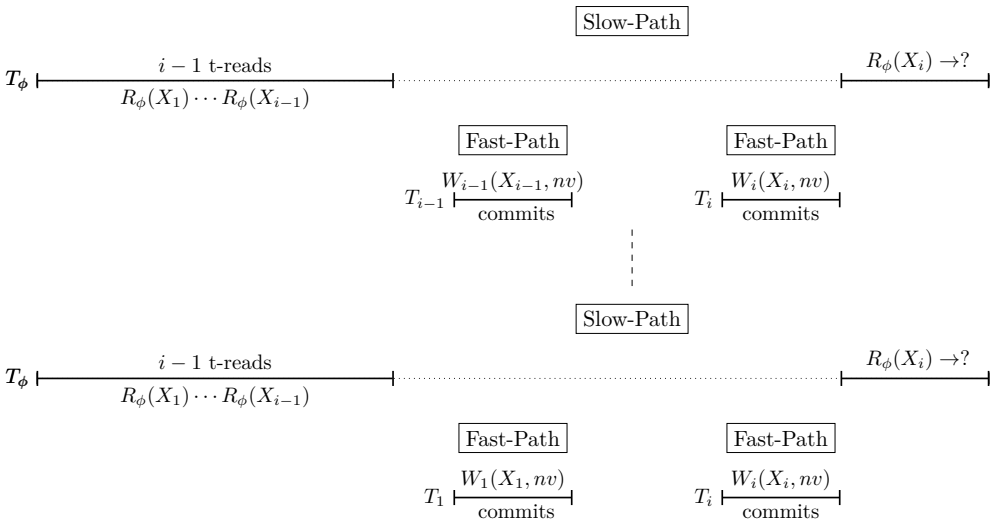
We describe a HyTM algorithm that is a tight bound for Theorem 1 and the instrumentation cost on the fast-path transactions established in [3]. Pseudocode appears in Algorithm 1. For each t-object X_j, our implementation maintains a base object v_j that stores X_j's *value* and a *sequence lock* r_j.



(a) Slow-path transaction T_ϕ performs $i-1$ distinct t-reads (each returning the initial value) followed by the t-read of X_i that returns value nv written by fast-path transaction T_i.



(b) Fast-path transaction T_i does not contend with any of the $i-1$ t-reads performed by T_ϕ and must be committed in this execution since it cannot incur a tracking set or capacity abort. The t-read of X_i must return nv because this execution is indistinguishable to T_ϕ from 1a.



(c) In each of these each $i-1$ executions, fast-path transactions cannot incur a tracking set or capacity abort. By opacity, the t-read of X_i by T_ϕ cannot return new value nv. Therefore, to distinguish the $i-1$ different executions, t-read of X_i by slow-path transaction T_ϕ is forced to access $i-1$ different base objects.

Figure 1 Proof steps for Theorem 1.

Fast-path transactions: For a fast-path transaction T_k executed by process p_i, the $read_k(X_j)$ implementation first reads r_j (direct) and returns A_k if some other process p_j holds a lock on X_j. Otherwise, it returns the value of X_j. As with $read_k(X_j)$, the $write(X_j, v)$ implementation returns A_k if some other process p_j holds a lock on X_j; otherwise process p_i increments the sequence lock r_j. If the cache has not been invalidated, p_i updates the shared memory during $tryC_k$ by invoking the *commit-cache* primitive.

Slow-path read-only transactions: Any $read_k(X_j)$ invoked by a slow-path transaction first reads the value of the t-object from v_j, adds r_j to $Rset(T_k)$ if its not held by a concurrent

transaction and then performs *validation* on its entire read set to check if any of them have been modified. If either of these conditions is true, the transaction returns A_k. Otherwise, it returns the value of X_j. Validation of the read set is performed by re-reading the values of the sequence lock entries stored in $Rset(T_k)$.

Slow-path updating transactions: An updating slow-path transaction T_k attempts to obtain exclusive write access to its entire write set. If all the locks on the write set were acquired successfully, T_k performs validation of the read set and if successful, updates the values of the t-objects in shared memory, releases the locks and returns C_k; else p_i aborts the transaction.

Direct accesses inside fast-path: Note that opacity is not violated even if the sequence lock accesses during t-read may be performed directly without incurring tracking set aborts.

4.2 Instrumentation-optimal HyTM that is progressive only for slow-path reading transactions

Algorithm 2 does not incur the linear instrumentation cost on the fast-path reading transactions (inherent to Algorithm 1), but provides progressiveness only for slow-path reading transactions. The instrumentation cost on fast-path t-reads is avoided by using a global lock that serializes all updating slow-path transactions during the $tryC_k$ procedure. Fast-path transactions simply check if this lock is held without acquiring it (similar to TLE [22]). While per-read instrumentation is avoided, Algorithm 2 still has per-write instrumentation.

4.3 Sacrificing progressiveness and minimizing contention window

Observe that the lower bound in Theorem 1 assumes progressiveness for both slow-path and fast-path transactions along with opacity and invisible reads. Note that Algorithm 2 retains the validation step complexity cost since it provides progressiveness for slow-path readers.

Hybrid NOrec [7] is a HyTM implementation that does not satisfy progressiveness (unlike its STM counterpart NOrec), but mitigates the step-complexity cost on slow-path transactions by performing incremental validation during a transactional read *iff* the shared memory has changed since the start of the transaction. Conceptually, Hybrid NOrec uses a global sequence lock *gsl* that is incremented at the start and end of each transaction's commit procedure. Readers can use the value of gsl to determine whether shared memory has changed between two configurations. Unfortunately, with this approach, two fast path transactions will always conflict on the gsl if their commit procedures are concurrent. To reduce the contention window for fast path transactions, the gsl is actually implemented as two separate locks (the second one called *esl*). A slow-path transaction locks both esl and gsl while it is committing. Instead of incrementing gsl, a fast path transaction checks if esl is locked and aborts if it is. Then, at the end of the fast path transaction's commit procedure, it increments gsl twice (quickly locking and releasing it and immediately commits in hardware). Although the window for fast path transactions to contend on gsl is small, our experiments have shown that contention on gsl has a significant impact on performance.

Algorithm 1 Progressive fast-path and slow-path opaque HyTM implementation; code for transaction T_k

```
1  Shared objects
2      v_j, value of each t-object X_j
3      r_j, a sequence lock of each t-object X_j
4
5  Code for fast-path transactions
6
7  read_k(X_j)
8      ov_j := v_j
9      or_j := r_j                          ▷ direct read
10     if or_j.isLocked() then return A_k
11     return ov_j
12
13 write_k(X_j, v)
14     or_j := r_j
15     if or_j.isLocked() then return A_k
16     r_j := or_j.IncSequence()
17     v_j := v
18     return OK
19
20 tryC_k()
21     commit-cache_i
22
23 Function: release(Q)
24     for each X_j ∈ Q do r_j := or_j.unlock()
25
26 Function: acquire(Q)
27     for each X_j ∈ Q
28         if r_j.tryLock()                 ▷ CAS/LLSC
29             Lset(T_k) := Lset(T_k) ∪ {X_j}
30         else
31             release(Lset(T_k))
32             return false
33     return true
34
35 Code for slow-path transactions
36
37 Read_k(X_j)
38     if X_j ∈ Wset(T_k) then return Wset(T_k).locate(X_j)
39     or_j := r_j
40     ov_j := v_j
41     Rset(T_k) := Rset(T_k) ∪ {X_j,or_j}
42     if or_j.isLocked() then return A_k
43     if not validate() then return A_k
44     return ov_j
45
46 write_k(X_j, v)
47     or_j := r_j
48     nv_j := v
49     if or_j.isLocked() then return A_k
50     Wset(T_k) := Wset(T_k) ∪ {X_j, nv_j, or_j}
51     return OK
52
53 tryC_k()
54     if Wset(T_k) = ∅ then return C_k
55     if not acquire(Wset(T_k)) then return A_k
56     if not validate()
57             release(Wset(T_k))
58             return A_k
59     for each X_j ∈ Wset(T_k) do v_j := nv_j
60     release(Wset(T_k))
61     return C_k
62
63 Function: validate()
64     if ∃ X_j ∈ Rset(T_k) : or_j.getSequence() ≠ r_j.getSequence() then return false
65     return true
```

Algorithm 2 Opaque HyTM implementation that is progressive only for slow-path reading transactions; code for T_k by process p_i

```
1  Shared objects
2      L, global lock

4  Code for fast-path transactions
5  start_k()
6      if L.isLocked() then return A_k

8  read_k(X_j)
9      ov_j := v_j
10     return ov_j

12 write_k(X_j, v)
13     or_j := r_j
14     r_j := or_j.IncSequence()
15     v_j := v
16     return OK

18 tryC_k()
19     return commit-cache_i

23 Code for slow-path transactions

25 tryC_k()
26     if Wset(T_k) = ∅ then return C_k
27     L.Lock()
28     if not acquire(Wset(T_k)) then return A_k
29     if not validate() then
30         release(Wset(T_k))
31         return A_k
32     for each X_j ∈ Wset(T_k) do v_j := nv_j
33     release(Wset(T_k))
34     return C_k

36 Function: release(Q)
37     for each X_j ∈ Q do r_j := nr_j.unlock()
38     L.unlock(); return OK
```

5 Evaluation

We now study the performance characteristics of Algorithms 1 and 2, Hybrid NOrec, TLE and TL2. Our experimental goals are:

(G1) to study the performance impact of instrumentation on the fast-path and validation on the slow-path,

(G2) to understand how HyTM algorithm design affects performance with Intel and IBM POWER8 HTMs, and

(G3) to determine whether direct accesses can be used to obtain performance improvements on IBM POWER8 using suspend/resume instructions to escape from a hardware transaction.

5.1 Experimental system

The experimental Intel system is a 2-socket Intel E7-4830 v3 with 12 cores per socket and 2 hyperthreads (HTs) per core, for a total of 48 threads. Each core has a private 32KB L1 cache and 256KB L2 cache (shared between HTs on a core). All cores on a socket share a 30MB L3 cache. This system has a non-uniform memory architecture (NUMA) in which threads have significantly different access costs to different parts of memory depending on which processor they are currently executing on. The machine has 128GB of RAM, and runs Ubuntu 14.04 LTS. All code was compiled with the GNU C++ compiler (G++) 4.8.4 with build target x86_64-linux-gnu and compilation options `-std=c++0x -O3 -mx32`.

We pin threads so that the first socket is saturated before we place any threads on the second socket. Thus, thread counts 1-24 run on a single socket. Furthermore, hyperthreading is engaged on the first socket for thread counts 13-24, and on the second socket for thread counts 37-48. Consequently, our graphs clearly show the effects of NUMA and hyperthreading.

The experimental POWER8 system is a IBM S822L with 2x 12-core 3.02GHz processor cards, 128GB of RAM, running Ubuntu 16.04 LTS. All code was compiled using G++ 5.3.1. This is a dual socket machine, and each socket has two NUMA *zones*. It is expensive to access memory on a different NUMA zone, and even more expensive if the NUMA zone is on a different socket. POWER8 uses the L2 cache for detecting tracking set aborts, and limits

the size of a transaction's read- and write-set to 8KB each [20]. This is in contrast to Intel which tracks conflicts on the entire L3 cache, and limits a transaction's read-set to the L3 cache size, and its write-set to the L1 cache size.

We pin one thread on each core within a NUMA zone before moving to the next zone. We remark that unlike the thread pinning policy for Intel which saturated the first socket before moving to the next, this proved to be the best policy for POWER8 which experiences severe negative scaling when threads are saturated on a single 8-way hardware multi-threaded core. This is because all threads on a core share resources, including the L1 and L2 cache, a single branch execution pipeline, and only two load-store pipelines.

5.2 Hybrid TM implementations

For TL2, we used the implementation published by its authors. We implemented the other algorithms in C++. Each hybrid TM algorithm first attempts to execute a transaction on the fast-path, and will continue to execute on the fast-path until the transaction has experienced 20 aborts, at which point it will fall back to the slow-path. We implemented Algorithm 1 on POWER8 where each read of a sequence lock during a transactional read operation was enclosed within a pair of suspend/resume instructions to access them without incurring tracking set aborts (Algorithm 1*). We remark that this does not affect the opacity of the implementation. We also implemented the variant of Hybrid NOrec (Hybrid NOrec*) in which the update to gsl is performed using a fetch-increment primitive between suspend/resume instructions, as is recommended in [23].

In each algorithm, instead of placing a lock next to each address in memory, we allocated a global array of one million locks, and used a simple hash function to map each address to one of these locks. This avoids the problem of having to change a program's memory layout to incorporate locks, and greatly reduces the amount of memory needed to store locks, at the cost of some possible false conflicts since many addresses map to each lock. Note that the exact same approach was taken by the authors of TL2.

We chose *not* to compile the hybrid TMs as separate libraries, since invoking library functions for each read and write can cause algorithms to incur enormous overhead. Instead, we compiled each hybrid TM directly into the code that uses it.

5.3 Experimental methodology

We used a simple unbalanced binary search tree (BST) microbenchmark as a vehicle to study the performance of our implementations. The BST implements a dictionary, which contains a set of keys, each with an associated value. For each TM algorithm and update rate $U \in \{40, 10, 0\}$, we run six timed *trials* for several thread counts n. Each trial proceeds in two phases: *prefilling* and *measuring*. In the prefilling phase, n concurrent threads perform 50% *Insert* and 50% *Delete* operations on keys drawn uniformly randomly from $[0, 10^5)$ until the size of the tree converges to a steady state (containing approximately $10^5/2$ keys). Next, the trial enters the measuring phase, during which threads begin counting how many operations they perform. In this phase, each thread performs $(U/2)$% *Insert*, $(U/2)$% *Delete* and $(100 - U)$% *Search* operations, on keys/values drawn uniformly from $[0, 10^5)$, for one second.

Uniformly random updates to an unbalanced BST have been proven to yield trees of logarithmic height with high probability. Thus, in this type of workload, almost all transactions succeed in hardware, and the slow-path is almost never used. To study performance when transactions regularly run on slow-path, we introduced an operation called a *RangeIncrement*

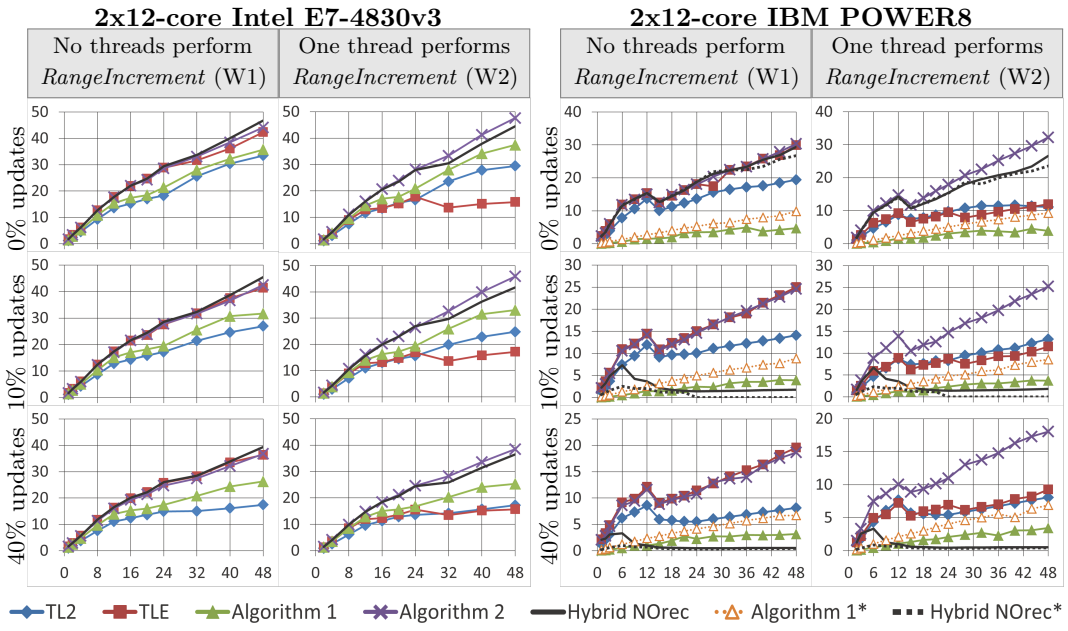


Figure 2 Results for a **BST microbenchmark**. The x-axis represents the number of concurrent threads. The y-axis represents operations per microsecond.

that often fails in hardware and must run on the slow-path. A *RangeIncrement*(low, hi) atomically increments the values associated with each key in the range $[low, hi]$ present in the tree. Note that a *RangeIncrement* is more likely to experience data conflicts and capacity aborts than BST updates, which only modify a single node.

We consider two types of workloads: (W1) all n threads perform *Insert*, *Delete* and *Search*, and (W2) $n-1$ threads perform *Insert*, *Delete* and *Search* and one thread performs only *RangeIncrement* operations. Figure 2 shows the results for both types of workloads.

5.4 Results

We first discuss the results for the Intel machine. We first discuss the 0% updates graph for workload type W1. In this graph, essentially all operations committed in hardware. In fact, in each trial, a small fraction of 1% of operations ran on the slow-path. Thus, any performance differences shown in the graph are essentially differences in the performance of the algorithms' respective fast-paths (with the exception of TL2). Algorithm 1, which has instrumentation in its fast-path read operations, has significantly lower performance than Algorithm 2, which does not. Since this is a read-only workload, this instrumentation is responsible for the performance difference.

In the W1 workloads, TLE, Algorithm 2 and Hybrid NOrec perform similarly (with a small performance advantage for Hybrid NOrec at high thread counts). This is because the fast-paths for these three algorithms have similar amounts of instrumentation: there is no instrumentation for reads or writes, and the transaction itself incurs one or two metadata accesses. In contrast, in the W2 workloads, TLE performs quite poorly, compared to the HyTM algorithms. In these workloads, transactions must periodically run on the slow-path, and in TLE, this entails acquiring a global lock that restricts progress for all other threads. At high thread counts this significantly impacts performance. Its performance decreases as the sizes of the ranges passed to *RangeIncrement* increase. Its performance is also negatively

impacted by NUMA effects at thread counts higher than 24. (This is because, when a thread p reads the lock and incurs a cache miss, if the lock was last held by another thread on the same socket, then p can fill the cache miss by loading it from the shared L3 cache. However, if the lock was last held by a thread on a different socket, then p must read the lock state from main memory, which is significantly more expensive.)

On the other hand, in each graph in the W2 workloads, the performance of each HyTM (and TL2) is similar to its performance in the corresponding W1 workload graph. For Algorithm 1 (and TL2), this is because of progressiveness. Although Algorithm 2 is not truly progressive, fast-path transactions will abort only if they are concurrent with the commit procedure of a slow-path transaction. In *RangeIncrement* operations, there is a long read-only prefix (which is exceptionally long because of Algorithm 2's quadratic validation) followed by a relatively small set of writes. Thus, *RangeIncrement* operations have relatively little impact on the fast-path. The explanation is similar for Hybrid NOrec (except that it performs less validation than Algorithm 2).

Observe that the performance of Hybrid NOrec decreases slightly, relative to Algorithm 2, after 24 threads. Recall that, in Hybrid NOrec, the global sequence number is a single point of contention on the fast-path. (In Algorithm 2, the global lock is only modified by slow-path transactions, so fast-path transactions do not have a single point of contention.) We believe this is due to NUMA effects, similar to those described in [5]. Specifically, whenever a threads on the first socket performs a fast-path transaction that commits and modifies the global lock, it causes cache invalidations for all other threads. Threads on socket two must then load the lock state from main memory, which takes much longer than loading it from the shared L3 cache. This lengthens the transaction's window of contention, making it more likely to abort. (In the 0% updates graph in the W2 workload, we still see this effect, because there is a thread performing *RangeIncrement* operations.)

We now discuss the results for the IBM POWER8 machine. Algorithm 1 performs poorly on POWER8: POWER8 transactions can only load 64 cache lines before they will abort [21]. Transactions read locks and tree nodes, which are in different cache lines: together, they often exceed 64 cache lines loaded in a tree operation, so most transactions cannot succeed in hardware. Consequently, on POWER8, it is incredibly important either to have minimal instrumentation in transactions, or for metadata to be located in the same cache lines as program data. Of course, the latter is not possible for HyTMs, which do not have control over the layout of program data. Consequently, Algorithm 2 outperforms Algorithm 1 in POWER8 quite easily by avoiding the per-read instrumentation.

Algorithm 1 is improved slightly by the expensive (on POWER8) suspend/resume on sequence locks during transactional reads, but it still performs relatively poorly. To make suspend/resume a practical tool, one could imagine attempting to collect several metadata accesses and perform them together to amortize the cost of a suspend/resume pair. For instance, in Algorithm 1, one might try to update the locks for all of the transactional writes at once, when the transaction commits. Typically one would accomplish this by logging all writes so that a process can remember which addresses it must lock at commit time. However, logging the writes inside the transaction would be at least as costly as just performing them.

Observe that Hybrid NOrec does far worse with updates in POWER8 than on the Intel machine. This is due to the fact that fetch-increment on a single location experiences severe negative scaling on the POWER8 processor: e.g., in one second, a single thread can perform 37 fetch-add operations while 6 threads perform a total of 9 million and 24 threads perform only 4 million fetch-add operations. In contrast, the Intel machine performs 32 million operations with 6 threads and 45 million with 24 threads. This is likely because this Intel processor provides fetch-add instructions while it must be emulated on POWER8.

In Hybrid NOrec*, the non-speculative increment of gsl actually makes performance worse. Recall that in Hybrid NOrec, if a fast-path transaction T_1 increments gsl, and then a software transaction T_2 reads gsl (as part of validation) before T_1 commits, then T_1 will abort, and T_2 will not see T_1's change to gsl. So, T_2 will have a higher chance of avoiding incremental validation (and, hence, will likely take less time to run, and have a smaller contention window). However, in Hybrid NOrec*, once T_1 increments gsl, T_2 will see the change to gsl, regardless of whether T_1 commits or aborts. Thus, T_2 will be forced to perform incremental validation. In our experiments, we observed that a much larger number of transactions ran on the fallback path in Hybrid NOrec* than in Hybrid NOrec (often several orders of magnitude more).

6 Related work and discussion

HyTM implementations and complexity. Early HyTMs like the ones described in [9, 15] provided progressiveness, but subsequent HyTM proposals like PhTM [18] and Hybrid-NOrec [7] sacrificed progressiveness for lesser instrumentation overheads. However, the clear trade-off in terms of concurrency vs. instrumentation for these HyTMs have not been studied in the context of currently available HTM architectures. This instrumentation cost on the fast-path was precisely characterized in [3]. In this paper, we proved the inherent cost of concurrency on the slow-path thus establishing a surprising, but intuitive complexity separation between progressive STMs and HyTMs. Moreover, to the best of our knowledge, this is the first work to consider the theoretical foundations of the cost of concurrency in HyTMs in theory and practice (on currently available HTM architectures). Proof of Theorem 1 is based on the analogous proof for step complexity of STMs that are *disjoint-access parallel* [16, 12]. Our implementation of Hybrid NOrec follows [23], which additionally proposed the use of direct accesses in fast-path transactions to reduce instrumentation overhead in the AMD Advanced Synchronization Facility (ASF) architecture.

Beyond the two path HyTM approach. *Employing an uninstrumented fast fast-path.* We now describe how every transaction may first be executed in a "fast" fast-path with almost no instrumentation and if unsuccessful, may be re-attempted in the fast-path and subsequently in slow-path. Specifically, we transform an opaque HyTM $\mathcal{M}$ to an opaque HyTM $\mathcal{M}'$ using a shared *fetch-and-add* metadata base object F that slow-path updating transactions increment (and resp. decrement) at the start (and resp. end). In $\mathcal{M}'$, a "fast" fast-path transaction checks first if F is 0 and if not, aborts the transaction; otherwise the transaction is continued as an uninstrumented hardware transaction. The code for the fast-path and the slow-path is identical to $\mathcal{M}$.

Recent work has investigated fallback to *reduced* hardware transactions [19] in which an all-software slow-path is augmented using a slightly faster slow-path that is optimistically used to avoid running some transactions on the true software-only slow-path. Amalgamated lock elision (ALE) was proposed in [2] which improves over TLE by executing the slow-path as a series of segments, each of which is a dynamic length hardware transaction. Invyswell [6] is a HyTM design with multiple hardware and software modes of execution that gives flexibility to avoid instrumentation overhead in uncontended executions. We remark that such multi-path approaches may be easily applied to each of the Algorithms proposed in this paper. However, in the search for an efficient HyTM, it is important to strike the fine balance between concurrency, hardware instrumentation and software validation cost. Our lower bound, experimental methodology and evaluation of HyTMs provides the first clear characterization of these trade-offs in both Intel and POWER8 architectures.

Acknowledgements. Computations were performed on the SOSCIP Consortium's Agile computing platform. SOSCIP is funded by the Federal Economic Development Agency of Southern Ontario, the Province of Ontario, IBM Canada Ltd., Ontario Centres of Excellence, Mitacs and 15 Ontario academic member institutions. This work was performed while Trevor Brown was a student at the University of Toronto.

References

1 Advanced Synchronization Facility Proposed Architectural Specification, March 2009. `http://developer.amd.com/wordpress/media/2013/09/45432-ASF_Spec_2.1.pdf`.

2 Yehuda Afek, Alexander Matveev, Oscar R. Moll, and Nir Shavit. Amalgamated lock-elision. In *Proceedings of 29th Int. Sym. on Distributed Computing, DISC'15*, pages 309–324, 2015. `doi:10.1007/978-3-662-48653-5_21`.

3 Dan Alistarh, Justin Kopinsky, Petr Kuznetsov, Srivatsan Ravi, and Nir Shavit. Inherent limitations of hybrid transactional memory. In *Proceedings of 29th Int. Sym. on Distributed Computing, DISC'15*, pages 185–199, 2015.

4 Hagit Attiya, Eshcar Hillel, and Alessia Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. *Theory of Computing Systems*, 49(4):698–719, 2011.

5 Trevor Brown, Alex Kogan, Yossi Lev, and Victor Luchangco. Investigating the performance of hardware transactions on a multi-socket machine. In *Proceedings of 28th ACM Sym. on Parallelism in Algorithms and Architectures, SPAA'16*, pages 121–132, 2016.

6 Irina Calciu, Justin Gottschlich, Tatiana Shpeisman, Gilles Pokam, and Maurice Herlihy. Invyswell: a hybrid transactional memory for haswell's restricted transactional memory. In *Int. Conf. on Par. Arch. and Compilation, PACT'14*, pages 187–200, 2014.

7 Luke Dalessandro, Francois Carouge, Sean White, Yossi Lev, Mark Moir, Michael L. Scott, and Michael F. Spear. Hybrid NOrec: a case study in the effectiveness of best effort hardware transactional memory. In *ASPLOS'11*, pages 39–52. ACM, 2011.

8 Luke Dalessandro, Michael F. Spear, and Michael L. Scott. Norec: Streamlining stm by abolishing ownership records. *SIGPLAN Not.*, 45(5):67–78, January 2010.

9 Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. *SIGPLAN Not.*, 41(11):336–346, October 2006.

10 Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, pages 194–208, Berlin, Heidelberg, 2006. Springer-Verlag.

11 K. Fraser. Practical lock-freedom. Technical report, Cambridge University Computer Laboratory, 2003.

12 Rachid Guerraoui and Michal Kapalka. *Principles of Transactional Memory, Synthesis Lectures on Distributed Computing Theory*. Morgan and Claypool, 2010.

13 Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proc. of 22nd Int. Sym. on Principles of Distr. Comp.*, PODC'03, pages 92–101, New York, NY, USA, 2003. ACM.

14 Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.

15 Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP'06, pages 209–220, New York, NY, USA, 2006. ACM.

16 Petr Kuznetsov and Srivatsan Ravi. Progressive transactional memory in time and space. In *Proceedings of 13th Int. Conf. on Parallel Computing Technologies, PaCT'15*, pages 410–425, 2015.

17 Hung Q. Le, G. L. Guthrie, Derek Williams, Maged M. Michael, Brad Frey, William J. Starke, Cathy May, Rei Odaira, and Takuya Nakaike. Transactional memory support in the IBM POWER8 processor. *IBM Journal of Research and Development*, 59(1), 2015.

18 Yossi Lev, Mark Moir, and Dan Nussbaum. Phtm: Phased transactional memory. In *In Workshop on Transactional Computing (Transact)*, 2007.

19 Alexander Matveev and Nir Shavit. Reduced hardware transactions: a new approach to hybrid transactional memory. In *Proceedings of the 25th ACM symposium on Parallelism in algorithms and architectures*, pages 11–22. ACM, 2013.

20 Takuya Nakaike, Rei Odaira, Matthew Gaudet, Maged M. Michael, and Hisanobu Tomari. Quantitative comparison of hardware transactional memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8. In *Proc. of 42nd Int. Sym. on Comp. Arch.*, ISCA '15, pages 144–157, NY, USA, 2015.

21 Andrew T. Nguyen. Investigation of hardware transactional memory. 2015. http://groups.csail.mit.edu/mag/Andrew-Nguyen-Thesis.pdf.

22 Ravi Rajwar and James R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proc. of 34th ACM/IEEE Int. Sym. on Microarchitecture*, MICRO'01, pages 294–305, Washington, DC, USA, 2001.

23 Torvald Riegel, Patrick Marlier, Martin Nowack, Pascal Felber, and Christof Fetzer. Optimizing hybrid transactional memory: The importance of nonspeculative operations. In *Proc. of 23rd ACM Sym. on Parallelism in Algs. and Arch.*, pages 53–64. ACM, 2011.

24 Nir Shavit and Dan Touitou. Software transactional memory. In *Principles of Distributed Computing (PODC)*, pages 204–213, 1995.

25 Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. Performance evaluation of intel® transactional synchronization extensions for high-performance computing. In *Proceedings of Int. Conf. on High Perf. Computing, Networking, Storage and Analysis*, SC'13, pages 19:1–19:11, New York, NY, USA, 2013.

Quadratic and Near-Quadratic Lower Bounds for the CONGEST Model*†

Keren Censor-Hillel[1], Seri Khoury[2], and Ami Paz[3]

1 **Department of Computer Science, Technion, Haifa, Israel**
ckeren@cs.technion.ac.il
2 **Department of Computer Science, Technion, Haifa, Israel**
serikhoury@cs.technion.ac.il
3 **Department of Computer Science, Technion, Haifa, Israel**
amipaz@cs.technion.ac.il

Abstract

We present the first super-linear lower bounds for natural graph problems in the CONGEST model, answering a long-standing open question.

Specifically, we show that any exact computation of a minimum vertex cover or a maximum independent set requires a near-quadratic number of rounds in the CONGEST model, as well as any algorithm for computing the chromatic number of the graph. We further show that such strong lower bounds are not limited to NP-hard problems, by showing two simple graph problems in P which require a quadratic and near-quadratic number of rounds.

Finally, we address the problem of computing an exact solution to weighted all-pairs-shortest-paths (APSP), which arguably may be considered as a candidate for having a super-linear lower bound. We show a simple linear lower bound for this problem, which implies a separation between the weighted and unweighted cases, since the latter is known to have a sub-linear complexity. We also formally prove that the standard Alice-Bob framework is incapable of providing a super-linear lower bound for exact weighted APSP, whose complexity remains an intriguing open question.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, G.2.2 Graph Theory

Keywords and phrases CONGEST, Lower Bounds, Minimum Vertex Cover, Chromatic Number, Weighted APSP

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.10

1 Introduction

It is well-known and easily proven that many graph problems are *global* for distributed computing, in the sense that solving them necessitates communication throughout the network. This implies tight $\Theta(D)$ complexities, where D is the diameter of the network, for global problems in the LOCAL model. In this model, a message of unbounded size can be sent over each edge in each round, which allows to learn the entire topology in D rounds. Global problems are widely studied in the CONGEST model, in which the size of each message is restricted to $O(\log n)$ bits, where n is the size of the network. The trivial complexity of learning the entire topology of an m-edges graph in the CONGEST model is

* Supported by ISF Individual Research Grant 1696/14.
† A full version of the paper is available at https://arxiv.org/abs/1705.05646.


© Keren Censor-Hillel, Seri Khoury, and Ami Paz;
licensed under Creative Commons License CC-BY

$O(m)$, and since m can be as large as $\Theta(n^2)$, one of the most basic questions for a global problem is how fast in terms of n it can be solved in the CONGEST model.

Some global problems admit fast $O(D)$-round solutions in the CONGEST model, such as constructing a breadth-first search tree [60]. Some others have complexities of $\tilde{\Theta}(D+\sqrt{n})$, such as constructing a minimum spanning tree, and various approximation and verification problems [33, 40, 46, 61, 62, 65]. Some problems are yet harder, with complexities that are near-linear in n [1, 33, 42, 52, 61]. For some problems, no $O(n)$ solutions are known and they are candidates to being even harder that the ones with linear-in-n complexities.

A major open question about global graph problems in the CONGEST model is whether natural graph problems for which a super-linear number of rounds is required indeed exist. In this paper, we answer this question in the affirmative. That is, our conceptual contribution is that **there exist super-linearly hard problems in the CONGEST model**. In fact, the lower bounds that we prove in this paper are as high as quadratic in n, or quadratic up to logarithmic factors, and hold even for networks of a constant diameter. Our lower bounds also imply linear and near-linear lower bounds for the CLIQUE-BROADCAST model.

We note that high lower bounds for the CONGEST model may be obtained rather artificially, by forcing large inputs and outputs that must be exchanged. However, we emphasize that all the problems for which we show our lower bounds can be reduced to simple decision problems, where each node needs to output a single bit. All inputs to the nodes, if any, consist of edge weights that can be represented by polylog$n$ bits.

Technically, we prove a lower bound of $\Omega(n^2/\log^2 n)$ on the number of rounds required for computing an exact minimum vertex cover, which also extends to computing an exact maximum independent set. This is in stark contrast to the recent $O(\log\Delta/\log\log\Delta)$-round algorithm of [8] for obtaining a $(2+\epsilon)$-approximation to the minimum vertex cover. Similarly, we give an $\Omega(n^2/\log^2 n)$ lower bound for 3-coloring a 3-colorable graph, which extends also for deciding whether a graph is 3-colorable, and also implies the same hardness for computing the chromatic number χ or computing a χ-coloring. These lower bounds hold even for randomized algorithms which succeed with high probability.[1]

An immediate question that arises is whether only NP-hard problems are super-linearly hard in the CONGEST model. We provide a negative answer to such a postulate, by showing two simple problems that admit polynomial-time sequential algorithms, but in the CONGEST model require $\Omega(n^2)$ rounds (identical subgraph detection) or $\Omega(n^2/\log n)$ rounds (weighted cycle detection). The latter also holds for randomized algorithms, while for the former we show a randomized algorithm that completes in $O(D)$ rounds, providing the strongest possible separation between deterministic and randomized complexities for global problems in the CONGEST model.

Finally, we address the intriguing open question of the complexity of computing exact weighted all-pairs-shortest-paths (APSP) in the CONGEST model. While the complexity of the unweighted version of APSP is $\Theta(n/\log n)$, as follows from [33, 43], the complexity of weighted APSP remains largely open, and only recently the first sub-quadratic algorithm was given in [29]. With the current state-of-the-art, this problem could be considered as a suspect for having a super-linear complexity in the CONGEST model. While we do not pin-down the complexity of weighted APSP in the CONGEST model, we provide a truly linear lower bound of $\Omega(n)$ rounds for it, which separates its complexity from that of the unweighted case. Moreover, we argue that it is not a coincidence that we are currently unable to show

[1] We say that an event occurs with high probability (w.h.p) if it occurs with probability at least $1-\frac{1}{n^c}$, for some constant $c>0$.

super-linear lower bound for weighted APSP, by formally proving that the commonly used framework of reducing a 2-party communication problem to a problem in the CONGEST model cannot provide a super-linear lower bound for weighted APSP, regardless of the function and the graph construction used. This implies that **obtaining any super-linear lower bound for weighted APSP provably requires a new technique**.

1.1 The Challenge

Many lower bounds for the CONGEST model rely on reductions from 2-party communication problems (see, e.g., [1, 17, 26, 28, 33, 42, 57, 58, 62, 65]). In this setting, two players, Alice and Bob, are given inputs of K bits and need to a single output a bit according to some given function of their inputs. One of the most common problem for reduction is Set Disjointness, in which the players need to decide whether there is an index for which both inputs are 1. That is, if the inputs represent subsets of $\{0, \ldots, K-1\}$, the output bit of the players needs to indicate whether their input sets are disjoint. The communication complexity of 2-party Set Disjointness is known to be $\Theta(K)$ [50].

In a nutshell, there are roughly two standard frameworks for reducing the 2-party communication problem of computing a function f to a problem P in the CONGEST model. One of these frameworks works as follows. A graph construction is given, which consists of some fixed edges and some edges whose existence depends on the inputs of Alice and Bob. This graph should have the property that a solution to P over it determines the solution to f. Then, given an algorithm ALG for solving P in the CONGEST model, the vertices of the graph are split into two disjoint sets, V_A and V_B, and Alice simulates ALG over V_A while Bob simulates ALG over V_B. The only communication required between Alice and Bob in order to carry out this simulation is the content of messages sent in each direction over the edges of the cut $C = E(V_A, V_B)$. Therefore, given a graph construction with a cut of size $|C|$ and inputs of size K for a function f whose communication complexity on K bits is at least $CC(f)$, the round complexity of ALG is at least $\Omega(CC(f)/|C| \log n)$.

The challenge in obtaining super-linear lower bounds was previously that the cuts in the graph constructions were large compared with the input size K. For example, the graph construction for the lower bound for computing the diameter in [33] has $K = \Theta(n^2)$ and $|C| = \Theta(n)$, which gives an almost linear lower bound. The graph construction in [33] for the lower bound for computing a $(3/2 - \epsilon)$-approximation to the diameter has a smaller cut of $|C| = \Theta(\sqrt{n})$, but this comes at the price of supporting a smaller input size $K = \Theta(n)$, which gives a lower bound that is roughly a square-root of n.

To overcome this difficulty, we leverage the recent framework of [1], which provides a bit-gadget whose power is in allowing a logarithmic-size cut. We manage to provide a graph construction that supports inputs of size $K = \Theta(n^2)$ in order to obtain our lower bounds for minimum vertex cover, maximum independent set and 3-coloring[2]. The latter is also inspired by, and is a simplification of, a lower bound construction for the size of proof labelling schemes [34].

Further, for the problems in P that we address, the cut is as small as $|C| = O(1)$. For one of the problems, the size of the input is such that it allows us to obtain the highest possible lower bound of $\Omega(n^2)$ rounds.

[2] It can also be shown, by simple modifications to our constructions, that these problems require $\Omega(m)$ rounds, for graphs with m edges.

With respect to the complexity of the weighted APSP problem, we show an embarrassingly simple graph construction that extends a construction of [57], which leads to an $\Omega(n)$ lower bound. However, we argue that a new technique must be developed in order to obtain any super-linear lower bound for weighted APSP. Roughly speaking, this is because given a construction with a set S of nodes that touch the cut, Alice and Bob can exchange $O(|S|n \log n)$ bits which encode the weights of all lightest paths from any node in their set to a node in S. Since the cut has $\Omega(|S|)$ edges, and the bandwidth is $\Theta(\log n)$, this cannot give a lower bound of more than $\Omega(n)$ rounds. With some additional work, our proof can be carried over to a larger number of players at the price of a small logarithmic factor, as well as to the second Alice-Bob framework used in previous work (e.g. [65]), in which Alice and Bob do not simulate nodes in a fixed partition, but rather in decreasing sets that partially overlap. Thus, determining the complexity of weighted APSP requires new tools, which we leave as a major open problem.

Roadmap. Section 3 contains our lower bound for computing exact minimum vertex cover or exact maximum independent set. In Section 4 we show our lower bound for computing exact weighted-all-pairs-shortest-paths, and prove that the Alice-Bob framework cannot give a super-linear lower bound for this task. Due to space limitations, our lower bounds for 3-coloring a 3-colorable graph, identical subgraphs detection, and weighted cycle detection appear only in the full version of the paper [18].

1.2 Additional Related Work

Vertex Coloring, Minimum Vertex Cover, and Maximum Independent Set: One of the most central problems in graph theory is vertex coloring, which has been extensively studied in the context of distributed computing (see, e.g., [9, 10, 11, 12, 13, 14, 19, 21, 22, 30, 31, 32, 38, 54, 56, 63, 66] and references therein). The special case of finding a $(\Delta + 1)$-coloring, where Δ is the maximum degree of a node in the network, has been the focus of many of these studies, but is a *local* problem, which can be solved in much less than a sublinear number of rounds.

Another classical problem in graph theory is finding a minimum vertex cover (MVC). In distributed computing, the time complexity of approximating MVC has been addressed in several cornerstone studies [5, 6, 8, 14, 35, 36, 37, 45, 47, 48, 49, 59, 64].

Observe that finding a minimum size vertex cover is equivalent to finding a maximum size independent set. However, these problems are not equivalent in an approximation-preserving way. Distributed approximations for maximum independent set has been studied in [7, 15, 23, 53].

Distance Computations: Distance computation problems have been widely studied in the CONGEST model for both weighted and unweighted networks [1, 33, 39, 40, 41, 42, 43, 51, 52, 57, 61]. One of the most fundamental problems of distance computations is computing all pairs shortest paths. For unweighted networks, an upper bound of $O(n/\log n)$ was recently shown by [43], matching the lower bound of [33]. Moreover, the possibility of bypassing this near-linear barrier for any constant approximation factor was ruled out by [57]. For the weighted case, however, we are still very far from understanding the complexity of APSP, as there is still a huge gap between the upper and lower bounds. Recently, Elkin [29] showed an $O(n^{\frac{5}{3}} \cdot \log^{\frac{2}{3}}(n))$ upper bound for weighted APSP, while the previously highest lower bound was the near-linear lower bound of [57] (which holds also for any $(\text{poly}\, n)$-approximation factor in the weighted case).

Distance computation problems have also been considered in the CONGESTED-CLIQUE model [16, 39, 41], in which the underlying communication network forms a clique. In this model [16] showed that unweighted APSP, and a $(1+o(1))$-approximation for weighted APSP, can be computed in $O(n^{0.158})$ rounds.

Subgraph Detection: The problem of finding subgraphs of a certain topology has received a lot of attention in both the sequential and the distributed settings (see, e.g., [2, 3, 4, 16, 24, 25, 26, 44, 55, 67] and references therein).

The problems of finding paths of length 4 or 5 with zero weight are also related to other fundamental problems, notable in our context is APSP [2].

2 Preliminaries

2.1 Communication Complexity

In a two-party communication complexity problem [50], there is a function $f : \{0,1\}^K \times \{0,1\}^K \to \{\text{TRUE}, \text{FALSE}\}$, and two players, Alice and Bob, who are given two input strings, $x, y \in \{0,1\}^K$, respectively, that need to compute $f(x,y)$. The *communication complexity* of a protocol π for computing f, denoted $CC(\pi)$, is the maximal number of bits Alice and Bob exchange in π, taken over all values of the pair (x,y). The *deterministic communication complexity* of f, denoted $CC(f)$, is the minimum over $CC(\pi)$, taken over all deterministic protocols π that compute f.

In a *randomized protocol* π, Alice and Bob may each use a random bit string. A randomized protocol π computes f if the probability, over all possible bit strings, that π outputs $f(x,y)$ is at least $2/3$. The *randomized communication complexity* of f, $CC^R(f)$, is the minimum over $CC(\pi)$, taken over all randomized protocols π that compute f.

In the *Set Disjointness* problem (DISJ_K), the function f is $\text{DISJ}_K(x,y)$, whose output is FALSE if there is an index $i \in \{0,\ldots,K-1\}$ such that $x_i = y_i = 1$, and TRUE otherwise. In the *Equality* problem (EQ_K), the function f is $\text{EQ}_K(x,y)$, whose output is TRUE if $x = y$, and FALSE otherwise.

Both the deterministic and randomized communication complexities of the DISJ_K problem are known to be $\Omega(K)$ [50, Example 3.22]. The deterministic communication complexity of EQ_K is in $\Omega(K)$ [50, Example 1.21], while its randomized communication complexity is in $\Theta(\log K)$ [50, Example 3.9].

2.2 Lower Bound Graphs

To prove lower bounds on the number of rounds necessary in order to solve a distributed problem in the CONGEST model, we use reductions from two-party communication complexity problems. To formalize them we use the following definition. Let $\mathcal{G}$ be the set of all graphs.

▶ **Definition 1** (Family of Lower Bound Graphs). Fix an integer K, a function $f : \{0,1\}^K \times \{0,1\}^K \to \{\text{TRUE}, \text{FALSE}\}$ and a predicate $P : \mathcal{G} \to \{\text{TRUE}, \text{FALSE}\}$. The family of graphs $\{G_{x,y} = (V, E_{x,y}) \mid x, y \in \{0,1\}^K\}$, is said to be a family of *lower bound graphs w.r.t. f and P* if the following properties hold:

(1) The set of nodes V is the same for all graphs, and we denote by $V = V_A \dot\cup V_B$ a fixed partition of it;
(2) The existence or the weight of edges in $V_A \times V_A$ may depend on x;
(3) The existence or the weight of edges in $V_B \times V_B$ may depend on y;
(4) $P(G_{x,y}) = f(x,y)$.

We use the following theorem, which is standard in the context of communication complexity-based lower bounds for the CONGEST model (see, e.g. [1, 26, 33, 41]) Its proof is by a standard simulation argument.

▶ **Theorem 2.** *Fix a function $f : \{0,1\}^K \times \{0,1\}^K \to \{\text{TRUE}, \text{FALSE}\}$ and a predicate P. If there is a family $\{G_{x,y}\}$ of lower bound graphs with $C = E(V_A, V_B)$ then any deterministic algorithm for deciding P in the CONGEST model requires $\Omega(CC(f)/|C|\log n)$ rounds, and any randomized algorithm for deciding P in the CONGEST model requires $\Omega(CC^R(f)/|C|\log n)$ rounds.*

Proof. Let ALG be a distributed algorithm in the CONGEST model that decides P in T rounds. Given inputs $x, y \in \{0,1\}^K$ to Alice and Bob, respectively, Alice constructs the part of $G_{x,y}$ for the nodes in V_A and Bob does so for the nodes in V_B. This can be done by items (1),(2) and (3) in Definition 1, and since $\{G_{x,y}\}$ satisfies this definition. Alice and Bob simulate ALG by exchanging the messages that are sent during the algorithm between nodes of V_A and nodes of V_B in either direction. (The messages within each set of nodes are simulated locally by the corresponding player without any communication). Since item (4) in Definition 1 also holds, we have that Alice and Bob correctly output $f(x, y)$ based on the output of ALG. For each edge in the cut, Alice and Bob exchange $O(\log n)$ bits per round. Since there are T rounds and $|C|$ edges in the cut, the number of bits exchanged in this protocol for computing f is $O(T|C|\log n)$. The lower bounds for T now follows directly from the lower bounds for $CC(f)$ and $CC^R(f)$. ◀

In what follows, for each decision problem addressed, we describe a fixed graph construction $G = (V, E)$, which we then generalize to a family of graphs $\{G_{x,y} = (V, E_{x,y}) \mid x, y \in \{0,1\}^K\}$, which we show to be a family lower bound graphs w.r.t. to some function f and the required predicate P. By Theorem 2 and the known lower bounds for the 2-party communication problem, we deduce a lower bound for any algorithm for deciding P in the CONGEST model.

▶ Remark. For our constructions that use the Set Disjointness function as f, we need to exclude the possibilities of all-1 input vectors, as otherwise the communication graph is not connected. However, this restriction does not change the asymptotic bounds for Set Disjointness, since computing this function while excluding all-1 input vectors can be reduced to computing this function for inputs that are shorter by one bit (by having the last bit be fixed to 0).

3 Minimum Vertex Cover and Maximum Independent Set

The first near-quadratic lower bound we present is for computing a minimum vertex cover, as stated in the following theorem.

▶ **Theorem 3.** *Any distributed algorithm in the CONGEST model for computing a minimum vertex cover or for deciding whether there is a vertex cover of a given size M requires $\Omega(n^2/\log^2 n)$ rounds.*

Finding the minimum size of a vertex cover is equivalent to finding the maximum size of a maximum independent set, because a set of nodes is a vertex cover if and only if its complement is an independent set. Thus, Theorem 4 is a direct corollary of Theorem 3.

▶ **Theorem 4.** *Any distributed algorithm in the CONGEST model for computing a maximum independent set or for deciding whether there is an independent set of a given size requires $\Omega(n^2/\log^2 n)$ rounds.*

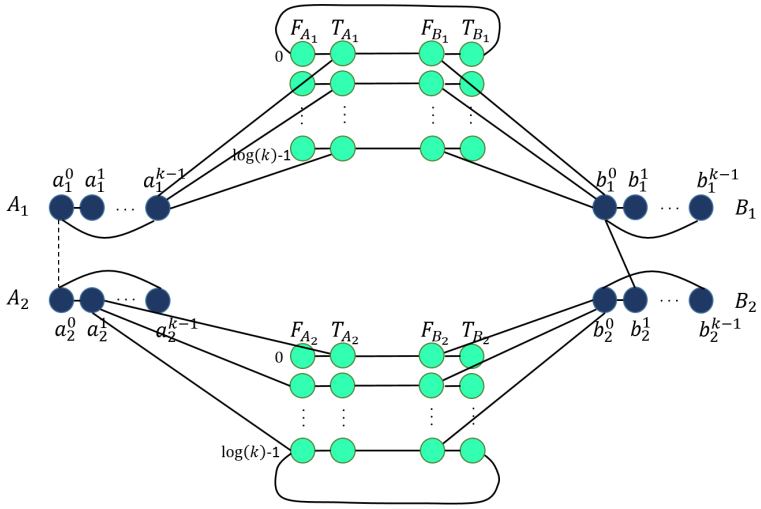


Figure 1 The family of lower bound graphs for deciding the size of a vertex cover, with many edges omitted for clarity. The node a_1^{k-1} is connected to all the nodes in T_{A_1}, and a_2^1 is connected to $t_{A_2}^0$ and to all the nodes in $F_{A_2} \setminus \{f_{A_2}^0\}$. Examples of edges from b_1^0 and b_2^0 to the bit-gadgets are also given. An additional edge, which is among the edges corresponding to the strings x and y, is $\{b_1^0, b_2^1\}$, while the edge $\{a_1^0, a_2^0\}$ does not exist. Here, $x_{0,0} = 1$ and $y_{0,1} = 0$.

Observe that a lower bound of L for deciding whether there is a vertex cover of some given size M or not implies a lower bound of $L - O(D)$ for computing a minimum vertex cover. This is because computing the size of a given subset of nodes can be easily done in $O(D)$ rounds using standard tools. Therefore, to prove Theorem 3 it is sufficient to prove its second part. We do so by describing a family of lower bound graphs with respect to the Set Disjointness function and the predicate P that says that the graph has a vertex cover of size M. We begin with describing the fixed graph construction $G = (V, E)$ and then define the family of lower bound graphs and analyze its relevant properties.

The fixed graph construction: Let k be a power of 2. The fixed graph (Figure 1) consists of four cliques of size k: $A_1 = \{a_1^i \mid 0 \leq i \leq k-1\}$, $A_2 = \{a_2^i \mid 0 \leq i \leq k-1\}$, $B_1 = \{b_1^i \mid 0 \leq i \leq k-1\}$ and $B_2 = \{b_2^i \mid 0 \leq i \leq k-1\}$. In addition, for each set $S \in \{A_1, A_2, B_1, B_2\}$, there are two corresponding sets of nodes of size $\log k$, denoted $F_S = \{f_S^h \mid 0 \leq h \leq \log k - 1\}$ and $T_S = \{t_S^h \mid 0 \leq h \leq \log k - 1\}$.

The latter are called *bit-gadgets* and their nodes are *bit-nodes*.

The bit-nodes are partitioned into $2 \log k$ 4-cycles: for each $h \in \{0, \ldots, \log k - 1\}$ and $\ell \in \{1, 2\}$, we connect the 4-cycle $(f_{A_\ell}^h, t_{A_\ell}^h, f_{B_\ell}^h, t_{B_\ell}^h)$. Note that there are no edges between pairs of nodes denoted f_S^h, or between pairs of nodes denoted t_S^h.

The nodes of each set $S \in \{A_1, A_2, B_1, B_2\}$ are connected to nodes in the corresponding set of bit-nodes, according to their binary representation, as follows. Let s_ℓ^i be a node in a set $S \in \{A_1, A_2, B_1, B_2\}$, i.e. $s \in \{a, b\}$, $\ell \in \{1, 2\}$ and $i \in \{0, \ldots, k-1\}$, and let i_h denote the bit number h in the binary representation of i. For such a node s_ℓ^i define $\text{bin}(s_\ell^i) = \{f_S^h \mid i_h = 0\} \cup \{t_S^h \mid i_h = 1\}$, and connect s_ℓ^i by an edge to each of the nodes in $\text{bin}(s_\ell^i)$. The next two claims address the basic properties of vertex covers of G.

▶ **Claim 5.** *Any vertex cover of G must contain at least $k-1$ nodes from each of the clique A_1, A_2, B_1 and B_2, and at least $4 \log k$ bit-nodes.*

Proof. In order to cover all the edges of each if the cliques on A_1, A_2, B_1 and B_2, any vertex cover must contain at least $k-1$ nodes of the clique. For each $h \in \{0, \ldots, \log k - 1\}$ and $\ell \in \{1, 2\}$, in order to cover the edges of the 4-cycle $(f^h_{A_\ell}, t^h_{A_\ell}, f^h_{B_\ell}, t^h_{B_\ell})$, any vertex cover must contain at least two of the cycle nodes. ◀

▶ **Claim 6.** *If $U \subseteq V$ is a vertex cover of G of size $4(k-1) + 4\log k$, then there are two indices $i, j \in \{0, \ldots, k-1\}$ such that $a^i_1, a^j_2, b^i_1, b^j_2$ are not in U.*

Proof. By Claim 5, U must contain $k-1$ nodes from each clique A_1, A_2, B_1 and B_2, and $4\log k$ bit-nodes, so it must not contain one node from each clique. Let $a^i_1, a^j_2, b^{i'}_1, b^{j'}_2$ be the nodes in A_1, A_2, B_1, B_2 which are not in U, respectively. To cover the edges connecting a^i_1 to $\text{bin}(a^i_1)$, U must contain all the nodes of $\text{bin}(a^i_1)$, and similarly, U must contain all the nodes of $\text{bin}(b^{i'}_1)$.

If $i \neq i'$ then there is an index $h \in \{0, \ldots, \log k - 1\}$ such that $i_h \neq i'_h$. So whether both nodes of the edge $\left(f^h_{A_1}, t^h_{B_1}\right)$ are in U, or both nodes of $\left(t^h_{A_1}, f^h_{B_1}\right)$ are. However, U contains exactly $4\log k$ bit-nodes and at least two nodes from each 4-cycle, and a simple counting argument implies that U also contain at most two nodes from each 4-cycle. So, the other nodes of the 4-cycle $\{f^h_{A_1}, t^h_{B_1}, t^h_{A_1}, f^h_{B_1}\}$ are not in U, and the other edge is not covered. Thus, it must be the case that $i = i'$. A similar argument shows $j = j'$. ◀

Adding edges corresponding to the strings x and y: Given two binary strings $x, y \in \{0,1\}^{k^2}$, we augment the graph G defined above with additional edges, which defines $G_{x,y}$. Assume that x and y are indexed by pairs of the form $(i,j) \in \{0, \ldots, k-1\}^2$. For each such pair (i,j) we add to $G_{x,y}$ the following edges. If $x_{i,j} = 0$, then we add an edge between the nodes a^i_1 and a^j_2, and if $y_{i,j} = 0$ then we add an edge between the nodes b^i_1 and b^j_2. To prove that $\{G_{xy}\}$ is a family of lower bound graphs, it remains to prove the next lemma.

▶ **Lemma 7.** *The graph $G_{x,y}$ has a vertex cover of cardinality $M = 4(k-1) + 4\log k$ iff* $\text{DISJ}(x,y) = \text{FALSE}$.

Proof. For the first implication, assume that $\text{DISJ}(x,y) = \text{FALSE}$ and let $i, j \in \{0, \ldots, k-1\}$ be such that $x_{i,j} = y_{i,j} = 1$. Note that in this case a^i_1 is not connected to a^j_2, and b^i_1 is not connected to b^j_2. We define a set $U \subseteq V$ as the union of the two sets of nodes $(A_1 \setminus \{a^i_1\}) \cup (A_2 \setminus \{a^j_2\}) \cup (B_1 \setminus \{b^i_1\}) \cup (B_2 \setminus \{b^j_2\})$ and $\text{bin}(a^i_1) \cup \text{bin}(a^j_2) \cup \text{bin}(b^i_1) \cup \text{bin}(b^j_2)$, and show that U is a vertex cover of $G_{x,y}$.

First, U covers all the edges inside the cliques A_1, A_2, B_1 and B_2, as it contains $k-1$ nodes from each clique. These nodes also cover all the edges connecting nodes in A_1 to nodes in A_2 and all the edges connecting nodes in B_1 to nodes in B_2. Furthermore, U covers any edge connecting some node $u \in (A_1 \setminus \{a^i_1\}) \cup (A_2 \setminus \{a^j_2\}) \cup (B_1 \setminus \{b^i_1\}) \cup (B_2 \setminus \{b^j_2\})$ with the bit-gadgets. For each node $s \in a^i_1, a^j_2, b^i_1, b^j_2$, the nodes $\text{bin}(s)$ are in U, so U also cover the edges connecting s to the bit gadget. Finally, U covers all the edges inside the bit gadgets, as from each 4-cycle $(f^h_{A_\ell}, t^h_{A_\ell}, f^h_{B_\ell}, t^h_{B_\ell})$ it contains two non-adjacent nodes: if $i_h = 0$ then $f^h_{A_1}, f^h_{B_1} \in U$ and otherwise $t^h_{A_1}, t^h_{B_1} \in U$, and if $j_h = 0$ then $f^h_{A_2}, f^h_{B_2} \in U$ and otherwise $t^h_{A_2}, t^h_{B_2} \in U$. We thus have that U is a vertex cover of size $4(k-1) + 4\log k$, as needed.

For the other implication, let $C \subseteq V$ be a vertex cover of $G_{x,y}$ of size $4(k-1) + 4\log k$. As the set of edges of G is contained in the set of edges of $G_{x,y}$, C is also a cover of G, and by Claim 6 there are indices $i, j \in \{0, \ldots, k-1\}$ such that $a^i_1, a^j_2, b^i_1, b^j_2$ are not in C. Since C is a cover, the graph does not contain the edges (a^i_1, a^j_2) and (b^i_1, b^j_2), so we conclude that $x_{i,j} = y_{i,j} = 1$, which implies that $\text{DISJ}(x,y) = \text{FALSE}$. ◀

Having constructed the family of lower bound graphs, we are now ready to prove Theorem 3.

Proof of Theorem 3. To complete the proof of Theorem 3, we divide the nodes of G (which are also the nodes of $G_{x,y}$) into two sets. Let $V_A = A_1 \cup A_2 \cup F_{A_1} \cup T_{A_1} \cup F_{A_2} \cup T_{A_2}$ and $V_B = V \setminus V_A$. Note that $n \in \Theta(k)$, and thus $K = |x| = |y| = \Theta(n^2)$. Furthermore, note that the only edges in the cut $E(V_A, V_B)$ are the edges between nodes in $\{F_{A_1} \cup T_{A_1} \cup F_{A_2} \cup T_{A_2}\}$ and nodes in $\{F_{B_1} \cup T_{B_1} \cup F_{B_2} \cup T_{B_2}\}$, which are in total $\Theta(\log n)$ edges. Since Lemma 7 shows that $\{G_{x,y}\}$ is a family of lower bound graphs, we can apply Theorem 2 on the above partition to deduce that because of the lower bound for Set Disjointness, any algorithm in the CONGEST model for deciding whether a given graph has a cover of cardinality $M = 4(k-1) + 4\log k$ requires at least $\Omega(K/\log^2(n)) = \Omega(n^2/\log^2(n))$ rounds. ◀

4 Weighted APSP

In this section we use the following natural extension of Definition 1, in order to address more general 2-party functions, as well as distributed problems that are not decision problems.

For a function $f : \{0,1\}^{K_1} \times \{0,1\}^{K_2} \to \{0,1\}^{L_1} \times \{0,1\}^{L_2}$ and a graph problem, we define a family of lower bound graphs in a way similar to Definition 1, replacing item (4) in Definition 1 with a generalized requirement: for $G_{x,y}$, the output values of the nodes in V_A in a solution to the problem uniquely determine the first L_1 bits of $f(x,y)$, and the output values of the of nodes in V_B uniquely determine the last L_2 bits of $f(x,y)$. Next, we argue that theorem similar to Theorem 2 holds for this case.

▶ **Theorem 8.** *Fix a function $f : \{0,1\}^{K_1} \times \{0,1\}^{K_2} \to \{0,1\}^{L_1} \times \{0,1\}^{L_2}$ and a graph problem P. If there is a family $\{G_{x,y}\}$ of lower bound graphs with $C = E(V_A, V_B)$ then any deterministic algorithm for solving P in the CONGEST model requires $\Omega(CC(f)/|C|\log n)$ rounds, and any randomized algorithm for deciding P in the CONGEST model requires $\Omega(CC^R(f)/|C|\log n)$ rounds.*

The proof is similar to that of Theorem 2. Notice that the only difference between the theorems, apart from the sizes of the inputs and outputs of f, are with respect to item (4) in the definition of a family of lower bound graphs. However, the essence of this condition remains the same and is all that is required by the proof: the values that a solution to P assigns to nodes in V_A determines the output of Alice for $f(x,y)$, and the values that a solution to P assigns to nodes in V_B determines the output of Bob for $f(x,y)$.

4.1 A Linear Lower Bound for Weighted APSP

Nanongkai [57] showed that any algorithm in the CONGEST model for computing a poly(n)-approximation for weighted all pairs shortest paths (APSP) requires at least $\Omega(n/\log n)$ rounds. In this section we show that a slight modification to this construction yields an $\Omega(n)$ lower bound for computing exact weighted APSP. As explained in the introduction, this gives a separation between the complexities of the weighted and unweighted versions of APSP. At a high level, while we use the same simple topology for our lower bound as in [57], the reason that we are able to restore the missing logarithmic factor is because our construction uses $O(\log n)$ bits for encoding the weight of each edge out of many optional weights, while in [57] only a single bit is used per edge for encoding one of only two options for its weight.

▶ **Theorem 9.** *Any distributed algorithm in the CONGEST model for computing exact weighted all pairs shortest paths requires at least $\Omega(n)$ rounds.*

The reduction is from the following, perhaps simplest, 2-party communication problem. Alice has an input string x of size K and Bob needs to learn the string of Alice. Any algorithm (possibly randomized) for solving this problem requires at least $\Omega(K)$ bits of communication, by a trivial information theoretic argument.

Notice that the problem of having Bob learn Alice's input is not a binary function as addressed in Section 2. Similarly, computing weighted APSP is not a decision problem, but rather a problem whose solution assigns a value to each node (which is its vector of distances from all other nodes). We therefore use the extended Theorem 8 above.

The fixed graph construction: The fixed graph construction $G = (V, E)$ is defined as follows. It contains a set of $n - 2$ nodes, denoted $A = \{a_0, ..., a_{n-3}\}$, which are all connected to an additional node a. The node a is connected to the last node b, by an edge of weight 0.

Adding edge weights corresponding to the string x: Given the binary string x of size $K = (n-2)\log n$ we augment the graph G with edge weights, which defines G_x, by having each non-overlapping batch of $\log n$ bits encode a weight of an edge from A to a. It is straightforward to see that G_x is a family of lower bound graphs for a function f where $K_2 = L_1 = 0$, since the weights of the edges determine the right-hand side of the output (while the left-hand side is empty).

Proof of Theorem 9. To prove Theorem 9, we let $V_A = A \cup \{a\}$ and $V_B = \{b\}$. Note that $K = |x| = \Theta(n \log n)$. Furthermore, note that the only edge in the cut $E(V_A, V_B)$ is the edge (a, b). Since we showed that $\{G_x\}$ is a family of lower bound graphs, we apply Theorem 8 on the above partition to deduce that because K bits are required to be communicated in order for Bob to know Alice's K-bit input, any algorithm in the CONGEST model for computing weighted APSP requires at least $\Omega(K/\log n) = \Omega(n)$ rounds. ◀

4.2 The Alice-Bob Framework Cannot Give a Super-Linear Lower Bound for Weighted APSP

In this section we argue that a reduction from any 2-party function with a constant partition of the graph into Alice and Bob's sides is provable incapable of providing a super-linear lower bound for computing weighted all pairs shortest paths in the CONGEST model. A more detailed inspection of our analysis shows a stronger claim: our claim also holds for algorithms for the CONGEST-BROADCAST model, where in each round each node must send the same $(\log n)$-bit message to all of its neighbors. The following theorem states our claim.

▶ **Theorem 10.** *Let $f : \{0,1\}^{K_1} \times \{0,1\}^{K_2} \to \{0,1\}^{L_1} \times \{0,1\}^{L_2}$ be a function and let $G_{x,y}$ be a family of lower bound graphs w.r.t. f and the weighted APSP problem. When applying Theorem 8 to f and $G_{x,y}$, the lower bound obtained for the number of rounds for computing weighted APSP is at most linear in n.*

Roughly speaking, we show that given an input graph $G = (V, E)$ and a partition of the set of vertices into two sets $V = V_A \cup V_B$, such that the graph induced by the nodes in V_A is simulated by Alice and the graph induced by nodes in V_B is simulated by Bob, Alice and Bob can compute weighted all pairs shortest paths by communicating $O(n \log n)$ bits of information for each node touching the cut $C = (V_A, V_B)$ induced by the partition. This means that for any 2-party function f and any family of lower bound graphs w.r.t. f and weighted APSP according to the extended definition of Section 4.1, since Alice and Bob can compute weighted APSP which determines their output for f by exchanging only $O(|V(C)|n \log n)$ bits, where $V(C)$ is the set of nodes touching C, the value $CC(f)$ is at most $O(|V(C)|n \log n)$. But then the lower bound obtained by Theorem 8 cannot be better than $\Omega(n)$, and hence no super-linear lower can be deduced by this framework as is.

Formally, given a graph $G = (V = V_A \dot\cup V_B, E)$ we denote $C = E(V_A, V_B)$. Let $V(C)$ denote the nodes touching the cut C, with $C_A = V(C) \cap V_A$ and $C_B = V(C) \cap V_B$. Let $G_A = (V_A, E_A)$ be the subgraph induced by the nodes in V_A and let $G_B = (V_B, E_B)$ be the subgraph induced by the nodes in V_B. For a graph H, we denote the weighted distance between two nodes u, v by $\text{wdist}_H(u, v)$.

▶ **Lemma 11.** *Let $G = (V = V_A \dot\cup V_B, E, w)$ be a graph with an edge-weight function $w : E \to \{1, \ldots, W\}$, such that $W \in \text{poly}\, n$. Suppose that G_A, C_B, C and the values of w on E_A and C are given as input to Alice, and that G_B, C_A, C and the values of w on E_B and C are given as input to Bob.*

Then, Alice can compute the distances in G from all nodes in V_A to all nodes in V and Bob can compute the distances from all nodes in V_B to all the nodes in V, using $O(|V(C)|\, n \log n)$ bits of communication.

Proof. We describe a protocol for the required computation, as follows. For each node $u \in C_B$, Bob sends to Alice the weighted distances in G_B from u to all nodes in V_B, that is, Bob sends $\{\text{wdist}_{G_B}(u, v) \mid u \in C_B, v \in V_B\}$ (or ∞ for pairs of nodes not connected in G_B). Alice constructs a virtual graph $G'_A = (V'_A, E'_A, w'_A)$ with the nodes $V'_A = V_A \cup C_B$ and edges $E'_A = E_A \cup C \cup (C_B \times C_B)$. The edge-weight function w'_A is defined by $w'_A(e) = w(e)$ for each $e \in E_A \cup C$, and $w'_A(u, v)$ for $u, v \in C_B$ is defined to be the weighted distance between u and v in G_B, as received from Bob. Alice then computes the set of all weighted distances in G'_A, $\{\text{wdist}_{G'_A}(u, v) \mid u, v \in V'_A\}$.

Alice assigns her output for the weighted distances in G as follows. For two nodes $u, v \in V_A \cup C_B$, Alice outputs their weighted distance in G'_A, $\text{wdist}_{G'_A}(u, v)$. For a node $u \in V'_A$ and a node $v \in V_B \setminus C_B$, Alice outputs $\min\{\text{wdist}_{G'_A}(u, x) + \text{wdist}_{G_B}(x, v) \mid x \in C_B\}$, where $\text{wdist}_{G'_A}$ is the distance in G'_A as computed by Alice, and wdist_{G_B} is the distance in G_B that was sent by Bob.

For Bob to compute his required weighted distances, for each node $u \in C_A$, similar information is sent by Alice to Bob, that is, Alice sends to Bob the weighted distances in G_A from u to all nodes in V_A. Bob constructs the analogous graph G'_B and outputs his required distance. The next paragraph formalizes this for completeness, but may be skipped by a convinced reader.

Formally, Alice sends $\{\text{wdist}_{G_A}(u, v) \mid u \in C_A, v \in V_A\}$. Bob constructs $G'_B = (V'_B, E'_B, w'_B)$ with $V'_B = V_B \cup C_A$ and edges $E'_B = E_B \cup C \cup (C_A \times C_A)$. The edge-weight function w'_B is defined by $w'_B(e) = w(e)$ for each $e \in E_B \cup C$, and $w'_B(u, v)$ for $u, v \in C_A$ is defined to be the weighted distance between u and v in G_A, as received from Alice (or ∞ if they are not connected in G_A). Bob then computes the set of all weighted distances in G'_B, $\{\text{wdist}_{G'_B}(u, v) \mid u, v \in V'_B\}$. Bob assigns his output for the weighted distances in G as follows. For two nodes $u, v \in V_B \cup C_A$, Bob outputs their weighted distance in G'_B, $\text{wdist}_{G'_B}(u, v)$. For a node $u \in V'_B$ and a node $v \in V_A \setminus C_A$, Bob outputs $\min\{\text{wdist}_{G'_B}(u, x) + \text{wdist}_{G_A}(x, v) \mid x \in C_A\}$, where $\text{wdist}_{G'_B}$ is the distance in G'_B as computed by Bob, and wdist_{G_A} is the distance in G_A that was sent by Alice. ◀

The proof of Theorem 10 appears in the full version of the paper [18].

▶ Remark. In the full version of the paper [18] we show that generalizing the Alice-Bob framework to a shared-blackboard multi-party setting is still insufficient for providing a super-linear lower bound for weighted APSP. We suspect that a similar argument can be applied for the framework of non-fixed Alice-Bob partitions (e.g., [65]), but this requires precisely defining these frameworks which is not addressed here.

5 Discussion

This work provides the first super-linear lower bounds for the CONGEST model, raising a plethora of open questions. First, we showed for some specific problems, namely, computing a minimum vertex cover, a maximum independent set and a χ-coloring, that they are nearly as hard as possible for the CONGEST model. However, we know that approximate solutions for some of these problems can be obtained much faster, in a polylogarithmic number of rounds or even less. A family of specific open questions is then to characterize the exact trade-off between approximation factors and round complexities for these problems.

Another specific open question is the complexity of weighted APSP, which has also been asked in previous work [27, 57]. Our proof that the Alice-Bob framework is incapable of providing super-linear lower bounds for this problem may be viewed as providing evidence that weighted APSP can be solved much faster than is currently known. Together with the recent sub-quadratic algorithm of [29], this brings another angle to the question: can weighted APSP be solved in linear time?

Finally, we propose a more general open question which addresses a possible classification of complexities of global problems in the CONGEST model. Some such problems have complexities of $\Theta(D)$, such as constructing a BFS tree. Others have complexities of $\tilde{\Theta}(D+\sqrt{n})$, such as finding an MST. Some problems have near-linear complexities, such as unweighted APSP. And now we know about the family of hardest problems for the CONGEST model, whose complexities are near-quadratic. Do these complexities capture all possibilities, when natural global graph problems are concerned? Or are there such problems with a complexity of, say, $\Theta(n^{1+\delta})$, for some constant $0 < \delta < 1$? A similar question was recently addressed in [20] for the LOCAL model, and we propose investigating the possibility that such a hierarchy exists for the CONGEST model.

Acknowledgements. We thank Amir Abboud, Ohad Ben Baruch, Michael Elkin, Yuval Filmus and Christoph Lenzen for useful discussions, and the anonymous reviewers.

References

1 Amir Abboud, Keren Censor-Hillel, and Seri Khoury. Near-linear lower bounds for distributed distance computations, even in sparse networks. In *30th International Symposium on Distributed Computing, DISC*, pages 29–42, 2016.

2 Amir Abboud and Kevin Lewi. Exact weight subgraphs and the k-sum conjecture. In *40th International Colloquium on Automata, Languages, and Programming, ICALP*, pages 1–12, 2013.

3 Amir Abboud, Kevin Lewi, and Ryan Williams. Losing weight by gaining edges. In *22th Annual European Symposium on Algorithms, ESA*, pages 1–12, 2014.

4 Noga Alon, Raphael Yuster, and Uri Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997.

5 Matti Åstrand, Patrik Floréen, Valentin Polishchuk, Joel Rybicki, Jukka Suomela, and Jara Uitto. A local 2-approximation algorithm for the vertex cover problem. In *23rd International Symposium on Distributed Computing, DISC*, pages 191–205, 2009.

6 Matti Åstrand and Jukka Suomela. Fast distributed approximation algorithms for vertex cover and set cover in anonymous networks. In *22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA*, pages 294–302, 2010.

7 Reuven Bar-Yehuda, Keren Censor-Hillel, Mohsen Ghaffari, and Gregory Schwartzman. Distributed approximation of maximum independent set and maximum matching. In *ACM Symposium on Principles of Distributed Computing, PODC*, 2017.

8 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. In *2016 ACM Symposium on Principles of Distributed Computing, PODC*, pages 3–8, 2016.

9 Leonid Barenboim. On the locality of some NP-complete problems. In *39th International Colloquium on Automata, Languages, and Programming, ICALP*, pages 403–415, 2012.

10 Leonid Barenboim. Deterministic $(\Delta + 1)$-coloring in sublinear (in Δ) time in static, dynamic, and faulty networks. *J. ACM*, 63(5):47:1–47:22, 2016.

11 Leonid Barenboim and Michael Elkin. Deterministic distributed vertex coloring in polylogarithmic time. *J. ACM*, 58(5):23:1–23:25, 2011.

12 Leonid Barenboim and Michael Elkin. Combinatorial algorithms for distributed graph coloring. *Distributed Computing*, 27(2):79–93, 2014.

13 Leonid Barenboim, Michael Elkin, and Fabian Kuhn. Distributed (delta+1)-coloring in linear (in delta) time. *SIAM J. Comput.*, 43(1):72–95, 2014.

14 Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The locality of distributed symmetry breaking. *J. ACM*, 63(3):20:1–20:45, 2016.

15 Marijke H. L. Bodlaender, Magnús M. Halldórsson, Christian Konrad, and Fabian Kuhn. Brief announcement: Local independent set approximation. In *ACM Symposium on Principles of Distributed Computing, PODC*, pages 93–95, 2016.

16 Keren Censor-Hillel, Petteri Kaski, Janne H. Korhonen, Christoph Lenzen, Ami Paz, and Jukka Suomela. Algebraic methods in the congested clique. In *2015 ACM Symposium on Principles of Distributed Computing, PODC*, 2015.

17 Keren Censor-Hillel, Telikepalli Kavitha, Ami Paz, and Amir Yehudayoff. Distributed construction of purely additive spanners. *Distributed Computing*, 2017.

18 Keren Censor-Hillel, Seri Khoury, and Ami Paz. Quadratic and near-quadratic lower bounds for the CONGEST model. *CoRR*, abs/1705.05646, 2017. URL: `https://arxiv.org/abs/1705.05646`.

19 Yi-Jun Chang, Tsvi Kopelowitz, and Seth Pettie. An exponential separation between randomized and deterministic complexity in the LOCAL model. In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS*, pages 615–624, 2016.

20 Yi-Jun Chang and Seth Pettie. A time hierarchy theorem for the LOCAL model. *CoRR*, abs/1704.06297, 2017.

21 Kai-Min Chung, Seth Pettie, and Hsin-Hao Su. Distributed algorithms for the lovász local lemma and graph coloring. In *ACM Symposium on Principles of Distributed Computing, PODC*, pages 134–143, 2014.

22 Richard Cole and Uzi Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32–53, 1986.

23 Andrzej Czygrinow, Michal Hanckowiak, and Wojciech Wawrzyniak. Fast distributed approximations in planar graphs. In *22nd International Symposium on Distributed Computing, DISC*, pages 78–92, 2008.

24 Søren Dahlgaard, Mathias Bæk Tejs Knudsen, and Morten Stöckel. Finding even cycles faster via capped k-walks. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC*, pages 112–120, 2017.

25 Danny Dolev, Christoph Lenzen, and Shir Peled. Tri, Tri Again: Finding triangles and small subgraphs in a distributed setting. In *26th International Symposium on Distributed Computing, DISC*, pages 195–209, 2012.

26 Andrew Drucker, Fabian Kuhn, and Rotem Oshman. On the power of the congested clique model. In *33rd ACM Symposium on Principles of Distributed Computing, PODC*, pages 367–376, 2014.

27 Michael Elkin. Distributed approximation: a survey. *SIGACT News*, 35(4):40–57, 2004.

28 Michael Elkin. An unconditional lower bound on the time-approximation trade-off for the distributed minimum spanning tree problem. *SIAM J. Comput.*, 36(2):433–456, 2006.

29 Michael Elkin. Distributed exact shortest paths in sublinear time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC*, pages 757–770, 2017.

30 Yuval Emek, Christoph Pfister, Jochen Seidel, and Roger Wattenhofer. Anonymous networks: randomization = 2-hop coloring. In *ACM Symposium on Principles of Distributed Computing, PODC*, pages 96–105, 2014.

31 Pierre Fraigniaud, Cyril Gavoille, David Ilcinkas, and Andrzej Pelc. Distributed computing with advice: information sensitivity of graph coloring. *Distributed Computing*, 21(6):395–403, 2009.

32 Pierre Fraigniaud, Marc Heinrich, and Adrian Kosowski. Local conflict coloring. In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS*, pages 625–634, 2016.

33 Silvio Frischknecht, Stephan Holzer, and Roger Wattenhofer. Networks cannot compute their diameter in sublinear time. In *ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 1150–1162, 2012.

34 Mika Göös and Jukka Suomela. Locally checkable proofs in distributed computing. *Theory of Computing*, 12(1):1–33, 2016.

35 Fabrizio Grandoni, Jochen Könemann, and Alessandro Panconesi. Distributed weighted vertex cover via maximal matchings. *ACM Trans. Algorithms*, 5(1):6:1–6:12, 2008.

36 Fabrizio Grandoni, Jochen Könemann, Alessandro Panconesi, and Mauro Sozio. A primal-dual bicriteria distributed algorithm for capacitated vertex cover. *SIAM J. Comput.*, 38(3):825–840, 2008.

37 Michal Hanckowiak, Michal Karonski, and Alessandro Panconesi. On the distributed complexity of computing maximal matchings. *SIAM J. Discrete Math.*, 15(1):41–57, 2001.

38 David G. Harris, Johannes Schneider, and Hsin-Hao Su. Distributed $(\Delta+1)$-coloring in sublogarithmic rounds. In *48th Annual ACM SIGACT Symposium on Theory of Computing, STOC*, pages 465–478, 2016.

39 Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. A deterministic almost-tight distributed algorithm for approximating single-source shortest paths. In *48th Annual ACM SIGACT Symposium on Theory of Computing, STOC*, pages 489–498, 2016.

40 Stephan Holzer, David Peleg, Liam Roditty, and Roger Wattenhofer. Distributed 3/2-approximation of the diameter. In *28th International Symposium on Distributed Computing, DISC*, pages 562–564, 2014.

41 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*, pages 6:1–6:16, 2015.

42 Stephan Holzer and Roger Wattenhofer. Optimal distributed all pairs shortest paths and applications. In *ACM Symposium on Principles of Distributed Computing, PODC*, pages 355–364, 2012.

43 Qiang-Sheng Hua, Haoqiang Fan, Lixiang Qian, Ming Ai, Yangyang Li, Xuanhua Shi, and Hai Jin. Brief announcement: A tight distributed algorithm for all pairs shortest paths and applications. In *28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA*, pages 439–441, 2016.

44 Allan Grønlund Jørgensen and Seth Pettie. Threesomes, degenerates, and love triangles. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS*, pages 621–630, 2014.

45 Samir Khuller, Uzi Vishkin, and Neal E. Young. A primal-dual parallel approximation technique applied to weighted set and vertex covers. *J. Algorithms*, 17(2):280–289, 1994.

46 Liah Kor, Amos Korman, and David Peleg. Tight bounds for distributed MST verification. In *28th International Symposium on Theoretical Aspects of Computer Science, STACS*, pages 69–80, 2011.

47 Christos Koufogiannakis and Neal E. Young. Distributed and parallel algorithms for weighted vertex cover and other covering problems. In *28th Annual ACM Symposium on Principles of Distributed Computing, PODC*, pages 171–179, 2009.

48 Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. The price of being near-sighted. In *17th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 980–989, 2006.

49 Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. Local computation: Lower and upper bounds. *J. ACM*, 63(2):17:1–17:44, 2016.

50 Eyal Kushilevitz and Noam Nisan. *Communication Complexity*. Cambridge University Press, New York, 1997.

51 Christoph Lenzen and Boaz Patt-Shamir. Fast partial distance estimation and applications. In *2015 ACM Symposium on Principles of Distributed Computing, PODC*, 2015.

52 Christoph Lenzen and David Peleg. Efficient distributed source detection with limited bandwidth. In *ACM Symposium on Principles of Distributed Computing, PODC*, pages 375–382, 2013.

53 Christoph Lenzen and Roger Wattenhofer. Leveraging linial's locality limit. In *22nd International Symposium on Distributed Computing, DISC*, pages 394–407, 2008.

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

55 Dániel Marx and Michal Pilipczuk. Everything you always wanted to know about the parameterized complexity of subgraph isomorphism (but were afraid to ask). In *31st International Symposium on Theoretical Aspects of Computer Science, STACS*, pages 542–553, 2014.

56 Thomas Moscibroda and Roger Wattenhofer. Coloring unstructured radio networks. *Distributed Computing*, 21(4):271–284, 2008.

57 Danupon Nanongkai. Distributed approximation algorithms for weighted shortest paths. In *Symposium on Theory of Computing, STOC,*, pages 565–573, 2014.

58 Danupon Nanongkai, Atish Das Sarma, and Gopal Pandurangan. A tight unconditional lower bound on distributed randomwalk computation. In *30th Annual ACM Symposium on Principles of Distributed Computing, PODC*, pages 257–266, 2011.

59 Alessandro Panconesi and Romeo Rizzi. Some simple distributed algorithms for sparse networks. *Distributed Computing*, 14(2):97–100, 2001.

60 David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

61 David Peleg, Liam Roditty, and Elad Tal. Distributed algorithms for network diameter and girth. In *39th International Colloquium on Automata, Languages, and Programming, ICALP*, pages 660–672, 2012.

62 David Peleg and Vitaly Rubinovich. A near-tight lower bound on the time complexity of distributed minimum-weight spanning tree construction. *SIAM J. Comput.*, 30(5):1427–1442, 2000.

63 Seth Pettie and Hsin-Hao Su. Distributed coloring algorithms for triangle-free graphs. *Inf. Comput.*, 243:263–280, 2015.

64 Valentin Polishchuk and Jukka Suomela. A simple local 3-approximation algorithm for vertex cover. *Inf. Process. Lett.*, 109(12):642–645, 2009.

65 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 J. Comput.*, 41(5):1235–1265, 2012.

66 Johannes Schneider and Roger Wattenhofer. Distributed coloring depending on the chromatic number or the neighborhood growth. In *18th International Colloquium on Structural Information and Communication Complexity, SIROCCO*, pages 246–257, 2011.

67 Virginia Vassilevska Williams and Ryan Williams. Finding, minimizing, and counting weighted subgraphs. *SIAM J. Comput.*, 42(3):831–854, 2013.

Derandomizing Local Distributed Algorithms under Bandwidth Restrictions*†

Keren Censor-Hillel[1], Merav Parter[2], and Gregory Schwartzman[3]

1 **Department of Computer Science, Technion, Haifa, Israel**
ckeren@cs.technion.ac.il
2 **MIT, CSAIL, Cambridge, USA**
parter@mit.edu
3 **Department of Computer Science, Technion, Haifa, Israel**
gregory.schwartzman@gmail.com

Abstract

This paper addresses the cornerstone family of *local problems* in distributed computing, and investigates the curious gap between randomized and deterministic solutions under bandwidth restrictions.

Our main contribution is in providing tools for derandomizing solutions to local problems, when the n nodes can only send $O(\log n)$-bit messages in each round of communication. We combine bounded independence, which we show to be sufficient for some algorithms, with the method of conditional expectations and with additional machinery, to obtain the following results.

First, we show that in the *Congested Clique* model, which allows all-to-all communication, there is a deterministic maximal independent set (MIS) algorithm that runs in $O(\log^2 \Delta)$ rounds, where Δ is the maximum degree. When $\Delta = O(n^{1/3})$, the bound improves to $O(\log \Delta)$.

Adapting the above to the CONGEST model gives an $O(D \log^2 n)$-round deterministic MIS algorithm, where D is the diameter of the graph. Apart from a previous unproven claim of a $O(D \log^3 n)$-round algorithm, the only known deterministic solutions for the CONGEST model are a coloring-based $O(\Delta + \log^* n)$-round algorithm, where Δ is the maximal degree in the graph, and a $2^{O(\sqrt{\log n \log\log n})}$-round algorithm, which is super-polylogarithmic in n.

In addition, we deterministically construct a $(2k-1)$-spanner with $O(kn^{1+1/k} \log n)$ edges in $O(k \log n)$ rounds in the Congested Clique model. For comparison, in the more stringent CONGEST model, where the communication graph is identical to the input graph, the best deterministic algorithm for constructing a $(2k-1)$-spanner with $O(kn^{1+1/k})$ edges runs in $O(n^{1-1/k})$ rounds.

1998 ACM Subject Classification G.2.2 Graph Algorithms

Keywords and phrases Local problems, congested clique, derandomization

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.11

1 Introduction

1.1 Motivation

A cornerstone family of problems in distributed computing are the so-called *local problems*. These include finding a maximal independent set (MIS), a $(\Delta+1)$-coloring where Δ is the

* This research is partially supported by the Israel Science Foundation (grant 1696/14).
† A full version of the paper is available at https://arxiv.org/abs/1608.01689.


© Keren Censor-Hillel, Merav Parter, and Gregory Schwartzman;
licensed under Creative Commons License CC-BY

maximal degree in the network graph, finding a maximal matching, constructing multiplicative spanners, and more. Intuitively, as opposed to *global problems*, local problems admit solutions that do not require communication over the entire network graph.

One fundamental characteristic of distributed algorithms for local problems is whether they are deterministic or randomized. Currently, there exists a curious gap between the known complexities of randomized and deterministic solutions for local problems. Interestingly, the main indistinguishability-based technique used for obtaining the relatively few lower bounds that are known seems unsuitable for separating these cases. A beautiful recent work of Chang et al. [14] sheds some light over this, by proving that the randomized complexity of any local problem is at least its deterministic complexity on instances of size $\sqrt{\log n}$. In addition, building upon a new lower bound technique of Brandt et al. [10], they show an exponential separation between the randomized and deterministic complexity of Δ-coloring trees. These results hold in the *LOCAL* model, which allows unbounded messages.

In this paper, we address the tension between the deterministic and randomized complexities of local problems in the *congested clique* model, where the communication graph is complete but the size of messages is restricted to $O(\log n)$ bits. The processed graph is an arbitrary input graph which, in contrast to the LOCAL model, is not necessarily the same as the communication graph. In some sense, the congested clique model is orthogonal to the LOCAL model, because the diameter of the communication graph is 1, but the size of messages is restricted. By showing how to derandomize known algorithms for the LOCAL model, we provide fast deterministic algorithms for constructing an MIS and multiplicative spanners in the congested clique model.

The curious phenomenon that shows up here is that the derandomization toolbox that was developed for sequential algorithms does not seem to lend itself for the LOCAL model, but it can be used in the congested clique model. This allows us to obtain deterministic algorithms for local problems in the congested clique model, whose complexities roughly match the complexities of their randomized counterparts in the LOCAL model. This can be contrasted with the exponential in Δ or near-exponential in n gaps between the deterministic and randomized complexities of these problems in the LOCAL model alone.

1.2 Our Contribution

Maximal Independent Set (MIS): We begin by derandomizing the MIS algorithm of Ghaffari [26], which runs in $O(\log \Delta) + 2^{O(\sqrt{\log \log n})}$ rounds, w.h.p[1]. In a nutshell, in this algorithm, nodes choose to mark themselves with probabilities that evolve depending on the previous probabilities of neighbors. In particular, if the sum of marking probabilities of a vertex's neighbors is large (resp., small) – the vertex reduces (resp. , increases) its own marking probability in the next round. A marked node that does not have any marked neighbors joins the MIS and all of its neighbors remove themselves from the graph. The analysis shows that after $O(\log \Delta)$ phases the graph consists of a convenient decomposition into small clusters for which the problem can be solved fast. This is called the shattering phenomena (see e.g., [7]).

We first show that a tighter analysis for the congested clique model of Ghaffari's MIS algorithm can improve its running time from $O(\log \Delta + \log^* n)$ (which follows from combining [26] with the new connectivity result of [27]) to $O(\log \Delta)$ rounds.

[1] As standard, *with high probability* means with probability that is at least $1 - 1/n^c$ for a constant c.

▶ **Theorem 1.** *There is a randomized algorithm that computes MIS in the congested clique model within $O(\log \Delta)$ rounds with high probability.*

For the derandomization, we use the method of conditional expectations (see e.g., [44, Chapter 6.3]). In our context, this shows the existence of an assignment to the random choices made by the nodes that attains the desired property of removing a sufficiently large part of the graph in each iteration, where removal is due to a node already having an output (whether the vertex is in the MIS or not). As in many uses of this method, we need to reduce the number of random choices that are made in order to be able to efficiently *compute* the above assignment.

However, we need to overcome several obstacles. First, we need to reduce the search space of a good assignment to the random choices of the nodes, by showing that pairwise independence (see, e.g., [44, Chapter 13]) is sufficient for the algorithm to work. Unfortunately, this does not hold directly in the original algorithm.

The first key ingredient is a slight modification of the constants used by Ghaffari's algorithm. Ghaffari's analysis is based on a definition of *golden nodes*, which are nodes that have a constant probability of being removed in the given phase. We show that this removal-probability guarantee holds also with pairwise independence upon our slight adaptation of the constants used by the algorithm.

Second, the shattering effect that occurs after $O(\log \Delta)$ rounds of Ghaffari's algorithm with *full independence*, no longer holds under pairwise independence. Instead, we take advantage of the fact that in the congested clique model, once the remaining graph has a linear number of edges then the problem can be solved locally in constant many rounds using Lenzen's routing algorithm [38]. Thus, we modify the algorithm so that after $O(\log \Delta)$ rounds, the remaining graph (containing all undecided nodes) contains $O(n)$ edges. The crux in obtaining this is that during the first $O(\log \Delta)$ phases, we favor the removal of *old* nodes, which, roughly speaking, are nodes that had many rounds in which they had a good probability of being removed. This prioritized (or biased) removal strategy allows us to employ an amortized (or accounting) argument to claim that every node that survives $O(\log \Delta)$ rounds, can blame a distinct set of Δ nodes for not being removed earlier. Hence, the total number of remaining nodes is bounded by $O(n/\Delta)$, implying a remaining number of edges of $O(n)$.

To simulate the $O(\log \Delta)$ randomized rounds of Ghaffari's algorithm, we enjoy the small search space (due to pairwise independence) and employ the method of conditional expectations on a random seed of length $O(\log n)$. Note that once we start conditioning on random variables in the seed, the random choices are no longer pairwise independent as they are in the unconditioned setting. However, we do not use the pairwise independence in the conditioning process. That is, the pairwise independence is important in showing that the *unconditional expectation* is large, and from that point on the conditioning does not reduce this value. As typical in MIS algorithms, the probability of a node being removed stems from the random choices made in its 2-neighborhood. With a logarithmic bandwidth, collecting this information is too costly. Instead, we use a pessimistic estimator to *bound* the conditional probabilities rather than compute them.

Finally, to make the decision of the partial assignment and inform the nodes, we leverage the power of the congested clique by having a leader that collects the relevant information for coordinating the decision regarding the partial assignment. In fact, the algorithm works in the more restricted *Broadcast Congested Clique* model, in which a node must send the same $O(\log n)$-bit message to all other nodes in any single round. Carefully placing all the pieces of the toolbox we develop, gives the following.

▶ **Theorem 2.** *There is a deterministic MIS algorithm for the broadcast congested clique model that completes in* $O(\log\Delta\log n)$ *rounds.*

If the maximal degree satisfies $\Delta = O(n^{1/3})$ then we can improve the running time in the congested clique model. The proof of the following is deferred to the full version [12].

▶ **Theorem 3.** *If* $\Delta = O(n^{1/3})$ *then there is a deterministic MIS algorithm for the congested clique model that completes in* $O(\log\Delta)$ *rounds.*

Combining Theorems 2 and 3 directly gives that the complexity is either $O(\log\Delta)$ rounds in case $\Delta = O(n^{1/3})$, and otherwise it is $O(\log^2\Delta)$ since $\log n$ is then asymptotically equal to $\log\Delta$. We conclude that there is a deterministic MIS algorithm for the congested clique model that completes in $O(\log^2\Delta)$ rounds.

Our techniques immediately extend to the CONGEST model. The state of the art for that setting is $O(2^{\sqrt{\log n\log\log n}})$ round algorithm, using the network decomposition of [5] (see Cor. 5.4 there). Wes then show that MIS can be computed in $O(D\cdot\log^2 n)$ rounds where D is the diameter of the graph. Here, we simulate $O(\log n)$ rounds of Ghaffari's algorithm rather than $O(\log\Delta)$ rounds as before. Each such randomized round is simulated by using $O(D\cdot\log n)$ deterministic rounds in which the nodes compute an $O(\log n)$ seed. Computing each bit of the seed, requires aggregation of the statistics to a leader which can be done in $O(D)$ rounds, and since the seed is of length $O(\log n)$, we have the following:

▶ **Theorem 4.** *There is a deterministic MIS algorithm for the CONGEST model that completes in* $\min\{O(D\log^2 n), O(2^{\sqrt{\log n\log\log n}})\}$ *rounds.*

The significance of the latter is that it is the first deterministic MIS algorithm in CONGEST to have only a polylogarithmic gap compared to its randomized counterpart when D is polylogarithmic. Notice that this logarithmic complexity is the best that is known even in the LOCAL model. In [49] it is shown that an MIS can be computed deterministically in $2^{O(\sqrt{\log n})}$ rounds via network decomposition, which is super-polylogarithmic in n. Moreover, the algorithm requires large messages and hence is unsuitable for CONGEST. Focusing on deterministic algorithms in CONGEST, the only known non-trivial solution is to use any $(\Delta+1)$-coloring algorithm running in $O(\Delta+\log^* n)$ rounds (for example [3, 6]) to obtain the same complexity for deterministic MIS in CONGEST (notice that there are faster coloring algorithms, e.g., [7], but the reduction has to pay for the number of colors anyhow). Our $O(D\log^2 n)$-round MIS algorithm is therefore unique in its parameters.

Multiplicative Spanners: We further exemplify our techniques in order to derandomize the Baswana-Sen algorithm for constructing a multiplicative spanner. For an integer k, a k-spanner S of $G=(V,E)$ is a subgraph (V,E_S) such that for every two neighbors v,u in G, their distance in S is at most k. This implies that also the distance for every other pair of nodes is stretched in S by no more than a multiplicative factor of k. The Baswana-Sen algorithm runs in $O(k^2)$ rounds and produces a $(2k-1)$-spanner with $O(kn^{1+1/k})$ edges. In a nutshell, the algorithm starts with a clustering defined by all singletons and proceeds with k iterations, in each of which the clusters get sampled with probability $n^{-1/k}$ and each node joins a neighboring sampled cluster or adds edges to unsampled clusters.

We need to make several technical modifications of our tools for this to work. The key technical difficulty is that we cannot have a single target function. This arises from the very nature of spanners, in that a small-stretch spanner always exists, but the delicate part is to balance between the stretch and the number of edges. This means that a single function

which takes care of having a good stretch alone will simply result in taking all the edges into the spanner, as this gives the smallest stretch. We overcome this challenge by defining two types of bad events which the algorithm tries to avoid simultaneously. One is that too many clusters get sampled, and the other is that too many nodes add too many edges to the spanner in this iteration. The careful balance between the two promises that we can indeed get the desired stretch and almost the same bound on the number of edges.

Additional changes we handle are that when we reduce the independence, we cannot go all the way down to pairwise independence and we need settle for d-wise independence, where $d = \Theta(\log n)$. Further, we can improve the iterative procedure to handle chunks of $\log n$ random bits, and evaluate them in parallel by assigning a different leader to each possible assignment for them. A careful analysis gives a logarithmic overhead compared to the original Baswana-Sen algorithm, but we also save a factor of k since the congested clique allows us to save the k rounds needed in an iteration of Baswana-Sen for communicating with the center of the cluster. This gives the following.

▶ **Theorem 5.** *There is a deterministic algorithm for the congested clique model that completes in $O(k \log n)$ rounds and produces a $(2k-1)$-spanner with $O(kn^{1+1/k} \log n)$ edges.*

As in the MIS algorithm, the above algorithm works also in the broadcast congested clique model, albeit here we lose the ability to parallelize over many leaders and thus we pay another logarithmic factor in the number of rounds, resulting in $O(k \log^2 n)$ rounds. The entire spanner construction is deferred to the full version [12].

1.3 Related Work

Distributed computation of MIS. The complexity of finding a maximal independent set is a central problem in distributed computing and hence has been extensively studied. The $O(\log n)$-round randomized algorithms date back to 1986, and were given by Luby [42], Alon et al. [1] and Israeli and Itai [35]. [7] showed a randomized MIS algorithm with $O(\log^2 \Delta) + 2^{O(\sqrt{\log \log n})}$ rounds. They also showed the bound of $O(\log \Delta) + 2^{O(\sqrt{\log \log n})}$ rounds for Maximal Matching and $(\Delta+1)$-coloring. Following [7], a recent breakthrough by Ghaffari [26] obtained a randomized algorithm in $O(\log \Delta) + 2^{O(\sqrt{\log \log n})}$ rounds.

The best deterministic algorithm is by Panconesi and Srinivasan [48], and completes in $2^{O(\sqrt{\log n})}$ rounds. On the lower bound side, Linial [40] gave an $\Omega(\log^* n)$ lower bounds for 3-coloring the ring, which also applies to finding an MIS. Kuhn et al. [37] gave lower bounds of $\Omega(\sqrt{\log n / \log \log n})$ and $\Omega(\sqrt{\log \Delta / \log \log \Delta})$ for finding an MIS.

Barenboim and Elkin [4] provide a thorough tour on coloring algorithms (naturally, excluding recent results). An excellent survey on local problems is given by Suomela [55].

Distributed constructions of spanners. The construction of spanners in the distribute setting has been studied extensively both in the randomized and deterministic setting [8, 15, 16, 17, 18, 52]. We emphasize that the construction of [18] cannot be implemented in the congested clique by simply applying Lenzen's routing scheme because although each node sends $O(n \log n)$ bits of information, this information may need to be received by many nodes, and is not split among receivers. A randomized spanner construction was given by Baswana and Sen in [8]. They show that their well-known centralized algorithm can be implemented in the distributed setting even with small messages. In particular, they show that a $(2k-1)$ spanner with an expected number of $O(n^{1+1/k})$ edges can be constructed in $O(k^2)$ rounds in the CONGEST model (and for unweighted graphs, the algorithm takes $O(k)$ rounds, see [23]).

Derandomization of similar randomized algorithms has been addressed mainly in the *centralized* setting [53]. We emphasize that we need entirely different techniques to derandomize the Baswana-Sen algorithm compared with the centralized derandomization of [53].

The existing *deterministic* distributed algorithms for spanner are not based on derandomization of the randomized construction. They mostly use messages of *unbounded* size and are mainly based on sparse partitions or network decomposition. The state of the art is due to Derbel et al [17]. They provide a *tight* algorithm for constructing $(2k-1)$-spanners with optimal stretch, size and construction time of k rounds. This was complemented by a matching lower bound, showing that any (even randomized) distributed algorithm requires k rounds in expectation. Much less efficient deterministic algorithms are known for the CONGEST model. [19] showed a construction of a $(2k-1)$-spanner in $O(n^{1-1/k})$ rounds. Deterministic construction with an improved tradeoff was recently obtained by [5], they showed a construction of $O(\log^{k-1} n)$-spanners with $O(n^{1+1/k})$ edges in $O(\log^{k-1} n)$ rounds.

Algorithms in the congested clique. The congested clique model was first addressed in Lotker et al. [41], who raised the question of whether the global problem of constructing a minimum spanning tree (MST) can be solved faster on a communication graph with diameter 1. Since then, the model gained much attention, with results about its computational power given in [21], faster MST algorithms [27, 30], distance computation [33, 34, 46], subgraph detection [20], algebraic computations [11, 25], and routing and sorting [38, 39, 51]. Local problems were addressed in [32] who study ruling sets. Connections to the MapReduce model is given in [31].

Derandomization in the parallel setting. Derandomization of local algorithms has attracted much attention in the *parallel* setting [1, 9, 13, 28, 29, 35, 36, 45, 50, 54]. Luby [43] showed that his MIS algorithm (and more) can be derandomized in the PRAM model using $O(m)$ machines and $O(\log^3 n \log\log n)$ time. In fact, this much simpler algorithm can also be executed on the congested clique model, resulting in an $O(\log^4 n)$ running time.

Similar variants of derandomization for MIS, maximal matching and $(\Delta+1)$-coloring were presented in [1, 35]. Berger and Rompel [9] developed a general framework for removing randomness from RNC algorithms when polylogarithmic independence is sufficient. The parallel setting bears some similarity to the all-to-all communication model but the barriers in these two models are different mainly because the complexity measure in the parallel setting is the computation time while in our setting local computation is for free. This raises the possibility of obtaining much better results in the congested clique model compared to what is known in the parallel setting.

Derandomization in the distributed setting. Naor and Stockmeyer [47] showed that constant-round randomized algorithms for problems that are locally checkable can be derandomized without an asymptotic overhead, extended by [14, 24] for larger time complexities and for a wider range of problems. Awerbuch et al. [2] claim to use the derandomized MIS algorithm of Luby [43] to obtain a deterministic CONGEST MIS algorithm. This claim is, however, not supported in their paper and is also late stated as open in [22].

2 Preliminaries and Notation

Our derandomization approach consists of first reducing the independence between the coin flips of the nodes. Then, we find some target function we wish to maintain during each

iteration of the derandomized algorithm. Finally, we find a pessimistic estimator for the target function and apply the method of conditional expectations to get a deterministic algorithm. Below we elaborate upon the above ingredients.

***d*-wise independent random variables.** In the algorithms we derandomize in the paper, a node $v \in V$ flips coins with probability p of being heads. As we show, it is enough to assume only d-wise independence between the coin flips of nodes. We show how to use a randomness *seed* of only $t = d\lceil \max\{\log n, \log 1/p\} \rceil$ bits to generate a coin flip for each $v \in V$, such that the coin flips are d-wise independent. We first need the notion of d-wise independent hash functions as presented in [56].

▶ **Definition 6** ([56, Definition 3.31])**.** For $N, M, d \in \mathbb{N}$ such that $d \leq N$, a family of functions $\mathcal{H} = \{h : [N] \to [M]\}$ is d-wise independent if for all distinct $x_1, x_2, ..., x_d \in [N]$, the random variables $H(x_1), ..., H(x_d)$ are independent and uniformly distributed in $[M]$ for a randomly chosen H in $\mathcal{H}$.

In [56] an explicit construction of $\mathcal{H}$ is presented, with parameters as stated next.

▶ **Lemma 7** ([56, Corollary 3.34])**.** *For every $\gamma, \beta, d \in \mathbb{N}$, there is a family of d-wise independent functions $\mathcal{H}_{\gamma,\beta} = \{h : \{0,1\}^\gamma \to \{0,1\}^\beta\}$ such that choosing a random function from $\mathcal{H}_{\gamma,\beta}$ takes $d \cdot \max\{\gamma, \beta\}$ random bits, and evaluating a function from $\mathcal{H}_{\gamma,\beta}$ takes time $poly(\gamma, \beta, d)$.*

Let us now consider some node $v \in V$ which needs to flip a coin with probability p that is d-wise independent with respect to the coin flips of other nodes. Using Lemma 7 with parameters $\gamma = \lceil \log n \rceil$ and $\beta = \lceil \log 1/p \rceil$, we can construct $\mathcal{H}$ such that every function $h \in \mathcal{H}$ maps the ID of a node to the result of its coin flip. Using only $t = d \cdot \max\{\gamma, \beta\}$ random bits we can flip d-wise independent biased coins with probability p for all nodes in v. We define Y to be a vector of t random coins. Note we can also look at Y as a vector of length $t/\log n$ where each entry takes values in $\{1, \ldots, \lceil \log n \rceil\}$. We use the latter when dealing with Y. From Y each node v can generate its random coin toss by accessing the corresponding $h \in \mathcal{H}$ and checking whether $h(ID(v)) = 0$. From Definition 6 it holds that $Pr[h(ID(v)) = 0] = p$, as needed.

The method of conditional expectations. Next, we consider the method of conditional expectations. Let $\phi : A^\ell \to \mathbb{R}$, and let $X = (X_1, ..., X_\ell)$ be a vector of random variables taking values in A. If $E[\phi(X)] \geq \alpha$ then there is an assignment of values $Z = (z_1, ..., z_\ell)$ such that $\phi(Z) \geq \alpha$. We describe how to *find* the vector Z. We first note that from the law of total expectation it holds that $E[\phi(X)] = \sum_{a \in A} E[\phi(X) \mid X_1 = a] Pr[X_1 = a]$, and therefore for at least some $a \in A$ it holds that $E[\phi(X) \mid X_1 = a] \geq \alpha$. We set this value to be z_1. We then repeat this process for the rest of the values in X, which results in the vector Z. In order for this method to work we need it to be possible to *compute* the conditional expectation of $\phi(X)$.

We now wish to use the method of conditional expectations after reducing the number of random bits used by the algorithm. Let us denote by $\bar{\rho}$ the original vector of random bits used by the algorithm. Taking Y as before to be the seed vector for $\bar{\rho}$, we have that $\bar{\rho}$ is a function of Y. We need to be able to compute $E[\phi(\bar{\rho}(Y)) \mid y[1] = a_1, \ldots, y[i] = a_i]$ for all possible values of i and $a_j, j \leq i$. Computing the conditional expectations for ϕ might be expensive. For this reason we use a *pessimistic estimator*. A pessimistic estimator of ϕ is a function $\phi' : A^\ell \to \mathbb{R}$ such that for all values of i and $a_j, j \leq i$ it holds that

$E[\phi(\bar{\rho}(Y)) \mid y_1 = b_1, \ldots, y_i = b_i] \geq E[\phi'(\bar{\rho}(Y)) \mid y_1 = b_1, \ldots, y_i = b_i]$. If ϕ' is a pessimistic estimator of ϕ whose expected value is still bounded from above by α, then we can use the method of conditional expectations on ϕ' and obtain $z_1, \ldots, z_n$, such that $\phi(z_1, \ldots, z_n) \geq \phi'(z_1, \ldots, z_n) \geq \alpha$.

Lenzen's routing algorithm. We make heavy use of the deterministic routing algorithm of Lenzen [38], which guarantees that if each node needs to send at most $O(n \log n)$ bits and receive at most $O(n \log n)$ bits then $O(1)$ rounds are sufficient.

3 Deterministic MIS

To prove Theorem 1, we consider the following modification of the randomized algorithm of Ghaffari [26]. The algorithm of Ghaffari consists of two parts. The first part (shown to have a good local complexity) consists of $O(\log \Delta)$ phases, each with $O(1)$ rounds. After this first part, it is shown that sufficiently many nodes are removed from the graph. The MIS for what remains is computed in the second part deterministically in time $2^{O(\sqrt{\log \log n})}$. We only use the first part of Ghaffari's algorithm, and the only change to it is a slight modification of the constants that are used.

We define a slight modification to the first part of Ghaffari's MIS Algorithm: Set $p_0(v) = 1/4$. Define $p_{t+1}(v) = 1/2 \cdot p_t(v)$, if $d_t(v) \geq 1/2$ and $p_{t+1}(v) = \min\{2p_t(v), 1/4\}$, otherwise. Here $d_t(v) = \sum_{u \in N(v)} p_t(u)$ is the *effective degree* of node v in phase t. In each phase t, the node v gets marked with probability $p_t(v)$ and if none of its neighbors is marked, v joins the MIS and gets removed along with its neighbors.

3.1 $O(\log \Delta)$ round randomized MIS algorithm in the congested clique

We begin by observing that in the congested clique, what remains after $O(\log \Delta)$ phases of Ghaffari's algorithm can be solved in $O(1)$ rounds. This provides an improved *randomized* runtime compared to [26], and specifically, has no dependence on n. The algorithm consists of two parts. In the first part, we run Ghaffari's algorithm for $O(\log \Delta)$ phases. We emphasize that this works with both Ghaffari's algorithm and with our modified Ghaffari's algorithm, since the values of the constants do not affect the asymptotic running time and correctness of the randomized first part of the algorithm. Then, in the second part, a leader collects all surviving edges and solves the remaining MIS deterministically on that subgraph. We show that the total number of edges incident to these nodes is $O(n)$ w.h.p., and hence using the deterministic routing algorithm of Lenzen [38], the second part can be completed in $O(1)$ rounds w.h.p. We note that the proof that $O(n)$ edges remain cannot be extended to the case of pairwise independence, which is needed for derandomization, since the concentration guarantees are rather weak. For this, we need to develop in the following section new machinery. The full proof Thm. 1 appears in the full version [12].

3.2 Derandomizing the modified MIS algorithm

3.2.1 Ghaffari's algorithm with pairwise independence

We review the main terminology and notation from [26], up to our modification of constants. Changing the constants is important as we are using pairwise independence and not complete independence as in the original algorithm of Ghaffari. A node v is called *light* if $d_t(v) < 1/4$. We define two types of *golden phases* for a node v. This is a modification of the corresponding definitions in [26].

Type-1 golden phase: $p_t(v) = 1/4$ and $d_t(v) \leq 1/2$;
Type-2 golden phase: $d_t(v) > 1/4$ and at least $d_t(v)/10$ of it arises from light nodes.

A node v is called *golden* in phase t, if phase t is a golden phase for v (of either type). Intuitively, a node v that is golden in phase t is shown to have a constant probability of being removed. Specifically, in a golden phase of type-1, v has a constant probability to join the MIS and in a golden phase of type-2, there is a constant probability that v has a neighbor that joins the MIS and hence v is removed.

We now prove the analogue of Lemma 3.3 in [26] for the setting in which the coin flips made by the nodes are not completely independent but are only *pairwise independent*. We show that a golden node for phase t is still removed with constant probability even under this weaker bounded independence guarantee. The proof of the following appears in the full version [12].

▶ **Lemma 8** (golden nodes with pairwise independence)**.** *Consider the modified Ghaffari's algorithm with pairwise independent coin flips.*

(1) *If t is a type-1 golden phase for a node v, then v joins the MIS in phase t with probability at least $1/8$.*

(2) *If t is a type-2 golden phase for a node v then v is removed in phase t with probability at least $\alpha = 1/160$.*

As a result, the following holds in the pairwise independence setting:

▶ **Lemma 9.** *Within $O(\log \Delta)$ phases, every node remains with probability at most $1/\Delta$.*

Recall that the proof from Subsection 3.1 that $O(n)$ edges remain cannot be extended to pairwise independence since the concentration guarantees are rather weak. Our algorithm will use pairwise independence but with some crucial modifications required in order to guarantee that after $O(\log \Delta)$ phases, only $O(n/\Delta)$ nodes remain undecided.

3.2.2 $O(\log n \log \Delta)$-round deterministic MIS in the congested clique

Using derandomization we show there is a deterministic MIS algorithm for the broadcast congested clique model that completes in $O(\log \Delta \log n)$ rounds, as stated in Theorem 2.

3.2.2.1 The challenge

Consider phase t in the modified Ghaffari's algorithm and let V_t be the set of golden nodes in this phase. Our goal is to select additional nodes into the MIS so that at least a constant fraction of the golden nodes are removed. Let $v_1, \ldots, v_{n'}$ be the nodes that are *not* removed in phase t. Towards derandomizing the algorithm, for each node, we define the corresponding random variables $x_1, \ldots, x_{n'}$ indicating whether v_i is marked in phase t. Let $X_i = (x_1 = b_1, \ldots, x_i = b_i)$ define a partial assignment for $v_1, \ldots, v_i$ (i.e., whether or not they are in the MIS in phase t). Let $X_0 = \emptyset$ denote the case where none of the decisions is fixed.

For a golden node v (in phase t), let $r_{v,t}$ be the random variable indicating whether v gets removed in phase t, and let R_t be the random variable of the number of removed golden nodes. By linearity of expectation, $\mathbb{E}(R_t) = \sum_v \mathbb{E}(r_{v,t})$ is the *expected* number of removed golden nodes in phase t. By Lemma 8, there is a constant c such that $\mathbb{E}(R_t) \geq c \cdot |V_t|$. Potentially, we could aim for the following: Given the partial assignment X_i, compute the two expectations of the number of removed golden nodes conditioned on the two possible assignments for x_{i+1}, $\mathbb{E}(R_t \mid X_i, x_{i+1} = 0)$ and $\mathbb{E}(R_t \mid X_i, x_{i+1} = 1)$, and choose x_{i+1} according to the larger expectation.

However, towards the above goal we face the following main challenges. (C1) The value of R_t cannot be easily computed, since when using probabilities of neighboring nodes we might be double-counting: a node might be removed while having more than a single neighbor that joins the MIS. (C2) The search space of size 2^n is too large and in particular, the conditional expectation computation consists of n steps. (C3) Even when using pairwise independence to enjoy an $O(\log n)$-bit seed, searching a good seed point in a space of size $O(\text{poly } n)$ in a brute force manner cannot be done efficiently in the congested clique. (C4) Despite our proof that golden nodes are removed with constant probability even with pairwise independence, it is still not clear how to implement the second part of the MIS algorithm, because showing that only $O(n/\Delta)$ nodes survive cannot be done with pairwise independence. That is, the proof from Subsection 3.1 that $O(n)$ edges remain inherently needs full independence.

Addressing (C4) requires a priority-based scheme for choosing the nodes that join the MIS, which requires a novel age-based weighting approach to be added to the MIS algorithm. Next, we describe our main derandomization tools and then provide our algorithm.

3.2.2.2 Derandomization tools

We define a pessimistic estimator to the conditional expectation $\mathbb{E}(R_t \mid X_i)$, which can be computed efficiently in our model. Then, we describe how to reduce the search space using pairwise independence. In our algorithm, the nodes will apply the method of conditional expectations on the estimator in order to find a "good" seed of length $O(\log n)$.

Tool 1: The pessimistic estimator function. Consider phase t and recall that V_t are the golden nodes in this phase. Similarly to the clever approach of [42, 43], we define a variable $\psi_{v,t}$ that will satisfy that $r_{v,t} \geq \psi_{v,t}$. The idea is to account for a removed node of type-2 only if it is removed because a *single* one of its neighbors joins the MIS. Since this can only occur for one of its neighbors, we avoid double-counting when computing the probabilities. This allows coping with challenge (C1).

Let $m_{v,t}$ be the random variable indicating the event that v is *marked*. Let $m_{v,u,t}$ indicate the event that both u and v are marked. Define $\psi_{v,t} = m_{v,t} - \sum_{u \in N(v)} m_{v,u,t}$ if v is of type-1, and $\psi_{v,t} = \sum_{u \in N(v)}(m_{u,t} - \sum_{w \in N(u)} m_{u,w,t} - \sum_{w' \in N(v) \setminus \{u\}} m_{u,w',t})$, if v is of type-2. Denoting $\Psi_t = \sum_{v \in V_t} \psi_{v,t}$ gives that Ψ_t is a lower bound on the number of removed golden nodes, i.e., $\Psi_t \leq R_t$. For a partial assignment $X_i = (x_1 = b_1, \ldots, x_i = b_i)$ indicating which of the nodes are in the MIS, we have[2]

$$\mathbb{E}(\psi_{v,t} \mid X_i) = \begin{cases} \Pr[m_{v,t} = 1 \mid X_i] - \sum_{u \in N(v)} \Pr[m_{v,u,t} \mid X_i], & \text{if } v \text{ is of type-1.} \\ \sum_{u \in N(v)}(\Pr[m_{u,t} = 1 \mid X_i] - \sum_{w \in N(u)} \Pr[m_{u,w,t} = 1 \mid X_i] - \\ \sum_{w' \in W(v) \setminus \{u\}} \Pr[m_{u,w',t} = 1 \mid X_i)], & \text{if } v \text{ is of type-2,} \end{cases} \tag{1}$$

where $W(v) \subseteq N(v)$ is a subset of v's neighbors satisfying that $\sum_{w \in W(v)} p_t(w) \in [1/40, 1/4]$ (as used in the proof of Lemma 8). By Lemma 8, it holds that $\mathbb{E}(\psi_{v,t}) \geq \alpha$ for $v \in V_t$. Hence, we have that: $\mathbb{E}(r_{v,t}) \geq \mathbb{E}(\psi_{v,t}) \geq \alpha$. Since $r_{v,t} \geq \psi_{v,t}$ even upon conditioning on the partial assignment X_i, we get: $\mathbb{E}(R_{v,t} \mid X_i) \geq \mathbb{E}(\Psi_t \mid X_i) = \sum_{v \in V_t} \mathbb{E}(\psi_{v,t} \mid X_i) \geq \alpha \cdot |V_t|$. Our algorithm will employ the method of conditional expectations on a *weighted* version of $\mathbb{E}(\Psi_t \mid X_i)$, as will be discussed later.

[2] For ease of presentation, here we condition on a n-length vector. However, our algorithm will use the pairwise independence – discussed in the following paragraph – to condition on a seed of length $O(\log n)$.

Tool 2: Pairwise independence. We now combine the method of conditional expectations with a small search space. We use Lemma 7 with $d = 2$, $\gamma = \Theta(\log n)$ and a prime number $\beta = O(\log \Delta)$. This is because we need the marking probability, $p_t(v)$, to be $\Omega(1/\text{poly } \Delta)$.

Consider phase t. Using the explicit construction of Lemma 7, if all nodes are given a shared random seed of length γ, they can sample a random hash function $h : \{0,1\}^\gamma \to \{0,1\}^\beta$ from $\mathcal{H}_{\gamma,\beta}$ which yields n pairwise independent choices. Specifically, flipping a biased coin with probability of $p_t(v)$ can be trivially simulated using the hash value $h(ID_v)$ where ID_v is an $O(\log n)$-bit ID of v.[3] Since h is a random function in the family, all random choices are pairwise independent and the analysis of of the golden phases goes through. This standard approach takes care of challenge (C2).

Even though using a seed of length $O(\log n)$ reduces the search space to be of polynomial size, still, exploring all possible $2^{O(\log n)} = O(n^c)$ seeds in a brute force manner is too time consuming. Instead, we employ the method of conditional expectations to find a *good seed*. That is, we will consider $\mathbb{E}(\Psi_t \mid Y_i)$ where $Y_i = (y_1 = b_1, \ldots, y_i = b_i)$ is a partial assignment to the *seed* $Y = (y_1, \ldots, y_a)$. The crux here is that since a *random* seed is good, then so is the expectation over seeds that are sampled uniformly at random. Hence, the method of conditional expectations will find a seed that is at least as good as the random selection. Specifically, we still use the pessimistic estimator of Equation (1), but we condition on the small seed Y_i rather than on X_i. This addresses challenge (C3).

Tool 3: An age-based weighted adaptation. To handle challenge (C4), we compute the expectation of a weighted version of Ψ_t, which favors *old* nodes where the age of a node is counted as the number of golden phases it experienced. Let $age(v)$ be the number of golden phases v has till phase t and recall that a golden node is removed with probability at least α. Define $\psi'_{v,t} = (1/(1-\alpha))^{age(v)} \cdot \psi_{v,t}$, and $\Psi'_t = \sum_{v \in V_t} \psi'_{v,t}$. We use the method of conditional expectations for:

$$\mathbb{E}(\Psi'_t \mid Y_i) = \sum_{v \in V_t} \mathbb{E}(\psi'_{v,t} \mid Y_i) \ , \tag{2}$$

rather than for $\mathbb{E}(\Psi_t \mid Y_i)$. The choice of this function will be made clear in the proof.

3.2.2.3 Algorithm Description

The first part of the algorithm consists of $\Theta(\log \Delta)$ phases, where in phase t, we derandomize phase t in the modified Ghaffari's algorithm using $O(\log n)$ deterministic rounds. In the second part, all nodes that remain undecided after the first part, send their edges to the leader using the deterministic routing algorithm of Lenzen. The leader then solves locally and notifies the relevant nodes to join the MIS. In the analysis section, we show that after the first part, only $O(n/\Delta)$ nodes remain undecided, and hence the second part can be implemented in $O(1)$ rounds.

From now on we focus on the first part. Consider phase t in the modified Ghaffari's algorithm. Note that at phase t, some of the nodes are already removed from the graph (either because they are part of the MIS or because they have a neighbor in the MIS). Hence, when we refer to nodes or neighboring nodes, we refer to the remaining graph induced on the undecided nodes.

[3] Flipping a biased coin with probability $1/2^i$, is the same as getting a uniformly distributed number y in $[1, b]$ and outputting 1 if and only if $y \in [1, 2^{b-i}]$.

Let $Y = (y_1, \ldots, y_\gamma)$ be the γ random variables that are used to select a hash function and hence induce a deterministic algorithm. We now describe how to compute the value of y_i in the seed, given that we already computed $y_1 = b_1, \ldots, y_{i-1} = b_{i-1}$. By exchanging IDs (of $\Theta(\log n)$ bits), as well as the values $p_t(v)$ and $d_t(v)$ with its neighbors, a node can check if it is a golden type-1 or type-2 node. In addition, every node maintains a counter, $age(v)$ referred to as the age of v, which measures the number of golden phases it had so far.

Depending on whether the node v is a golden type-1 or type-2 node, based on Equation (1), it computes a lower bound on the conditional probability that it is removed given the partial seed assignment $Y_{i,b} = (y_1, \ldots, y_i = b)$ for every $b \in \{0, 1\}$. These lower bound values are computed according to the proofs of Lemma 8. Specifically, a golden node v of type-1, uses the IDs of its neighbors and their $p_t(u)$ values to compute the following: $\mathbb{E}(\psi_{v,t} \mid Y_{i,b}) = \Pr[m_{v,t} = 1 \mid Y_{i,b}] - \sum_{u \in N(v)} \Pr[m_{u,t} = 1 \mid Y_{i,b}]$, where $\Pr[m_{v,t} = 1 \mid Y_{i,b}]$ is the conditional probability that v is marked in phase t (see full version [12] for full details about this computation). For a golden node v of type-2 the lower bound is computed differently. First, v defines a subset of neighbors $W(v) \subseteq N(v)$, satisfying that $\sum_{w \in W(v)} p_t(w) \in [1/40, 1/4]$, as in the proof of Lemma 8. Let $M_{t,b}(u)$ be the conditional probability on $Y_{i,b}$ that u is marked but none of its neighbors are marked. Let $M_{t,b}(u, W(v))$ be the conditional probability on $Y_{i,b}$ that another node other than u is marked in $W(v)$.[4] By exchanging the values $M_{t,b}(u)$, v computes: $\mathbb{E}(\psi_{v,t} \mid Y_{i,b}) = \sum_{u \in W(v)} \Pr[m_{u,t} = 1 \mid Y_{i,b}] - M_{t,b}(u) - M_{t,b}(u, W(v))$.

Finally, as in Equation (2), the node sends to the leader the values $\mathbb{E}(\psi'_{v,t} \mid Y_{i,b}) = 1/(1-\alpha)^{age(v)} \cdot \mathbb{E}(\psi_{v,t} \mid Y_{i,b})$ for $b \in \{0, 1\}$. The leader computes the sum of the $\mathbb{E}(\psi'_{v,t} \mid Y_{i,b})$ values of all golden nodes V_t, and declares that $y_i = 0$ if $\sum_{v \in V_t} \mathbb{E}(\psi'_{v,t} \mid Y_{i,b}) \geq \sum_{v \in V_t} \mathbb{E}(\psi'_{v,t} \mid Y_{i,b})$, and $y_i = 1$ otherwise. This completes the description of computing the seed Y.

Once the nodes compute Y, they can simulate phase t of the modified Ghaffari's algorithm. In particular, the seed Y defines a hash function $h \in \mathcal{H}_{\gamma,\beta}$ and $h(ID(v))$ can be used to simulate the random choice with probability $p_t(v)$. The nodes that got marked send a notification to neighbors and if none of their neighbors got marked as well, they join the MIS and notify their neighbors. Nodes that receive join notification from their neighbors are removed from the graph. This completes the description of the first part of the algorithm. A pseudocode appears in the full version [12].

Analysis. The correctness proof of the algorithm uses a different argument than that of Ghaffari [26]. Our proof does not involve claiming that a constant fraction of the golden nodes are removed, because in order to be left with $O(n/\Delta)$ undecided nodes we have to favor removal of old nodes. The entire correctness is based upon Lemma 15 in the full paper attached, which justifies the definition of the expectation given in Equation (2). The remaining $O(n)$ edges incident to the undecided nodes can be collected at the leader in $O(1)$ rounds using the deterministic routing algorithm of Lenzen [38]. The leader then solves MIS for the remaining graph locally and informs the nodes. This completes the correctness of the algorithm. Theorem 2 follows.

[4] The term $M_{t,b}(u, W(v))$ is important as it is what prevents double counting, because the corresponding random variables defined by the neighbors of v are mutually exclusive.

3.3 An $O(D \log^2 n)$ deterministic MIS algorithm for CONGEST

Here we provide a fast deterministic MIS algorithm for the harsher CONGEST model. For comparison, in terms of n alone, the best deterministic MIS algorithm is by Panconesi and Srinivasan [48] from more than 20 years ago and is bounded by $2^{O(\sqrt{\log n})}$ rounds. However, the algorithm requires large messages and hence is suitable for the LOCAL model but not for CONGEST. The only known non-trivial deterministic solution for CONGEST is to use any $(\Delta+1)$-coloring algorithm running in $O(\Delta+\log^* n)$ rounds (for example [3, 6]) to obtain the same complexity for deterministic MIS in CONGEST (notice that there are faster coloring algorithms, but the reduction has to pay for the number of colors anyhow). The following is our main result for CONGEST.

▶ **Theorem 4** (restated). *There is a deterministic MIS algorithm for the CONGEST model that completes in* $\min\{O(D \log^2 n), O(2^{\sqrt{\log n \log\log n}})\}$ *rounds.*

Proof. The bound of $2^{O(\sqrt{\log n \log\log n})}$ follows by the network decomposition of [5]. To get a bound of $O(D \log^2 n)$ rounds, we use a similar algorithm to Theorem 2 with two main differences. First, we run Ghaffari's algorithm for $O(\log n)$ rounds instead of $O(\log \Delta)$ rounds. Each round is simulated by a phase with $O(D \log n)$ rounds. Specifically, in each phase, we need to compute the seed of length $O(\log n)$, this is done bit by bit using the method of conditional expectations exactly as described eariler and aggregating the result at some leader node (aggregation is done in the standard way). The leader then notifies the assignment of the bit to the entire graph. Since each bit in the seed is computed in $O(D)$ rounds, overall the run time is $O(D \log^2 n)$. For the correctness, we assume towards contradiction that after $\Omega(\log n)$ rounds, at least one node remains undecided. Then, we show that every node that survives can charge $\Omega(n^c)$ nodes that are removed, which is a contradiction as there only n nodes. ◀

4 Discussion

We have shown how to derandomize an MIS algorithm and a spanner construction in the congested clique model, and derandomize an MIS algorithm in the CONGEST model. This greatly improves upon the previously known results. Whereas our techniques imply that many local algorithms can be derandomized in the congested-clique (e.g., hitting set, ruling sets, coloring, matching etc.), the situation appears to be fundamentally different for global tasks such as connectivity, min-cut and MST. For instance, the best randomized MST algorithm in the congested-clique has time complexity of $O(\log^* n)$ rounds [27], but the best deterministic bound is $O(\log \log n)$ rounds [41]. Derandomization of such global tasks might require different techniques.

The importance of randomness in *local* computation lies in the fact that recent developments [14] show separations between randomized and deterministic complexities in the unlimited bandwidth setting of the LOCAL model. While some distributed algorithms happen to use small messages, our understanding of the impact of message size on the complexity of local problems is in its infancy.

This work opens a window to many additional intriguing questions. First, we would like to see many more local problems being derandomized despite congestion restrictions. Alternatively, significant progress would be made by otherwise devising deterministic algorithms for this setting. Finally, understanding the relative power of randomization with bandwidth restrictions is a worthy aim for future research.

Acknowledgments. We are very grateful to Mohsen Ghaffari for many helpful discussions and useful observations involving the derandomization of his MIS algorithm.

References

1 Noga Alon, László Babai, and Alon Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *Journal of algorithms*, 7(4):567–583, 1986.

2 Baruch Awerbuch, Andrew V. Goldberg, Michael Luby, and Serge A. Plotkin. Network decomposition and locality in distributed computation. In *FOCS*, pages 364–369, 1989.

3 Leonid Barenboim. Deterministic $(\Delta + 1)$-coloring in sublinear (in Δ) time in static, dynamic, and faulty networks. *J. ACM*, 63(5):47:1–47:22, 2016.

4 Leonid Barenboim and Michael Elkin. Distributed graph coloring: Fundamentals and recent developments. *Synthesis Lectures on Distributed Computing Theory*, 4(1):1–171, 2013.

5 Leonid Barenboim, Michael Elkin, and Cyril Gavoille. A fast network-decomposition algorithm and its applications to constant-time distributed computation. In *SIROCCO*, pages 209–223, 2015.

6 Leonid Barenboim, Michael Elkin, and Fabian Kuhn. Distributed (delta+1)-coloring in linear (in delta) time. *SIAM J. Comput.*, 43(1):72–95, 2014.

7 Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The locality of distributed symmetry breaking. *J. ACM*, 63(3):20:1–20:45, 2016.

8 Surender Baswana and Sandeep Sen. A simple and linear time randomized algorithm for computing sparse spanners in weighted graphs. *Random Structures & Algorithms*, 30(4):532–563, 2007.

9 Bonnie Berger and John Rompel. Simulating (log cn)-wise independence in nc. *Journal of the ACM (JACM)*, 38(4):1026–1046, 1991.

10 Sebastian Brandt, Orr Fischer, Juho Hirvonen, Barbara Keller, Tuomo Lempiäinen, Joel Rybicki, Jukka Suomela, and Jara Uitto. A lower bound for the distributed lovász local lemma. In *STOC*, pages 479–488, 2016.

11 Keren Censor-Hillel, Petteri Kaski, Janne H. Korhonen, Christoph Lenzen, Ami Paz, and Jukka Suomela. Algebraic methods in the congested clique. In *PODC*, pages 143–152, 2015.

12 Keren Censor-Hillel, Merav Parter, and Gregory Schwartzman. Derandomizing local distributed algorithms under bandwidth restrictions. *arXiv preprint arXiv:1608.01689*, 2016. URL: `https://arxiv.org/abs/1608.01689`.

13 Karthekeyan Chandrasekaran, Navin Goyal, and Bernhard Haeupler. Deterministic algorithms for the lovász local lemma. *SIAM Journal on Computing*, 42(6):2132–2155, 2013.

14 Yi-Jun Chang, Tsvi Kopelowitz, and Seth Pettie. An exponential separation between randomized and deterministic complexity in the LOCAL model. In *FOCS*, pages 615–624, 2016.

15 Bilel Derbel and Cyril Gavoille. Fast deterministic distributed algorithms for sparse spanners. *Theor. Comput. Sci.*, 399(1-2):83–100, 2008.

16 Bilel Derbel, Cyril Gavoille, and David Peleg. Deterministic distributed construction of linear stretch spanners in polylogarithmic time. In *DISC*, pages 179–192, 2007.

17 Bilel Derbel, Cyril Gavoille, David Peleg, and Laurent Viennot. On the locality of distributed sparse spanner construction. In *PODC*, pages 273–282, 2008.

18 Bilel Derbel, Cyril Gavoille, David Peleg, and Laurent Viennot. Local computation of nearly additive spanners. In *DISC*, pages 176–190, 2009.

19 Bilel Derbel, Mohamed Mosbah, and Akka Zemmari. Sublinear fully distributed partition with applications. *Theory of Computing Systems*, 47(2):368–404, 2010.

20 Danny Dolev, Christoph Lenzen, and Shir Peled. "tri, tri again": Finding triangles and small subgraphs in a distributed setting - (extended abstract). In *DISC*, pages 195–209, 2012.

21 Andrew Drucker, Fabian Kuhn, and Rotem Oshman. On the power of the congested clique model. In *PODC*, pages 367–376, 2014.

22 Michael Elkin. A faster distributed protocol for constructing a minimum spanning tree. *J. Comput. Syst. Sci.*, 72(8):1282–1308, 2006.

23 Michael Elkin. A near-optimal distributed fully dynamic algorithm for maintaining sparse spanners. In *PODC*, pages 185–194, 2007.

24 Laurent Feuilloley and Pierre Fraigniaud. Randomized local network computing. In *SPAA*, pages 340–349, 2015.

25 François Le Gall. Further algebraic algorithms in the congested clique model and applications to graph-theoretic problems. In *DISC*, pages 57–70, 2016.

26 Mohsen Ghaffari. An improved distributed algorithm for maximal independent set. In *SODA*, pages 270–277, 2016.

27 Mohsen Ghaffari and Merav Parter. Mst in log-star rounds of congested clique. In *PODC*, pages 19–28, 2016.

28 Mark Goldberg and Thomas Spencer. A new parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 18(2):419–427, 1989.

29 Yijie Han. A fast derandomization scheme and its applications. *SIAM Journal on Computing*, 25(1):52–82, 1996.

30 James W. Hegeman, Gopal Pandurangan, Sriram V. Pemmaraju, Vivek B. Sardeshmukh, and Michele Scquizzato. Toward optimal bounds in the congested clique: Graph connectivity and MST. In *PODC*, pages 91–100, 2015.

31 James W. Hegeman and Sriram V. Pemmaraju. Lessons from the congested clique applied to mapreduce. In *SIROCCO*, pages 149–164, 2014.

32 James W. Hegeman, Sriram V. Pemmaraju, and Vivek Sardeshmukh. Near-constant-time distributed algorithms on a congested clique. In *DISC*, pages 514–530, 2014.

33 Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. A deterministic almost-tight distributed algorithm for approximating single-source shortest paths. In *STOC*, pages 489–498, 2016.

34 Stephan Holzer and Nathan Pinsker. Approximation of distances and shortest paths in the broadcast congest clique. In *OPODIS*, pages 6:1–6:16, 2015.

35 Amos Israeli and Alon Itai. A fast and simple randomized parallel algorithm for maximal matching. *Inf. Process. Lett.*, 22(2):77–80, 1986.

36 Richard M Karp and Avi Wigderson. A fast parallel algorithm for the maximal independent set problem. In *STOC*, pages 266–272, 1984.

37 Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. Local computation: Lower and upper bounds. *J. ACM*, 63(2):17, 2016.

38 Christoph Lenzen. Optimal deterministic routing and sorting on the congested clique. In *PODC*, pages 42–50, 2013.

39 Christoph Lenzen and Roger Wattenhofer. Tight bounds for parallel randomized load balancing: extended abstract. In *STOC*, pages 11–20, 2011.

40 Nathan Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992.

41 Zvi Lotker, Elan Pavlov, Boaz Patt-Shamir, and David Peleg. MST construction in $O(\log \log n)$ communication rounds. In *SPAA*, pages 94–100, 2003.

42 Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM journal on computing*, 15(4):1036–1053, 1986.

43 Michael Luby. Removing randomness in parallel computation without a processor penalty. *Journal of Computer and System Sciences*, 47(2):250–286, 1993.

44 Michael Mitzenmacher and Eli Upfal. *Probability and computing - randomized algorithms and probabilistic analysis*. Cambridge University Press, 2005.

45 Rajeev Motwani, Joseph Naor, and Moni Naor. The probabilistic method yields deterministic parallel algorithms. *J. Comput. Syst. Sci.*, 49(3):478–516, 1994.

46 Danupon Nanongkai. Distributed approximation algorithms for weighted shortest paths. In *STOC*, pages 565–573, 2014.

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

48 Alessandro Panconesi and Aravind Srinivasan. Improved distributed algorithms for coloring and network decomposition problems. In *STOC*, pages 581–592, 1992.

49 Alessandro Panconesi and Aravind Srinivasan. On the complexity of distributed network decomposition. *J. Algorithms*, 20(2):356–374, 1996.

50 Grammati Pantziou, Paul Spirakis, and Christos Zaroliagis. Fast parallel approximations of the maximum weighted cut problem through derandomization. In *FSTTCS*, pages 20–29, 1989.

51 Boaz Patt-Shamir and Marat Teplitsky. The round complexity of distributed sorting: extended abstract. In *PODC*, pages 249–256, 2011.

52 Seth Pettie. Distributed algorithms for ultrasparse spanners and linear size skeletons. *Distributed Computing*, 22(3):147–166, 2010.

53 Liam Roditty, Mikkel Thorup, and Uri Zwick. Deterministic constructions of approximate distance oracles and spanners. In *ICALP*, pages 261–272, 2005.

54 Anand Srivastav and Lasse Kliemann. Parallel algorithms via the probabilistic method. In *Handbook of Parallel Computing - Models, Algorithms and Applications.* Chapman and Hall/CRC, 2007.

55 Jukka Suomela. Survey of local algorithms. *ACM Comput. Surv.*, 45(2):24, 2013.

56 Salil P. Vadhan. Pseudorandomness. *Foundations and Trends in Theoretical Computer Science*, 7(1-3):1–336, 2012.

On the Number of Objects with Distinct Power and the Linearizability of Set Agreement Objects

David Yu Cheng Chan[1], Vassos Hadzilacos[2], and Sam Toueg[3]

1 **Department of Computer Science, University of Toronto, Ontario, Canada**
davidchan@cs.toronto.edu
2 **Department of Computer Science, University of Toronto, Ontario, Canada**
vassos@cs.toronto.edu
3 **Department of Computer Science, University of Toronto, Ontario, Canada**
sam@cs.toronto.edu

Abstract

We first prove that there are uncountably many objects with distinct computational powers. More precisely, we show that there is an uncountable set of objects such that for any two of them, at least one cannot be implemented from the other (and registers) in a wait-free manner. We then strengthen this result by showing that there are uncountably many *linearizable* objects with distinct computational powers. To do so, we prove that for all positive integers n and k, there is a linearizable object that is computationally equivalent to the k-set agreement task among n processes. To the best of our knowledge, these are the first linearizable objects proven to be computationally equivalent to set agreement tasks.

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases Set Agreement, Asynchronous System, Shared Memory

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.12

1 Introduction

One of the fundamental problems in distributed computing is to determine whether two shared objects are *equivalent*, i.e., whether each can be implemented from the other (and registers) in a wait-free manner.[1] Any two objects that are not equivalent do not have the same computational power, since at least one cannot implement the other.

To address this fundamental problem, Herlihy proposed the following object classification scheme: an object O is in level n of a hierarchy if, together with registers, O can be used to solve consensus among at most n processes [12]. It is clear that objects in different levels of this hierarchy are not equivalent. Unfortunately, the converse is not true: every level $n \in \mathbb{Z}^+$ of this hierarchy contains objects that are not equivalent [2, 7, 11, 16].[2] So Herlihy's hierarchy does not classify objects in a "precise" way. This motivates the search for a *precise* object classification scheme, i.e., one that partitions the universe $\mathcal{U}$ of all shared objects such that the following property holds: two objects are equivalent *if and only if* they are in the same cell of the partition.

In this paper, we first prove that there is an uncountable number of objects that are not equivalent to one another (and so they have distinct computational power). Thus any precise

[1] Throughout this paper, we consider only objects, tasks, and implementations that are *wait free*, so we subsequently omit all references to wait freedom.
[2] We denote by $\mathbb{Z}^+$ the set of all positive integers.


© David Yu Cheng Chan, Vassos Hadzilacos, and Sam Toueg;
licensed under Creative Commons License CC-BY

classification scheme of the universe $\mathcal{U}$ of objects contains an *uncountable* number of cells. So the cells of a precise classification scheme cannot be labeled using simple integers (as in Herlihy's hierarchy) or finite sequences of integers.

Our proof of the above result uses the (n, k)-*set agreement task* where each of n processes has a proposal value and must decide on one of the proposed values such that there are at most k distinct decision values [8]. The *set agreement power* of an object O is the infinite sequence $\vec{n} = (n_1, n_2, \ldots, n_k, \ldots)$ where, for all $k \geq 1$, n_k is the largest integer such that instances of O and registers can solve the (n_k, k)-set agreement task, or ∞ if instances of O and registers can solve the (n, k)-set agreement task for every integer n [10].

Let $\mathcal{S}$ denote the set of all infinite sequences of positive integers $\vec{n} = (n_1, n_2, \ldots, n_k, \ldots)$ such that $n_{k+1} \geq 2n_k$ for all $k \geq 1$. We use a result in [10] to prove that:

For all $\vec{n} \in \mathcal{S}$, there is an object $R_{\vec{n}}$ with set agreement power $\vec{n}$. (1)

Note that this is not obvious because it is *not* the case that *every* infinite sequence of positive integers $\vec{n} = (n_1, n_2, \ldots, n_k, \ldots)$ has a corresponding object $R_{\vec{n}}$ with set agreement power $\vec{n}$. Finally, we use a standard diagonalization argument to prove that the set $\mathcal{S}$ is uncountable. Therefore, by (1), there is an uncountable number of objects, namely the $R_{\vec{n}}$ objects, that have distinct set agreement power. Since objects with different set agreement power are not equivalent, we conclude that there are uncountably many objects that are not equivalent (and so they have distinct computational power).

Next, we prove that the above result holds even if we restrict the universe $\mathcal{U}$ of objects to contain only objects that are *linearizable* [13]. This result would be immediate if the $R_{\vec{n}}$ objects used in our proof were such objects. Our $R_{\vec{n}}$ objects, however, are *not* linearizable: this is because they are constructed using (n, k)-set agreement objects which, as described in [5, 9], are simply "*black-boxes*" that solve the (n, k)-set agreement task. In fact, to the best of our knowledge, all the (n, k)-set agreement objects used in the literature to date have *not* been defined as linearizable objects.

To show that there are uncountably many *linearizable* objects with distinct computational power, we proceed as follows:

1. We first prove that for all positive integers n and k, there is a linearizable object, denoted $LSA(n, k)$, that is computationally equivalent to the (n, k)-set agreement task in the following sense: the (n, k)-set agreement task can be solved using the $LSA(n, k)$ object, and the $LSA(n, k)$ object can be implemented using any solution to the (n, k)-set agreement task (and registers). This also implies that the linearizable $LSA(n, k)$ object is equivalent to the (n, k)-set agreement "black-box" object.
2. We then construct linearizable objects, denoted $LR_{\vec{n}}$, that are equivalent to the $R_{\vec{n}}$ objects. Roughly speaking we do so by replacing the (n, k)-set agreement "black-box" objects used to construct $R_{\vec{n}}$ with our linearizable $LSA(n, k)$ objects. Since there is an uncountable number of $R_{\vec{n}}$ objects with distinct computational power, and $R_{\vec{n}}$ is equivalent to $LR_{\vec{n}}$, there is also an uncountable number of linearizable $LR_{\vec{n}}$ objects with distinct computational power.

Proving that there is a linearizable object $LSA(n, k)$ that is *computationally* equivalent to the (n, k)-set agreement task is not obvious because the two are not *behaviourally* equivalent. Indeed, any linearizable object for the (n, k)-set agreement task imposes restrictions that are not inherent to this task [15, 6]. To see this, suppose that all the proposal values are distinct, and two processes propose concurrently. With the (n, k)-set agreement task, each of these two processes could decide the proposal value of the other. But a linearizable (n, k)-set agreement object does not allow this behaviour: whichever of the two processes is linearized

first cannot decide the proposal value of the other process. Since the set of behaviours allowed by linearizable (n, k)-set agreement objects is a proper subset of those allowed by the corresponding task, it is conceivable that such objects inherently have *greater computational power* than the task. Our result about the $LSA(n, k)$ objects shows that this is not the case.

In summary, the main contributions of this paper are to show that:

1. For all $n, k \in \mathbb{Z}^+$, there is a linearizable object that is computationally equivalent to the (n, k)-set agreement task.
2. The number of linearizable objects with distinct computational power is uncountable.

2 Some Basic Definitions

Object equivalence. Given any pair of shared memory objects O and O', we denote by $O \succeq O'$ the relation: there exists an implementation of O' from instances of O and registers. We say that O and O' are *equivalent*, denoted $O \equiv O'$, if and only if $O \succeq O'$ and $O' \succeq O$. Furthermore, given any object O and any *collection* of objects $\mathcal{C}$, (i) we denote by $\mathcal{C} \succeq O$ the relation: there exists an implementation of O from instances of objects in $\mathcal{C}$ and registers, and (ii) we denote by $O \succeq \mathcal{C}$ the relation: there exists an implementation of each object in $\mathcal{C}$ from instances of O and registers. We say O and $\mathcal{C}$ are *equivalent*, denoted $O \equiv \mathcal{C}$, if and only if $O \succeq \mathcal{C}$ and $\mathcal{C} \succeq O$.

Redirection objects. Let $\mathbb{Z}^*$ denote the set of all positive integers and the value ∞, i.e., $\mathbb{Z}^* = \mathbb{Z}^+ \cup \{\infty\}$. For all $n, k \in \mathbb{Z}^*$, we denote by $SA(n, k)$ the "black-box" (n, k)-*set agreement object* described in [5, 9]. Then, given any infinite sequence $\vec{n} = (n_1, n_2, \ldots, n_k, \ldots)$ such that $n_k \in \mathbb{Z}^*$ for all $k \in \mathbb{Z}^+$, we define a "redirection" object $R_{\vec{n}}$ that is equivalent to the collection $\mathcal{SA}_{\vec{n}}$ of set agreement objects $\bigcup_{k=1}^{\infty}\{SA(n_k, k)\}$. The object $R_{\vec{n}}$ supports the operation PROPOSE(v, k), for any value v and any integer $k \in \mathbb{Z}^+$. Intuitively, when an operation PROPOSE(v, k) is applied to the $R_{\vec{n}}$ object, it is redirected and applied as a PROPOSE(v) operation on the (n_k, k)-set agreement object $SA(n_k, k)$ in $\mathcal{SA}_{\vec{n}}$ and the response is returned. More precisely, for each $k \in \mathbb{Z}^+$, consider the set of all PROPOSE$(-, k)$ operations that are applied to the object $R_{\vec{n}}$. The values returned to these operations satisfy the properties of the $SA(n_k, k)$ object: If there are at most n_k such operations, then (a) k-*agreement*: at most k distinct values are returned to these operations, and (b) *validity*: each value returned to these operations was proposed by one of them.

▶ **Observation 1.** *For all infinite sequences $\vec{n} = (n_1, n_2, \ldots, n_k, \ldots)$ such that $n_k \in \mathbb{Z}^*$ for all $k \in \mathbb{Z}^+$, we have:*

(a) *For all $k \in \mathbb{Z}^+$, $R_{\vec{n}} \succeq SA(n_k, k)$; so $R_{\vec{n}} \succeq \mathcal{SA}_{\vec{n}}$.*

(b) *$\mathcal{SA}_{\vec{n}} \succeq R_{\vec{n}}$.*

(c) *Thus $R_{\vec{n}} \equiv \mathcal{SA}_{\vec{n}}$.*

3 Uncountability of Objects with Distinct Power

Let b be a non-negative integer, and $(a_1, a_2, \ldots)$ be non-negative integers such that only finitely many of the a_ℓ's are non-zero. Suppose that for each $\ell \in \mathbb{Z}^+$, we are given a_ℓ copies of $SA(n_\ell, \ell)$ objects. Using these objects, we can solve k-set agreement among n processes where $n \leq b + \sum_{\ell=1}^{\infty} a_\ell n_\ell$, and $k \geq b + \sum_{\ell=1}^{\infty} a_\ell \ell$. To do so, we partition the n processes as follows: for every a_ℓ that is non-zero, we create a_ℓ groups of at most n_ℓ processes each, and we also create one group of at most b processes. In each of the a_ℓ groups of at most n_ℓ processes, every process uses an $SA(n_\ell, \ell)$ object to propose its value and returns the object's

response; in the last group of at most b processes, each process returns its proposal value. In this way, the n processes return at most $b + \sum_{\ell=1}^{\infty} a_\ell \ell$ distinct values, each proposed by some process. By extending a result of Chaudhuri and Reiners [9], Delporte *et al.* proved that the ability to partition the n processes in such a manner is also a *necessary* condition to solve k-set agreement among n processes [10]. The following theorem follows from Theorem 2 of [10]:

▶ **Theorem 2** (Extended Set Agreement Partial Order Theorem). *Let $m \in \mathbb{Z}^*$ and $k \in \mathbb{Z}^+$, and let $\vec{n} = (n_1, n_2, \ldots, n_\ell, \ldots)$ be an infinite sequence such that $n_\ell \in \mathbb{Z}^*$ for all $\ell \in \mathbb{Z}^+$. Then $\mathcal{SA}_{\vec{n}} \succeq SA(m,k)$ if and only if there exists an infinite sequence $(a_1, a_2, \ldots)$ of non-negative integers and an integer $b \in \mathbb{N}$ such that:*[3]

- $b + \sum_{\ell=1}^{\infty} a_\ell n_\ell \geq m$.
- $b + \sum_{\ell=1}^{\infty} a_\ell \ell \leq k$.

Recall that $\mathcal{S}$ is the set of all infinite sequences of positive integers $\vec{n} = (n_1, n_2, \ldots, n_k, \ldots)$ such that $n_{k+1} \geq 2n_k$ for all $k \in \mathbb{Z}^+$. We now prove some properties of $\mathcal{S}$ that will be useful for applying Theorem 2.

▶ **Lemma 3.** *For all $\vec{n} \in \mathcal{S}$ and all $k \in \mathbb{Z}^+$, $\frac{n_{k+1}}{k+1} \geq \frac{n_k}{k}$.*

Proof. By definition, for all $\vec{n} \in \mathcal{S}$ and $k \in \mathbb{Z}^+$, $n_{k+1} \geq 2n_k$. Furthermore, since $k \in \mathbb{Z}^+$, $\frac{2}{k+1} \geq \frac{1}{k}$. Consequently, $\frac{n_{k+1}}{k+1} \geq \frac{2n_k}{k+1} = (\frac{2}{k+1})n_k \geq (\frac{1}{k})n_k = \frac{n_k}{k}$. ◀

▶ **Corollary 4.** *For all $\vec{n} \in \mathcal{S}$ and all $k, k' \in \mathbb{Z}^+$ where $k' \geq k$, $\frac{n'_k}{k'} \geq \frac{n_k}{k}$.*

▶ **Observation 5.** *For all $\vec{n} \in \mathcal{S}$ and all $k \in \mathbb{Z}^+$, $n_k \geq k$.*

▶ **Lemma 6.** *For all $\vec{n} \in \mathcal{S}$ and all $k \in \mathbb{Z}^+$, $\mathcal{SA}_{\vec{n}} \not\succeq SA(n_k + 1, k)$.*

Proof. Let $\vec{n} = (n_1, n_2, \ldots, n_k, \ldots)$ be an infinite sequence in $\mathcal{S}$ and let $k \in \mathbb{Z}^+$. Furthermore, let $\mathcal{A}$ be the set of all infinite sequences of non-negative integers. Then for all $a = (a_1, a_2, \ldots, a_k, \ldots) \in \mathcal{A}$ and $b \in \mathbb{N}$, we define the predicate $\mathcal{P}(a, b, \vec{n}, k)$ to be true if and only if the following inequalities are true:

$$b + \sum_{\ell=1}^{\infty} a_\ell n_\ell \geq n_k + 1 \tag{2}$$

$$b + \sum_{\ell=1}^{\infty} a_\ell \ell \leq k \tag{3}$$

By Theorem 2, it suffices to show that $\mathcal{P}(a, b, \vec{n}, k)$ is false for all $a \in \mathcal{A}$ and $b \in \mathbb{N}$.

Case 1. There exists an integer $k' > k$ such that $a_{k'} > 0$.
Then $b + \sum_{\ell=1}^{\infty} a_\ell \ell \geq a_{k'} k' \geq k' > k$. Thus inequality (3) is false, and so $\mathcal{P}(a, b, \vec{n}, k)$ is false.

Case 2. For all $k' \in \mathbb{Z}^+$ such that $k' > k$, we have $a_{k'} = 0$.

Case 2(a). $b + \sum_{\ell=1}^{k} a_\ell = 0$
Since a is a sequence of non-negative integers and b is a non-negative integer, this implies $b = 0$ and for all $1 \leq \ell \leq k$, $a_\ell = 0$. Then $b + \sum_{\ell=1}^{\infty} a_\ell n_\ell = 0 < n_k + 1$. Thus the inequality (2) is false, and so $\mathcal{P}(a, b, \vec{n}, k)$ is false.

[3] We denote by $\mathbb{N}$ the set of all natural numbers, including 0.

Case 2(b). $b + \sum_{\ell=1}^{k} a_\ell > 0$.

Thus either $a_\ell > 0$ for some $1 \leq \ell \leq k$, or $b > 0$. We define the function:

$$f(a, b, \vec{n}, k) = k(b + \sum_{\ell=1}^{k} a_\ell n_\ell) - (n_k + 1)(b + \sum_{\ell=1}^{k} a_\ell \ell)$$

$f(a, b, \vec{n}, k)$ is the left side of (2) multiplied by the right side of (3), minus the left side of (3) multiplied by the right side of (2). Thus for all $a \in \mathcal{A}$ and $b \in \mathbb{N}$, if $\mathcal{P}(a, b, \vec{n}, k)$ is true, then $f(a, b, \vec{n}, k) \geq 0$.

By algebra,

$$\begin{aligned}
f(a, b, \vec{n}, k) &= k(b + \sum_{\ell=1}^{k} a_\ell n_\ell) - (n_k + 1)(b + \sum_{\ell=1}^{k} a_\ell \ell) \\
&= bk - b(n_k + 1) + k(\sum_{\ell=1}^{k} a_\ell n_\ell) - (n_k + 1)(\sum_{\ell=1}^{k} a_\ell \ell) \\
&= b(k - (n_k + 1)) + (\sum_{\ell=1}^{k} k a_\ell n_\ell) - (\sum_{\ell=1}^{k} (n_k + 1) a_\ell \ell) \\
&= b(k - (n_k + 1)) + \sum_{\ell=1}^{k} (k a_\ell n_\ell - (n_k + 1) a_\ell \ell) \\
&= b(k - (n_k + 1)) + \sum_{\ell=1}^{k} a_\ell (k n_\ell - (n_k + 1)\ell)
\end{aligned}$$

By Observation 5, $0 > k - (n_k + 1)$. Thus if $b > 0$, then $b(k - (n_k + 1)) < 0$, whereas if $b = 0$, then $b(k - (n_k + 1)) = 0$. By Corollary 4, for all $1 \leq \ell \leq k$,

$$\begin{aligned}
&\frac{n_k}{k} \geq \frac{n_\ell}{\ell} \\
&\Rightarrow n_k \ell \geq k n_\ell \\
&\Rightarrow 0 \geq k n_\ell - n_k \ell \\
&\Rightarrow 0 > k n_\ell - (n_k + 1)\ell
\end{aligned}$$

Thus for all $\ell \in [1..k]$, if $a_\ell > 0$, then $a_\ell(k n_\ell - (n_k + 1)\ell) < 0$, whereas if $a_\ell = 0$, then $a_\ell(k n_\ell - (n_k + 1)\ell) = 0$. Therefore, every term of the sum $b(k - (n_k + 1)) + \sum_{\ell=1}^{k} a_\ell(k n_\ell - (n_k + 1)\ell)$ is either 0 or negative. Furthermore, since $b + \sum_{\ell=1}^{k} a_\ell > 0$, either $a_\ell > 0$ for some $1 \leq \ell \leq k$, or $b > 0$, so at least one of the terms is negative. Therefore, $f(a, b, \vec{n}, k) < 0$, and so $\mathcal{P}(a, b, \vec{n}, k)$ is false.

So, for all $a \in \mathcal{A}$ and $b \in \mathbb{N}$, $\mathcal{P}(a, b, \vec{n}, k)$ is false. Thus $\mathcal{SA}_{\vec{n}} \not\succeq SA(n_k + 1, k)$. ◀

By Observation 1(b), $\mathcal{SA}_{\vec{n}} \succeq R_{\vec{n}}$, so by Lemma 6,

▶ **Corollary 7.** *For all $\vec{n} \in \mathcal{S}$ and all $k \in \mathbb{Z}^+$, $R_{\vec{n}} \not\succeq SA(n_k + 1, k)$.*

Since for all $\vec{n} \in \mathcal{S}$ and all $k \in \mathbb{Z}^+$, $R_{\vec{n}} \succeq SA(n_k, k)$ by Observation 1(a), by Corollary 7,

▶ **Corollary 8.** *For all $\vec{n} \in \mathcal{S}$, $R_{\vec{n}}$ has set agreement power $\vec{n}$.*

We now prove:

▶ **Lemma 9.** *$\mathcal{S}$ is uncountable.*

Proof. We prove this via a standard diagonalization argument. Assume, for contradiction, that $\mathcal{S}$ is countable. In other words, there exists some enumeration $\mathcal{E}$ of all elements in $\mathcal{S}$. For all $i \in \mathbb{Z}^+$ and $j \in \mathbb{Z}^+$, let $\mathcal{E}[i,j]$ denote the j-th number in the i-th sequence of the enumeration $\mathcal{E}$.

Now consider the infinite sequence $\vec{n} = (n_1, n_2, \ldots, n_k, \ldots)$ where:

- $n_1 = \mathcal{E}[1,1] + 1$.
- For all $i \geq 2$, $n_i = \mathcal{E}[i,i] + 2n_{i-1}$.

Since $\mathcal{E}$ is an enumeration of infinite sequences of positive integers, n_i is a positive integer for all $i \in \mathbb{Z}^+$. Furthermore, by construction, n_i is at least twice as large as n_{i-1} for all $i \geq 2$. Consequently, $\vec{n}$ is an infinite sequence of positive integers where each integer is at least twice as large as its predecessor, so $\vec{n} \in \mathcal{S}$. By construction, however, $n_i \neq \mathcal{E}[i,i]$ for all $i \in \mathbb{Z}^+$, so $\vec{n}$ is not any of the infinite sequences in the enumeration $\mathcal{E}$ of $\mathcal{S}$. This is a contradiction, so we conclude that $\mathcal{S}$ is uncountable. ◀

▶ **Theorem 10.** *There are uncountably many objects that are not equivalent to each other.*

Proof. By Lemma 9, $\mathcal{S}$ is uncountable. By Corollary 8, for all $\vec{n} \in \mathcal{S}$, there is an object with set agreement power $\vec{n}$. So there are uncountably many objects with distinct set agreement power. By definition, objects with different set agreement power are not equivalent. ◀

Note that the uncountably many objects that Theorem 10 refers to were constructed using the "black-box" set agreement objects described in [5, 9]. In Section 5, we strengthen this result by proving that it holds even for *linearizable* objects [13]. To do this, we first prove that each set agreement task is equivalent to a linearizable object.

4 Linearizable Set Agreement Objects

Henceforth, we consider objects with *ports*. With such an object, each process can apply any operation to any port $i \in [1..n]$, and must then wait for a response from that port. We assume that accesses to the ports are *well-formed*: no port is accessed concurrently by multiple processes; i.e., while an operation on a port is pending, no process can apply another operation on that port. If accesses to the object are not well-formed, the behaviour of the object is undefined.

For all $n, k \in \mathbb{Z}^+$, we now define $LSA(n,k)$, a simple *linearizable* object that we will prove is *equivalent* to the (n,k)-set agreement task in the following sense: the (n,k)-set agreement task can be solved using the $LSA(n,k)$ object, and the $LSA(n,k)$ object can be implemented using any solution to the (n,k)-set agreement task (and registers).

The behaviour of the $LSA(n,k)$ object when it is accessed sequentially is given by its *sequential specification*, described below. The behaviour of $LSA(n,k)$ when it is accessed concurrently (in a well-formed manner) is *linearizable* [13].

The sequential specification of $LSA(n,k)$ is given by Algorithm 1. $LSA(n,k)$ has n *ports*: each process can apply a PROPOSE(v) operation for any value v to any port $i \in [1..n]$. The state of the $LSA(n,k)$ object consists of:

- The set V_{in} of all values proposed to $LSA(n,k)$; V_{in} is initially empty.
- The set V_{out} of all values returned by $LSA(n,k)$; V_{out} is initially empty.

The sequential specification of $LSA(n,k)$ can be formally given in terms of a set of states, a set of operations, a set of responses, and a state transition relation. For brevity, we omit this formal definition here.

From the above definition of $LSA(n,k)$, we have:

Algorithm 1 Sequential specification of the $LSA(n,k)$ object.

Code for port $i \in [1..n]$:

1: **procedure** PROPOSE(v)
2: $V_{in} \leftarrow V_{in} \cup \{v\}$
3: Let v' be nondeterministically chosen from V_{in} such that $|V_{out} \cup \{v'\}| \leq k$.
4: $V_{out} \leftarrow V_{out} \cup \{v'\}$
5: **return** v'

▶ **Observation 11.** *For all $n, k \in \mathbb{Z}^+$, if the $LSA(n,k)$ object is accessed in a well-formed manner, it satisfies:*

- *k-**agreement**: There are at most k distinct return values.*
- ***Validity**: If an operation op returns a value v, then v was proposed by op or by an operation linearized before op.*

This implies:

▶ **Observation 12.** *For all $n, k \in \mathbb{Z}^+$, the (n,k)-set agreement task can be solved using an $LSA(n,k)$ object.*

We now show that the converse also holds: for all $n, k \in \mathbb{Z}^+$, given any algorithm that solves the (n,k)-set agreement task, one can implement the linearizable object $LSA(n,k)$. This implementation of $LSA(n,k)$, shown in Algorithm 2, uses:

- $P[1..n]$: any algorithm that solves the (n,k)-set agreement task, where $P[i]$ is the protocol executed by process i.
- $X[1..n]$: an atomic snapshot object with n fields, initially all **nil**; each $X[i]$ stores a value output by $P[i]$.
- $R[1..n]$: an array of registers, initially all **nil**; each $R[i]$ stores the return value of the first operation performed on port i ($R[i]$ is used to have all operations on port i return the same value).

Note that the atomic snapshot object $X[1..n]$ can be implemented using only registers [1].

To perform an operation PROPOSE(v) on port i, a process executes the following steps:

1. It reads $R[i]$ and returns the same value as the previous operation on port i, if it exists (line 3).
2. It executes protocol $P[i]$ with proposal value v, and stores the decided value into a temporary local variable $pval_i$ (line 4).
3. It writes $pval_i$ to the i-th field of the atomic snapshot X (line 5), letting the other processes know that $pval_i$ was decided by protocol $P[i]$.
4. It takes a snapshot of X (line 6) to check whether its *own* proposed value v was decided by any protocol of $P[1..n]$; if so, it writes v to $R[i]$ (line 7), otherwise it writes $pval_i$ to $R[i]$ (line 8).
5. It returns the value in $R[i]$ (line 9).

Note that, for every $i \in [1..n]$, protocol $P[i]$ is only executed by the first operation on port i, and thus it is executed *at most once*.

▶ **Theorem 13.** *For all $n, k \in \mathbb{Z}^+$, Algorithm 2 implements the linearizable object $LSA(n,k)$ using any algorithm $P[1..n]$ that solves the (n,k)-set agreement task and registers.*

Proof. Let H be any history of this implementation of the $LSA(n,k)$ object where accesses to ports are well-formed.

Algorithm 2 Implementing the linearizable object $LSA(n,k)$ using any algorithm $P[1..n]$ that solves the (n,k)-set agreement task.

```
1: Code for port i ∈ [1..n]:
2: procedure PROPOSE(v)
3:     if R[i].READ() ≠ nil then return R[i].READ()
4:     pval_i ← P[i].EXECUTE(v)
5:     X[i] ← pval_i
6:     X' ← X.SCAN()
7:     if v ∈ X' then R[i].WRITE(v)
8:     else R[i].WRITE(pval_i)
9:     return R[i].READ()
```

▶ **Claim 14** (k-agreement). *In H, there are at most k distinct return values.*

Proof. A value v is returned by an operation on port i only if $v \neq$ **nil** has been written into $R[i]$ (line 3 or 9). A value v is written into $R[i]$ only if v was returned by X.SCAN() (lines 6 to 7) or v has been written into $pval_i$ (line 8). For $v \neq$ **nil** to be returned by X.SCAN(), v must have been written into X by an operation on some port i' (line 5), and therefore previously written into $pval_{i'}$. For all $j \in [1..n]$, v is assigned to $pval_j$ only if v was returned by the protocol $P[j]$ (line 4). Thus any value returned by the object must have previously been returned by some protocol of the (n,k)-set agreement task solution. Since each protocol $P[j]$ for $j \in [1..n]$ is executed at most once, at most k distinct values are returned by the protocols and hence the object. ◀

We now construct a completion H' of H as follows: for port i with an incomplete operation op, if op is the first operation on port i and some other port has a complete operation op' in H that returns the value proposed by op, complete op immediately after it is invoked by returning its own proposal value (i.e., the same value that op' returns); otherwise remove op. Next, we construct a linearization L of H' as follows:

1. Linearize every operation that returns its own proposal value at the point it is invoked.
2. Linearize every operation that returns via line 3 at the point it is invoked.
3. Linearize every remaining operation at the point when it takes a snapshot of X (line 6).

From the above, it is clear that every operation is linearized at some point during its execution interval.

▶ **Claim 15** (Validity). *In H', if an operation op returns a value v, then v was proposed by op or by an operation linearized before op in the linearization L of H'.*

Proof. Suppose an operation op on port i proposes v' and returns v. If $v' = v$, the claim holds. Now suppose $v' \neq v$. Recall that when constructing the completion H' of H, incomplete operations are only completed by returning their own proposal values. Thus op is a complete operation in H. There are two cases:

Case 1: op is the first operation on port i.

Then, since op proposes v' and returns $v \neq v'$, from the code of Algorithm 2 and the way we linearize operations, it is clear that:

(1) op obtained $pval_i = v$ from executing $P[i]$.EXECUTE(v') in line 4,
(2) op wrote $pval_i = v$ in X in line 5, and
(3) op is linearized the instant it takes a snapshot of X in line 6.

Since $P[1..n]$ is a solution for the (n,k)-set agreement task, by (1), at least one port i' has an operation op' that starts executing $P[i'].\text{EXECUTE}(v)$ *before or at the same time* as op obtains v in line 4. Clearly, op' proposes v and is invoked *before* it starts executing $P[i'].\text{EXECUTE}(v)$, and so *before* op obtains v in line 4. We now show that op' is linearized before op, and therefore v was proposed by an operation linearized before op. There are three subcases:

Case 1(a): op' is incomplete in H.

First, recall that when constructing the completion H' of H, an incomplete operation in H is completed with its own proposal value if it is the first operation on its port, and its proposal value is decided by some complete operation in H. Since op' executes $P[i'].\text{EXECUTE}(v)$, it is the first operation on port i'. Furthermore, a complete operation in H, namely op, returns the value v that is proposed by op', so op' is completed in H' by returning v immediately after it is invoked. Then, since op' returns its own proposal value, op' is linearized at the moment it is invoked. Thus op' is linearized before op obtains v in line 4. Therefore op' is linearized before op executes line 6, and thus before op is linearized.

Case 1(b): op' is complete in H and it returns the value v that it proposed.

Then op' is linearized at the moment op' is invoked. Thus op' is linearized before before op obtains v in line 4. Therefore op' is linearized before op executes line 6, and thus before op is linearized.

Case 1(c): op' is complete in H and it returns a value different from the value v that it proposed.

Since op' executes $P[i'].\text{EXECUTE}(v)$, it is the first operation on port i'. Thus op' is linearized when it takes a snapshot of X in line 6, and this snapshot does not contain v (otherwise op' would return v). Since the snapshot does not contain v, it occurs *before* op writes v in X in line 5 (see (2) above). Therefore op' is linearized before op executes line 6, and thus before op is linearized.

Case 2: op is not the first operation on port i.

Then the first operation op' on port i also returns the same value v. Thus from case 1, v was proposed by op' or by an operation linearized before op'. Since the history H is well-formed, i.e., operations on port i are not concurrent, op' is linearized before op. So v was proposed by an operation linearized before op. ◀

Let H_L be the *sequential history* obtained by ordering all the operations in the complete history H' by their linearization points in L. Since the completion of H to H', and the linearization of H' to H_L does not introduce new return values, from Claim 14 we have:

▶ **Observation 16** (k-agreement). *In H_L, there are at most k distinct return values.*

By the definition of H_L and Claim 15, we have:

▶ **Observation 17** (Validity). *In H_L, if an operation op returns a value v, then v was proposed by op or by an operation before op.*

To prove that Algorithm 2 implements the linearizable object $LSA(n,k)$, it suffices to show that H_L satisfies the sequential specification of $LSA(n,k)$.

Suppose, for contradiction, that H_L violates the sequential specification of $LSA(n,k)$, and let op be the first operation in H_L that does so. Let v be the value proposed by op, and v' be the value returned by op. According to the sequential specification of $LSA(n,k)$, v' should be such that (i) v' is in $V_{in} \cup \{v\}$, and (ii) $|V_{out} \cup \{v'\}| \leq k$. Thus, since op violates the

sequential specification of $LSA(n,k)$, either (i) v' is not in $V_{in} \cup \{v\}$, or (ii) $|V_{out} \cup \{v'\}| > k$. We consider these two cases below:

Case 1: v' is not in $V_{in} \cup \{v\}$.

By the sequential specification of $LSA(n,k)$, $V_{in} \cup \{v\}$ is the set of all values proposed by *op* and the operations before *op* in H_L. Thus *op* returns a value v' that was *not* proposed by *op* or by an operation that is before *op* in H_L. So H_L violates Observation 17.

Case 2: $|V_{out} \cup \{v'\}| > k$.

By the sequential specification of $LSA(n,k)$, $V_{out} \cup \{v'\}$ is the set of all values returned by *op* and the operations before *op* in H_L. Thus the operations in H_L return more than k distinct values. So H_L violates Observation 16. ◀

▶ **Theorem 18.** *For all $n, k \in \mathbb{Z}^+$, the linearizable object $LSA(n,k)$ is equivalent to the (n,k)-set agreement task, that is:*

(a) *The (n,k)-set agreement task can be solved using $LSA(n,k)$, and*

(b) *$LSA(n,k)$ can be implemented using any algorithm that solves the (n,k)-set agreement task (and registers).*

Proof. Part (a) follows by Observation 12, and Part (b) is immediate from Theorem 13. ◀

5 Uncountability of Linearizable Objects with Distinct Power

In this section, we prove that there are uncountably many linearizable objects with distinct computational power.

For every infinite sequence $\vec{n} = (n_1, n_2, \ldots, n_k, \ldots)$ such that $n_k \in \mathbb{Z}^+$ for all $k \in \mathbb{Z}^+$, we define a *linearizable* "redirection" object $LR_{\vec{n}}$ that is equivalent to the collection $\mathcal{LSA}_{\vec{n}}$ of linearizable objects $\bigcup_{k=1}^{\infty}\{LSA(n_k,k)\}$. The object $LR_{\vec{n}}$ has a port i for every integer $i \in \mathbb{Z}^+$, each process can apply a PROPOSE(v,k) operation for any value v and any integer $k \in \mathbb{Z}^+$ to any port $i \in \mathbb{Z}^+$. Intuitively, when an operation PROPOSE(v,k) is applied on a port i of the $LR_{\vec{n}}$ object, the operation PROPOSE(v) is applied to port i of the $LSA(n_k,k)$ object in $\mathcal{LSA}_{\vec{n}}$ and its response is returned. If no such port exists (because $i > n_k$), the operation simply returns $\perp$ without changing the state.

The behaviour of the $LR_{\vec{n}}$ object when it is accessed sequentially is given by its *sequential specification*, described below. The behaviour of $LR_{\vec{n}}$ when it is accessed concurrently (in a well-formed manner) is *linearizable* [13].

The sequential specification of $LR_{\vec{n}}$ is given by Algorithm 3. The state of the $LR_{\vec{n}}$ object consists of the following:

- For all $k \in \mathbb{Z}^+$, the set V_{in}^k of all the values proposed by PROPOSE$(-,k)$ operations on ports 1 to n_k of $LR_{\vec{n}}$; V_{in}^k is initially empty.
- For all $k \in \mathbb{Z}^+$, the set V_{out}^k of all the values returned by PROPOSE$(-,k)$ operations on ports 1 to n_k of $LR_{\vec{n}}$; V_{out}^k is initially empty.

From the above, it is clear that the sequential specification of $LR_{\vec{n}}$ can be formally given in terms of a set of states, a set of operations, a set of responses, and a state transition relation. For brevity, we omit this formal definition here.

▶ **Observation 19.** *For all infinite sequences $\vec{n} = (n_1, n_2, \ldots, n_k, \ldots)$ such that $n_k \in \mathbb{Z}^+$ for all $k \in \mathbb{Z}^+$, we have that:*

(a) *For all $k \in \mathbb{Z}^+$, $LR_{\vec{n}} \succeq LSA(n_k,k)$; so $LR_{\vec{n}} \succeq \mathcal{LSA}_{\vec{n}}$.*

(b) *$\mathcal{LSA}_{\vec{n}} \succeq LR_{\vec{n}}$.*

(c) *Thus $LR_{\vec{n}} \equiv \mathcal{LSA}_{\vec{n}}$.*

Algorithm 3 Sequential specification of the $LR_{\vec{n}}$ object.

Code for port $i \in \mathbb{Z}^+$:

1: **procedure** PROPOSE(v, k)
2: **if** $i > n_k$ **then return** $\perp$
3: $V_{in}^k \leftarrow V_{in}^k \cup \{v\}$
4: Let v' be nondeterministically chosen from V_{in}^k such that $|V_{out}^k \cup \{v'\}| \leq k$.
5: $V_{out}^k \leftarrow V_{out}^k \cup \{v'\}$
6: **return** v'

▶ **Lemma 20.** *For all infinite sequences $\vec{n} = (n_1, n_2, \ldots, n_k, \ldots)$ such that $n_k \in \mathbb{Z}^+$ for all $k \in \mathbb{Z}^+$, we have that the linearizable object $LR_{\vec{n}}$ is equivalent to the object $R_{\vec{n}}$.*

Proof. By Observation 1(c), $R_{\vec{n}} \equiv \mathcal{SA}_{\vec{n}}$, where $\mathcal{SA}_{\vec{n}}$ is the collection of "black-box" set agreement objects: $\bigcup_{k=1}^{\infty}\{SA(n_k, k)\}$. By Observation 19(c), $LR_{\vec{n}} \equiv \mathcal{LSA}_{\vec{n}}$, where $\mathcal{LSA}_{\vec{n}}$ is the collection of linearizable set agreement objects: $\bigcup_{k=1}^{\infty}\{LSA(n_k, k)\}$.

By definition, for all $k \in \mathbb{Z}^+$, the "black-box" set agreement object $SA(n_k, k)$ is equivalent to the (n_k, k)-set agreement task. By Theorem 18, for all $k \in \mathbb{Z}^+$, $LSA(n_k, k)$ is equivalent to the (n_k, k)-set agreement task. Thus, for all $k \in \mathbb{Z}^+$, $SA(n_k, k) \equiv LSA(n_k, k)$. Therefore, $\bigcup_{k=1}^{\infty}\{SA(n_k, k)\} \equiv \bigcup_{k=1}^{\infty}\{LSA(n_k, k)\}$, i.e., $\mathcal{SA}_{\vec{n}} \equiv \mathcal{LSA}_{\vec{n}}$. So, by transitivity, $R_{\vec{n}} \equiv LR_{\vec{n}}$. ◀

Consequently, by Corollary 8 and Lemma 20,

▶ **Corollary 21.** *For all $\vec{n} \in \mathcal{S}$, $LR_{\vec{n}}$ has set agreement power $\vec{n}$.*

▶ **Theorem 22.** *There are uncountably many linearizable objects that are not equivalent to each other.*

Proof. By Lemma 9, $\mathcal{S}$ is uncountable. By Corollary 21, for all $\vec{n} \in \mathcal{S}$, there is a linearizable object with set agreement power $\vec{n}$. So there are uncountably many linearizable objects with distinct set agreement power. By definition, objects with different set agreement power are not equivalent. ◀

Thus, there are uncountably many linearizable objects with distinct computational power.

6 Concluding remark

In this paper, we used Theorem 2 to prove that there are uncountably many objects with distinct computational power. We can use the same theorem to prove an interesting result about the *robustness* [14] of classifications of certain objects. Consider the subset of shared objects $\mathcal{U}_C \in \mathcal{U}$ that are equivalent to their set agreement power, namely:

$$\mathcal{U}_C = \{O \mid O \equiv \bigcup_{k=1}^{\infty}\{SA(n_k, k)\} \text{ where } \vec{n} = (n_1, n_2, \ldots) \text{ is the set agreement power of } O\}$$

In some sense, $\mathcal{U}_C$ is a generalization of the family of objects known as *Common2* [3, 4], which is the set of objects that are equivalent to the 2-consensus object. Thus $\mathcal{U}_C$ contains every object in Common2, which includes several common objects such as *stack*, *swap*, *fetch&add*, and *test&set* [3, 4].

If we restrict Herlihy's [12] consensus hierarchy to $\mathcal{U}_C$, we can prove the resulting hierarchy is *robust* [14] in the following sense: in $\mathcal{U}_C$, for all $n \in \mathbb{Z}^+$, any non-empty set of objects with

consensus number at most n cannot be used to implement any object with consensus number $n' > n$. In fact, we can prove a more general result, as we now describe.

To describe our robustness result, we first define what it means for a sequence to *dominate* another. Given any pair of sequences $\vec{v} = (v_1, v_2, \ldots, v_\ell)$ and $\vec{v}' = (v'_1, v'_2, \ldots, v'_\ell)$ of the same length $\ell \in \mathbb{Z}^+$, we say that v *dominates* v', denoted $\vec{v} \geq \vec{v}'$, if for all $k \in [1..\ell]$, $v_k \geq v'_k$; similarly, we say that v *strictly dominates*, denoted $\vec{v} > \vec{v}'$, if $\vec{v} \geq \vec{v}'$ and $\vec{v} \neq \vec{v}'$.

Consider the set of objects $\mathcal{U}_C$ that are equivalent to their set agreement power. For any integer $\ell \in \mathbb{Z}^+$, we can partition $\mathcal{U}_C$ into equivalence classes such two objects are in the same class if and only if the first ℓ components of their set agreement powers are the same; we call this an *ℓ-partition of $\mathcal{U}_C$*, and denote it $\mathcal{P}_\ell$ (note that the 1-partition of $\mathcal{U}_C$ is simply Herlihy's consensus hierarchy restricted to the objects in $\mathcal{U}_C$ [12]). Let C be any equivalence class of $\mathcal{P}_\ell$. By definition, the first ℓ components of the set agreement power of every object in C is some sequence $\vec{v} = (v_1, v_2, \ldots, v_\ell)$; this sequence is the *label* of C. If C and C' are equivalence classes of $\mathcal{P}_\ell$ with labels $\vec{v}$ and $\vec{v}'$ respectively, we say that C *dominates* C' if $\vec{v} \geq \vec{v}'$, and C *strictly dominates* C' if $\vec{v} > \vec{v}'$.

Our generalized robustness result for can now be stated as follows: Consider the ℓ-partition $\mathcal{P}_\ell$ of $\mathcal{U}_C$, and let C be any equivalence class of $\mathcal{P}_\ell$. Objects in equivalence classes that are dominated by C cannot implement objects in equivalence classes that strictly dominate C. A proof of this result is given in the appendix.

References

1. Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, Sep 1993. doi:10.1145/153724.153741.
2. Yehuda Afek, Faith Ellen, and Eli Gafni. Deterministic objects: Life beyond consensus. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC'16, pages 97–106, New York, NY, USA, 2016. ACM. doi:10.1145/2933057.2933116.
3. Yehuda Afek, Eli Gafni, and Adam Morrison. Common2 extended to stacks and unbounded concurrency. *Distributed Computing*, 20(4):239–252, 2007. doi:10.1007/s00446-007-0023-3.
4. Yehuda Afek, Adam Morrison, and Guy Wertheim. From bounded to unbounded concurrency objects and back. In *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC'11, pages 119–128, New York, NY, USA, 2011. ACM. doi:10.1145/1993806.1993823.
5. Elizabeth Borowsky and Eli Gafni. The implication of the Borowsky-Gafni simulation on the set-consensus hierarchy. Technical Report 930021, UCLA Computer Science Dept., July 1993.
6. Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. Specifying concurrent problems: Beyond linearizability and up to tasks. In Yoram Moses, editor, *Distributed Computing: 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, pages 420–435, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. doi:10.1007/978-3-662-48653-5_28.
7. David Yu Cheng Chan, Vassos Hadzilacos, and Sam Toueg. Bounded disagreement. In Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone, editors, *20th International Conference on Principles of Distributed Systems (OPODIS 2016)*, volume 70 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.OPODIS.2016.5.

8 Soma Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132–158, 1993. doi:http://dx.doi.org/10.1006/inco.1993.1043.

9 Soma Chaudhuri and Paul Reiners. Understanding the set consensus partial order using the Borowsky-Gafni simulation. In *Distributed Algorithms*, volume 1151 of *Lecture Notes in Computer Science*, pages 362–379. Springer Berlin Heidelberg, 1996. doi:10.1007/3-540-61769-8_23.

10 Carole Delporte-Gallet, Hugues Fauconnier, Eli Gafni, and Petr Kuznetsov. Set-Consensus Collections are Decidable. In Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone, editors, *20th International Conference on Principles of Distributed Systems (OPODIS 2016)*, volume 70 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:15, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.OPODIS.2016.7.

11 Maurice Herlihy. Impossibility results for asynchronous PRAM (extended abstract). In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA'91, pages 327–336, New York, NY, USA, 1991. ACM. doi:10.1145/113379.113409.

12 Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 11(1):124–149, Jan 1991. doi:10.1145/114005.102808.

13 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, jul 1990. doi:10.1145/78969.78972.

14 Prasad Jayanti. On the robustness of herlihy's hierarchy. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing*, PODC'93, pages 145–157, New York, NY, USA, 1993. ACM. doi:10.1145/164051.164070.

15 Gil Neiger. Set-linearizability. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC'94, pages 396–, New York, NY, USA, 1994. ACM. doi:10.1145/197917.198176.

16 Ophir Rachman. Anomalies in the wait-free hierarchy. In *Proceedings of the 8th International Workshop on Distributed Algorithms*, WDAG'94, pages 156–163, London, UK, UK, 1994. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=645951.675479.

A Appendix

▶ **Theorem 23.** *Consider the ℓ-partition $\mathcal{P}_\ell$ of $\mathcal{U}_C$, and let C be any equivalence class of $\mathcal{P}_\ell$. Objects in equivalence classes that are dominated by C cannot implement objects in equivalence classes that strictly dominate C.*

Proof. Assume, for contradiction, that there is an equivalence class C of $\mathcal{P}_\ell$ and a set of objects $\mathcal{O}$ such that:

(a) every object in $\mathcal{O}$ is in an equivalence class that is dominated by C;
(b) $\mathcal{O}$ implements an object O', i.e., $\mathcal{O} \succeq O'$; and
(c) O' is in an equivalence class C' that strictly dominates C.

Let $\vec{v} = (v_1, v_2, \ldots, v_\ell)$ and $\vec{v}' = (v'_1, v'_2, \ldots, v'_\ell)$ be the labels of C and C', respectively. Since C' dominates C, $\vec{v}' > \vec{v}$, and so there is a $d \in [1..\ell]$ such that $v'_d > v_d$. Since O' is in C', and the label of C' is $\vec{v}' = (v'_1, v'_2, \ldots, v'_\ell)$, O' implements the (v'_k, k)-set agreement object $SA(v'_k, k)$ for each $k \in [1..\ell]$. In particular, O' implements $SA(v'_d, d)$, i.e., $O' \succeq SA(v'_d, d)$.

Henceforth, O^i is an arbitrary object in $\mathcal{O}$. Let $\vec{n}^i = (n^i_1, n^i_2, \ldots, n^i_k, \ldots)$ be the set agreement power of O^i. Since every object in $\mathcal{O}$ (including O^i) is in an equivalence class that is dominated by C, and C has label $\vec{v} = (v_1, v_2, \ldots, v_\ell)$, we have that for all $k \in [1..\ell]$,

$n_k^i \leq v_k$. Let $\mathcal{SA}_{\vec{n}^i}$ be the collection of set agreement objects $\bigcup_{k=1}^{\infty}\{SA(n_k^i, k)\}$. Since $O^i \in \mathcal{U}_C$, by definition of $\mathcal{U}_C$, $O^i \equiv \mathcal{SA}_{\vec{n}^i}$.

Let $\vec{n} = (n_1, n_2, \ldots, n_k, \ldots)$ be the infinite sequence such that for all $k \in [1..\ell]$, $n_k = v_k$, and for all $k > \ell$, $n_k = \infty$. Thus for all $k \in \mathbb{Z}^+$, $n_k \geq n_k^i$. Let $\mathcal{SA}_{\vec{n}}$ be the collection of set agreement objects $\bigcup_{k=1}^{\infty}\{SA(n_k, k)\}$. Since for all $k \in \mathbb{Z}^+$, $n_k \geq n_k^i$, we have $\mathcal{SA}_{\vec{n}} \succeq \mathcal{SA}_{\vec{n}^i}$. Since $O^i \equiv \mathcal{SA}_{\vec{n}^i}$, $\mathcal{SA}_{\vec{n}} \succeq O^i$. Recall that O^i is an arbitrary object in $\mathcal{O}$, so $\mathcal{SA}_{\vec{n}}$ implements every object in $\mathcal{O}$, i.e., $\mathcal{SA}_{\vec{n}} \succeq \mathcal{O}$. Since $\mathcal{O} \succeq O'$ and $O' \succeq SA(v'_d, d)$, we have $\mathcal{SA}_{\vec{n}} \succeq SA(v'_d, d)$.

Since $\mathcal{SA}_{\vec{n}} = \bigcup_{k=1}^{\infty}\{SA(n_k, k)\}$ implements $SA(v'_d, d)$, by Theorem 2, there is an infinite sequence $(a_1, a_2, \ldots)$ of non-negative integers and an integer $b \in \mathbb{N}$ such that:

$$b + \sum_{k=1}^{\infty} a_k n_k \geq v'_d$$
$$b + \sum_{k=1}^{\infty} a_k k \leq d$$

Note that for all $k > d$, $a_k = 0$, otherwise $b + \sum_{k=1}^{\infty} a_k k > d$. Thus we have:

$$b + \sum_{k=1}^{d} a_k n_k \geq v'_d$$
$$b + \sum_{k=1}^{d} a_k k \leq d$$

Let O^* be an arbitrary object in the equivalence class C, and $\vec{n}^* = (n_1^*, n_2^*, \ldots, n_k^*, \ldots)$ be the set agreement power of O^*. Since O^* is in C and the label of C is $\vec{v} = (v_1, v_2, \ldots, v_\ell)$, for all $k \in [1..\ell]$, $n_k^* = v_k = n_k$. Thus, since $d \in [1..\ell]$, we have:

$$b + \sum_{k=1}^{d} a_k n_k^* \geq v'_d$$
$$b + \sum_{k=1}^{d} a_k k \leq d$$

Since for all $k > d$, $a_k = 0$, we have:

$$b + \sum_{k=1}^{\infty} a_k n_k^* \geq v'_d$$
$$b + \sum_{k=1}^{\infty} a_k k \leq d$$

Let $\mathcal{SA}_{\vec{n}^*}$ be the collection of set agreement objects $\bigcup_{k=1}^{\infty}\{SA(n_k^*, k)\}$. By the above equations and Theorem 2, $\mathcal{SA}_{\vec{n}^*} \succeq SA(v'_d, d)$. Since the set agreement power of O^* is $\vec{n}^* = (n_1^*, n_2^*, \ldots, n_k^*, \ldots)$ and $O^* \in \mathcal{U}_C$, by definition of $\mathcal{U}_C$, we have that $O^* \equiv \mathcal{SA}_{\vec{n}^*}$. Thus $O^* \succeq SA(v'_d, d)$. Hence the d-set agreement number of O^* is at least $v'_d > v_d$. However, recall that for all $k \in [1..\ell]$, $n_k^* = v_k$, so in particular $n_d^* = v_d$. Therefore the d-set agreement number of O^* is v_d — a contradiction. ◀

Fast Plurality Consensus in Regular Expanders*

Colin Cooper[1], Tomasz Radzik[2], Nicolás Rivera[3], and Takeharu Shiraga[4]

1 **Department of Informatics, King's College London, UK**
colin.cooper@kcl.ac.uk

2 **Department of Informatics, King's College London, UK**
tomasz.radzik@kcl.ac.uk

3 **Department of Informatics, King's College London, UK**
nicolas.rivera@kcl.ac.uk

4 **Department of Information and System Engineering, Chuo University, Japan**
shiraga@ise.chuo-u.ac.jp

Abstract

In a *voting process* on a graph vertices revise their opinions in a distributed way based on the opinions of nearby vertices. The voting completes when the vertices reach consensus, that is, they all have the same opinion. The classic example is synchronous pull voting where at each step, each vertex adopts the opinion of a random neighbour. This very simple process, however, can be slow and the final opinion is not necessarily the one with the initial largest support. It was shown earlier that if there are initially only two opposing opinions, then both these drawbacks can be overcome by a synchronous *two-sample* voting, in which at each step each vertex considers its own opinion and the opinions of two random neighbours.

If there are initially three or more opinions, a problem arises when there is no clear majority. One class of opinions may be largest (the plurality opinion), although its total size is less than that of two other opinions put together. We analyse the performance of the two-sample voting on d-regular graphs for this case. We show that, if the difference between the initial sizes A_1 and A_2 of the largest and second largest opinions is at least $Cn\max\{\sqrt{(\log n)/A_1}, \lambda\}$, then the largest opinion wins in $O((n\log n)/A_1)$ steps with high probability. Here C is a suitable constant and λ is the absolute second eigenvalue of transition matrix $P = Adj(G)/d$ of a simple random walk on the graph G. Our bound generalizes the results of Becchetti et al. [SPAA 2014] for the related *three-sample voting process* on complete graphs. Our bound implies that if $\lambda = o(1)$, then the two-sample voting can consistently converge to the largest opinion, even if $A_1 - A_2 = o(n)$. If λ is constant, we show that the case $A_1 - A_2 = o(n)$ can be dealt with by sampling using short random walks. Finally, we give a simple and efficient *push* voting algorithm for the case when there are a number of large opinions and any of them is acceptable as the final winning opinion.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Plurality consensus; Regular expanders

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.13

* Colin Cooper and Tomasz Radzik were supported in part by EPSRC grant EP/M005038/1, "Randomized algorithms for computer networks". Nicolás Rivera was supported by funding from Becas CHILE. Takeharu Shiraga was supported by JSPS KAKENHI Grant Number 15J03840. Work carried out while Takeharu Shiraga was visiting King's College London with the support of the ELC project (Grant-in-Aid for Scientific Research on Innovative Areas MEXT Japan).

© Colin Cooper, Tomasz Radzik, Nicolás Rivera, and Takeharu Shiraga;
licensed under Creative Commons License CC-BY

1 Introduction

The problem of reaching consensus in a graph by means of local interactions is an abstraction of such behavior in human society as well as some processes in computer networks. In a *voting process* on a graph, vertices revise their opinions in a systematic and distributed way based on opinions of other vertices, typically on the opinions of a sample of their local neighbours. The aim is that eventually a single opinion will emerge, and that this opinion will reflect the relative importance of the original mix of opinions in some way. Voting processes have application in various fields of computing including consensus and leader election in large networks [9, 22], serialisation of read/write in replicated data-bases [21], and analysis of social behavior [16]. In general, a voting process should be conceptually simple, fast, fault-tolerant and straightforward to implement [22, 23].

In a synchronous voting process each vertex of a connected graph has one of several possible opinions. In each time-step, each vertex, using the same protocol, queries the opinion of one or more of its neighbours and decides whether to modify or to keep its current opinion. A simple voting protocol is ideally memoryless: in the current step, each vertex uses only its current opinion and the current opinions of the queried neighbours. When all vertices have a common (and thus final) opinion, we say a consensus has been reached. For a given voting process, the main quantities of interest are the probability that a particular opinion wins and the expected time to reach consensus.

In the classical *voter model* each vertex initially has a distinct opinion, but in general we assume that each vertex holds one of k different opinions. The simplest case, *two party voting*, is when there are initially two opinions ($k = 2$). If there are at least three opinions ($k \geq 3$), then the problem is often referred to as *plurality consensus*. We would like the dominant opinion to eventually become the final opinion of all vertices. The probability of this, however, strongly depends on the voting process.

Pull voting

The most well known voting process is synchronous pull voting. In this model, at each step each vertex changes its opinion to that of a random neighbour. We assume henceforth that the graphs which we consider are connected and non-bipartite, so that a consensus is possible. For such a graph, the probability that pull voting ends with a particular opinion taking over the whole graph is proportional to the initial degree of this opinion in the graph [22]. More precisely, if A is the set of vertices initially holding a given opinion, then

$$\mathbf{Pr}(A \text{ wins in the voting process}) = \sum_{v \in A} \frac{d(v)}{2m} = \frac{d(A)}{2m}, \tag{1}$$

where $d(v)$ is the degree of vertex v and m is the number of edges in the graph. Surprisingly, the probability here depends only on the voting process and the total degree $d(A)$, but does not depend on the details of the initial arrangement of opinions on the graph.

For an n-vertex graph, let $\mathbf{E}(T)$ be the expected value of the time to consensus T. Much of the early work was on analysing $\mathbf{E}(T)$ for classical pull voting in an asynchronous model in a continuous time setting. Here the vertices have independent exponentially distributed waiting times (Poisson clocks); see e.g. Cox [15] and Aldous [1]. In the synchronous model the expected time to consensus is $O(H_{\max} \log n)$, where $H_{\max} = O(n^3)$ is the maximum hitting time of any vertex by a random walk; see Aldous and Fill [2]. For regular expanders the expected consensus time is $\Theta(n)$, see [12].

Two- and three-sample voting

Since the classical pull voting tends to be slow ($\mathbf{E}(T) = \Theta(n)$ for regular expanders) and may be viewed as undemocratic (giving only weak preference for the largest initial opinion as shown in (1)), there has been considerable interest in modifying this simple voting process to avoid these two problems. Instead of taking the opinion of only one neighbour, the next simplest approach is to sample the opinions of a larger number of neighbours (say two or three), compare them in some way, and hope that the so-called 'power of two choices' improves the performance of voting. The consequences of this approach are as follows. Firstly, the number of neighbours queried affects the consensus time and the voting outcome. Secondly, the relative size of the opinions affects the ability of the process to ensure that the largest initial opinion wins.

In this setting we study the following protocols for two-sample and three-sample voting. In the two-sample voting model, at each step, each vertex v chooses two random neighbours with replacement, and if the selected vertices have the same opinion, then v adopts it; otherwise v keeps its current opinion. In the three-sample voting model, each vertex v chooses three random neighbours with replacement and adopts the majority opinion among them. If there is no majority, v picks the opinion of the first sampled neighbour. Other rules are equally possible here, e.g. v keeps its opinion. The rule we choose is the one used by Becchetti et al. [4].

For d-regular expanders, two-sample voting was studied in [13] for the case where there are initially two opinions ($k = 2$). Provided the initial difference between the sizes of the two opinions is sufficiently large, the initial majority wins with high probability (w.h.p.)[1] and voting is completed in $\mathcal{O}(\log n)$ steps. This is tight since the diameter of a d-regular graph is $\Omega(\log n)$ for constant d. This result is extended in [14] to non-regular expander graphs.

Two- and three-sample voting for plurality consensus

Not so much is known about improving the performance of voting by using two or more samples in the case where there are initially three or more opinions ($k \geq 3$). Generally, analysing plurality voting protocols tends to be more involved than analysing two party voting. The case when some opinion has an absolute majority can usually be reduce to two-party voting by grouping the other opinions into a single minority class. Difficulties arise when there is no clear majority, that is, when the largest opinion is smaller than two (or more) other opinions put together. We note that the well established techniques used in analysis of the classical pull voting ('single-sample' voting), for example the correspondence with multiple coalescing random walks [1, 12], do not have ready extensions or generalisations to multi-sample voting.

Plurality consensus using the three-sample voting protocol given above was studied by Becchetti et al. [4]. They proved that for the complete graph, if the difference between the initial sizes A_1 and A_2 of the largest and second largest opinions is at least $24n\sqrt{2(\log n)/A_1}$, then the largest opinion wins in $O((n \log n)/A_1)$ steps w.h.p. They also showed that this result is tight for some ranges of the parameters. Subsequently Becchetti et al. [3] analyse another simple plurality voting protocol, which can be viewed as a variation of two-sampling, showing that polylog convergence on complete graphs can be achieved even if $A_1 = o(n/\text{polylog}n)$, provided that $A_1/A_2 \geq \alpha > 1$ for a constant α and only $O(\text{polylog}n)$ opinions are initially 'comparable' in size with the largest opinion. Detailed parameterized bounds are presented

[1] "With high probability" (w.h.p.) means in this paper probability at least $1 - n^{-\alpha}$, for a constant $\alpha > 0$.

in [3], but they do not improve the worst-case general bound given in [4]. Two- and three-sample voting is memoryless, so requires only k states ($\log k$ bits) per vertex to store the current opinion held by this vertex. The protocol in [3] requires $k+1$ states, as it allows each vertex to hold the "undecided" opinion.

Elsässer et al. [17] consider asynchronous two-sample voting and show that it converges on complete graphs within $O(k \log n)$ rounds, subject to suitable bounds on the number of opinions and the initial difference between the largest and the second largest opinions. Becchetti et al. [5] analyse robustness of three-sample voting and show that it converges on complete graphs in the number of rounds polynomial in k and $\log n$, even if an adversary corrupts $o(\sqrt{n})$ vertices in each round. Subsequently, Ghaffari and Lengler [19] improve the number of rounds to $O(k \log n)$, allowing at the same time for a stronger adversary.

Berenbrink et al. [6] analyse two-sample and three-sample voting for the case of a large number of initial opinions. They show that three-sample voting converges on complete graphs in $O(n^{3/4} \log n)$ rounds for any number of initial opinions $k \leq n$, but two-sample voting requires $\Omega(n/\log n)$, if initially each opinion is supported by only $O(\log n)$ vertices.

Push voting

The voting processes which we have discussed so far are examples of *pull protocols*: each vertex 'pulls' the information from its (selected) neighbours. In a *push protocol*, a vertex 'pushes' its own information onto its neighbours. While the push communication paradigm is natural and effective in rumor spreading (broadcasting) protocols, it has found so far only limited use in voting protocols. It is not clear how synchronous push voting could be defined, so push voting has been mostly confined to asynchronous processes. An example is the work of Copper et al. [11], who consider both pull and push voting in the context of asynchronous 'discordant voting.' Elsässer et al. [17] develop an asynchronous plurality consensus algorithm which combines two-sample voting with push-pull broadcasting. In the current paper, we propose a general simple framework for push voting.

Other related previous work

Berenbrink et al. [7] propose two synchronous pull-based plurality-consensus protocols for complete graphs. Their protocols achieve plurality consensus in $O(\log k \log\log n)$ rounds with $\log k + \Theta(\log\log k)$ bits of memory per vertex and in $O(\log n \log\log n)$ rounds with $\log k + 4$ bits of memory per vertex, provided a sufficient initial bias towards the largest opinion. Independently, Ghaffari and Parter [20] showed a plurality consensus algorithm of a similar type, which converges on complete graphs in $O(\log n \log k)$ rounds and requires $\log k + O(1)$ bits of memory per vertex.

While [3, 4, 5, 7, 17, 20] analyse plurality voting only on complete graphs, Berenbrink et al. [8] consider arbitrary connected graphs. They show two protocols, which are based on earlier work on distributed load balancing, and show a detailed analysis of their performance in various communication models. They achieve, for example, a w.h.p. $O(\log n)$ bound on the number of rounds for expanders in *diffusion model* (in each round each vertex exchanges messages with all its neighbours), but each vertex requires $\Theta((n/(A_1 - A_2))^2 \log n \log k)$ bits of memory. In contrast, two- and three-sample voting, as well as protocols of the type considered in [3], are very simple protocols requiring only $O(\log n)$ bits of memory per vertex.

2 Our contributions

Our contributions are two-fold. Firstly we extend the analysis of plurality consensus on the complete graph (using voting) to the case of regular graphs. Secondly we give a push based algorithm which reaches consensus on the complete graph in four rounds, and apply this to the problem of approximate plurality consensus.

Plurality consensus on regular graphs

The earlier analysis of plurality consensus on the complete graph can be extended to regular graphs. For regular expanders our results have the same asymptotic convergence time as given for K_n in [4]. We use the two-sample voting process of [14], but generalize the analysis from two-party voting to k-party voting.

Let G be a connected regular n-vertex graph and let λ be the second largest absolute eigenvalue of the transition matrix $P = P(G)$ of a random walk on G. Let A_1 be the set of vertices with the largest initial opinion and A_2 the set with the second largest opinion. If no confusion arises, we also let A stand for the size of set A.

▶ **Theorem 1.** *Let G be a regular n-vertex graph and let the initial sizes of the opinions be $A_1, A_2, \ldots, A_k$ in non-increasing order. Assume that $A_1 - A_2 \geq Cn \max\{\sqrt{(\log n)/A_1}, \lambda\}$, where λ is the absolute second eigenvalue of $P(G)$ and $C > 0$ is a suitably large constant. With probability at least $1 - 1/n$, after at most $O((n/A_1)\log(A_1/(A_1 - A_2)) + \log n)$ rounds, the two-sample voting completes and the final opinion is the largest initial opinion.*

We note the following w.h.p. property of the second eigenvalue λ for random d-regular graphs for $d = o(n^{1/2})$. For d constant it is a result of Friedman [18] that $\lambda \leq \gamma/\sqrt{d}$, where $\gamma = 2 + \epsilon$ for some small $\epsilon > 0$. For d growing with n, the following estimate of λ is given in [10]. Provided $d = o(n^{1/2})$ there exists constant $\gamma > 0$ such that w.h.p. $\lambda \leq \gamma/\sqrt{d}$. In either case the size separation condition in Theorem 1 is $A_1 - A_2 \geq C'n/\sqrt{d}$.

Theorem 1 can be applied to a number of specific scenarios. Consider, for example, the case where all k opinions are fairly evenly represented, but with one opinion slightly larger than the average n/k. More specifically, assume that $A_1 \geq (n/k)(1 + \varepsilon)$, for some $0 < \varepsilon \leq 1$, and that $A_2 \leq A_1/(1 + \varepsilon)$. Theorem 1 implies the following corollary for this case.

▶ **Corollary 2.** *Let the number of opinions be $k \leq ((1/C)^2 n/\log n)^{1/3}$, $A_1 \geq (n/k)(1 + \varepsilon)$, $A_2 \leq A_1/(1 + \varepsilon)$, and $\lambda \leq \varepsilon/(Ck)$, where $C > 0$ is the constant from Theorem 1 and $\varepsilon = k^{3/2}/((1/C^2)n/\log n)^{1/2} \leq 1$. Then with probability at least $1 - 1/n$, after $O(k \log n)$ rounds the two-sample voting completes and the final opinion is the largest initial opinion.*

In Section 5 we show that the statements of Theorem 1 and Corollary 2 also hold for the three-sample voting protocol analyzed by Becchetti et. al. [4]. We note that the bound on the running time in Theorem 1 is $O(\log n)$, if A_1 is $\Omega(n/\log n)$, provided that $A_1 - A_2$ is also $\Omega(n/\log n)$ and the graph has the λ parameter appropriately small. This includes complete graphs. The bound obtained in [4] for complete graphs is $O(\log n)$ only if $A_1 = \Theta(n)$.

If the graph is not very expansive (λ is too large), we can improve the ability of two-sample voting to discriminate the plurality as follows. In ℓ-extended two-sample voting model (see [14]), each vertex makes two independent random walks of length ℓ and carries out two-sample voting using the opinions on the terminal vertices of these walks. Random walks of length ℓ replace the transition matrix P used in the proof of Theorem 1 by P^ℓ. If the graph is regular, then the only effect on the proofs is to replace all eigenvalues by their ℓ-th power. This reduces the absolute second eigenvalue from λ to λ^ℓ. The effect is to replace the condition $A_1 - A_2 \geq Cn \max\{\sqrt{(\log n)/A_1}, \lambda\}$ of Theorem 1 by the improved condition $A_1 - A_2 \geq Cn \max\{\sqrt{(\log n)/A_1}, \lambda^\ell\}$, and thus diminishing the influence of λ.

Approximate plurality consensus by push voting

In Section 6 we describe a push based algorithm (*Algorithm Pushy*) and show that it reaches consensus on K_n in four push rounds whatever the initial mix of opinions. In each round each vertex activates with some (small) probability p and sends its opinion to all its neighbours (so to all other vertices, if the graph is complete). Each vertex, after receiving the opinions of the activated vertices, tallies up different opinions and adopts the opinion with the highest count, breaking ties uniformly at random.

▶ **Theorem 3.** *On the complete graph, and irrespective of the initial number and distribution of opinions, after four rounds of Algorithm Pushy with the parameter* $p = \Theta((\log n)/n)$*, there is a consensus opinion with probability* $1 - o(1)$*.*

If the opinion sets A_i, $1 \le i \le k$, are all of the same size, then in algorithm Pushy each opinion has the same probability of becoming the consensus opinion, so in this case the algorithm gives a completely fair solution to plurality. If there are few large opinion classes of similar sizes, then while there is no guarantee that the largest opinion wins, there is only small probability that the winning opinion is outside of those large ones. As an example, suppose there are $\ell \ge 1$ classes of size at least $\Theta(n/\ell)$, where $A_1 \ge A_2 \ge \cdots \ge A_\ell$ and $\ell = O(\log n/\log\log n)$, and let $\delta = \sqrt{\ell \log\log n/\log n}$. Then the probability that the winning opinion comes from opinions for which $A_j(1+O(\delta)) \le A_1$ is only $o(1)$. In particular, if $A_2(1+O(\delta)) \le A_1$, then the largest opinion A_1 wins with probability $1 - o(1)$.

We can either use Pushy on its own, or combine it with another consensus algorithm to try to solve plurality approximately but quickly in the following sense. Suppose there is a given value s such that if possible we would like to choose a consensus opinion which was initially supported by at least s vertices. According to (the proof of) Theorem 1, and assuming $s = \Omega(n^{2/3}\log n)$ and that the initial largest opinion size is at least s, then after $T = \Theta((n/s)\log n)$ rounds of two-sample voting, w.h.p. we will have discarded any opinion of initial size at most $s/(1+\varepsilon)$. We then use Pushy to return quickly a consensus opinion chosen from the remaining opinions.

On the complete graph K_n, algorithm Pushy requires $O(\log^2 n)$ storage per vertex, and the constant number of push rounds mean that the total number of message transmissions is $O(n\log n)$, which is the same order as in $O(\log n)$ rounds of two-sample voting. In this paper we focus on proving Theorem 3, leaving for separate investigations possibilities of implementing algorithm Pushy on other interaction models .

3 Preliminary Markov chain results

In this section we establish some technical results used in our proof of Theorem 1. Consider a connected and non-bipartite graph $G = (V, E)$ with n vertices and m edges. Let P be the transition matrix of a simple random walk on G. A random walk on a connected and non-bipartite graph defines a reversible Markov chain with stationary distribution $\pi(x) = d(x)/(2m)$, where $d(x)$ denotes the degree of vertex x. The reversibility of P means that $\pi(x)P(x,y) = \pi(y)P(y,x)$, for all vertices x, y.

Let $1 = \lambda_1 > \lambda_2 \ge \ldots \ge \lambda_n > -1$ be the eigenvalues of P and define $\lambda = \lambda(P)$ by $\lambda = \max\{|\lambda_2|, |\lambda_n|\}$. We also consider the matrix $P^2 = P \times P$ (standard matrix product), which is the transition matrix of the two-step random walk, is also reversible and has the same stationary distribution and eigenvectors as P. Moreover, the eigenvalues of P^2 are the squares of the eigenvalues of P. In particular, $\lambda(P^2) = (\lambda(P))^2$. Given $A, B \subseteq V$ and $x \in V$,

we define $P(x, A) = \sum_{y \in A} P(x, y)$ and the *flow function* $Q(A, B)$ from A to B as

$$Q(A, B) = \sum_{x \in A} \pi(x) P(x, B). \tag{2}$$

$Q(A, B)$ is the probability a random walk starting from the stationary distribution makes a transition from a vertex in A to a vertex in B. Due to reversibility of P, $Q(A, B) = Q(B, A)$.

We use the following standard inequalities, often referred to as the *Expander Mixing Lemma for Inhomogeneous Graphs*. Let $A, B \subseteq V$, and $A^c = V \setminus A$, then

$$|Q(A, A^c) - \pi(A)\pi(A^c)| \leq \lambda \pi(A)\pi(A^c), \tag{3}$$

$$|Q(A, B) - \pi(A)\pi(B)| \leq \lambda \sqrt{\pi(A)\pi(B)\pi(A^c)\pi(B^c)}. \tag{4}$$

We also need the following lower bounds for Q^2, which can be proven using (4).

▶ **Lemma 4.** *For any $A, B \subseteq V$, we have*

$$Q(A, B)^2 \geq (\pi(A)\pi(B))^2 - 2\lambda(\pi(A)\pi(B))^{3/2}(\pi(A^c)\pi(B^c))^{1/2}. \tag{5}$$

In two-sample voting, the probability vertex x adopts the opinion B in one step is $P(x, B)^2 = (d_B(x)/d(x))^2$. Given $A, B \subseteq V$, define the quantity

$$R(A, B) = \sum_{x \in A} \pi(x)(P(x, B))^2.$$

$R(A, B)$ is the stationary measure π resulting from vertices in set A choosing opinion B in one round of two-sample voting. We do not require here A and B to be disjoint, so $A = B$ is possible. The next lemma shows that there is a connection between two-sample voting and two-step random walks. While two-sample voting does not refer to random walks in any explicit way, the transition matrix P of a random walk appears in the analysis of this voting process because of Lemma 5.

▶ **Lemma 5.** *For any $A \subseteq V$, we have $R(V, A) = Q_2(A, A)$, where Q_2 is the flow function for the two-step transition matrix P^2.*

Proof. From definition of $R(V, A)$, reversibility of P and $P^2(x, y) = \sum_{z \in V} P(x, z)P(z, y)$:

$$\begin{aligned} R(V, A) &= \sum_{x \in V} \pi(x) P(x, A)^2 = \sum_{x \in V} \pi(x) P(x, A) \sum_{y \in A} P(x, y) \\ &= \sum_{y \in A} \pi(y) \sum_{x \in V} P(y, x) P(x, A) = \sum_{y \in A} \pi(y) P^2(y, A) = Q_2(A, A). \end{aligned}$$ ◀

If G is a complete graph (with a loop at each vertex), then $R(V, A) = \pi(A)^2 = (|A|/n)^2$ and $R(A, B) = \pi(A)\pi(B)^2 = |A| \cdot |B|^2/n^3$. The next two lemmas give bounds on deviations from these values in regular graphs.

▶ **Lemma 6.** *For $A \subseteq V$, we have*

$$|R(V, A) - \pi(A)^2| = |Q_2(A, A^c) - \pi(A)\pi(A^c)| \leq \lambda^2 \pi(A)\pi(A^c). \tag{6}$$

Proof. By Lemma 5, $R(V, A) = Q_2(A, A)$. Also $Q_2(A, A) = Q_2(A, V) - Q_2(A, A^c) = \pi(A) - Q_2(A, A^c)$, so

$$R(V, A) - \pi(A)^2 = \pi(A) - Q_2(A, A^c) - \pi(A)^2 = \pi(A)\pi(A^c) - Q_2(A, A^c).$$

Taking the absolute value of both sides gives the first equality in (6). To obtain the inequality, apply (3) to P^2, Q_2 and λ^2 as the second largest absolute eigenvalue of P^2. ◀

▶ **Lemma 7.** *Let $A, B \subseteq V$, then*

$$R(A,B) \;\geq\; \frac{Q(A,B)^2}{\pi(A)} \geq \pi(A)\pi(B)^2 - 2\lambda\pi(A)^{1/2}\pi(B)^{3/2}\pi(A^c)^{1/2}\pi(B^c)^{1/2}.$$

Proof. The second inequality is from Lemma 4. From convexity of the function $z \mapsto z^2$,

$$R(A,B) = \pi(A)\sum_{x\in A}\frac{\pi(x)}{\pi(A)}(P(x,B))^2 \geq \pi(A)\left(\sum_{x\in A}\frac{\pi(x)}{\pi(A)}P(x,B)\right)^2 = \frac{1}{\pi(A)}Q(A,B)^2. \tag{7}$$

◀

Suppose the family of sets $\mathcal{C} = (A_1, \ldots, A_k)$ is a partitioning of V. Define the quantity $S_{\mathcal{C}}(A) = \sum_{i=1}^k R(A, A_i)$. For a complete graph, $S_{\mathcal{C}}(V) = \sum_{i=1}^k \pi(A_i)^2$ and the following lemma bounds the deviation from this value in regular graphs.

▶ **Lemma 8.** *Consider a partitioning $\mathcal{C} = (A_1, \ldots, A_k)$ of V. Then*

$$\left|S_{\mathcal{C}}(V) - \sum_{i=1}^k \pi(A_i)^2\right| \leq \lambda^2\left(1 - \sum_{i=1}^k \pi(A_i)^2\right).$$

Proof. Using Lemma 6, we get

$$\left|S_{\mathcal{C}}(V) - \sum_{i=1}^k \pi(A_i)^2\right| \;\leq\; \sum_{i=1}^k \left|R(V,A_i) - \pi(A_i)^2\right| \;\leq\; \lambda^2\left(1 - \sum_{i=1}^k \pi(A_i)^2\right).$$

◀

▶ **Lemma 9.** *Let $\mathcal{C} = (A_1, \ldots, A_k)$ be a partitioning of V. For any $A \subseteq V$,*

$$S_{\mathcal{C}}(A) \;\geq\; \pi(A)\sum_{i=1}^k \pi(A_i)^2 - 2\lambda\pi(A)^{1/2}\sum_{i=1}^k \pi(A_i)^{3/2}, \tag{8}$$

$$S_{\mathcal{C}}(A) \;\leq\; \pi(A)\sum_{i=1}^k \pi(A_i)^2 + 2\lambda\pi(A)^{1/2}\sum_{i=1}^k \pi(A_i)^{3/2} + \lambda^2. \tag{9}$$

Proof. Lemma 7 gives:

$$S_{\mathcal{C}}(A) \;=\; \sum_{i=1}^k R(A,A_i) \geq \pi(A)\sum_{i=1}^k \pi(A_i)^2 - 2\lambda\pi(A)^{1/2}\pi(A^c)^{1/2}\sum_{i=1}^k \pi(A_i)^{3/2}, \tag{10}$$

and Inequality (8) follows. To show Inequality (9), observe that $S_{\mathcal{C}}(A) + S_{\mathcal{C}}(A^c) = S_{\mathcal{C}}(V)$ and use Lemma 8 and (10) applied to A^c

$$\begin{aligned} S_{\mathcal{C}}(A) &= S_{\mathcal{C}}(V) - S_{\mathcal{C}}(A^c) \\ &\leq \pi(A)\sum_{i=1}^k \pi(A_i)^2 + 2\lambda\pi(A)^{1/2}\pi(A^c)^{1/2}\sum_{i=1}^k \pi(A_i)^{3/2} + \lambda^2\left(1 - \sum_{i=1}^k \pi(A_i)^2\right). \end{aligned}$$

◀

4 Proof of Theorem 1

From now on we assume the graph is d-regular, so $\pi(x) = 1/n$, and for $A \subseteq V$, $\pi(A) = |A|/n$. Furthermore, $nR(A,B) = \sum_{x\in A}(d_B(x)/d)^2$ is the expected number of vertices in A which pick two opinions in B. When clear from the context, we use A instead of $|A|$.

Let A_j be the set of vertices with opinion j. At any step, the opinions are ordered according to their sizes: $A_1 \geq A_2 \geq \ldots \geq A_k$. Thus $\mathcal{C} = (A_1, \ldots, A_k)$ is a partition of V.

Let A'_j be the set of vertices with opinion j after one round. We have the following equality, where the second term in (11) is the expected weight of vertices changing their opinion to A_j and the third term is the expected weight of vertices changing their opinion from A_j (using the measure π as the weight of a set of vertices).

$$\begin{aligned}\mathbf{E}(\pi(A'_j)|\mathcal{C}) &= \pi(A_j) + R(V \setminus A_j, A_j) - \sum_{i\neq j} R(A_j, A_i) &(11)\\ &= \pi(A_j) + R(V, A_j) - R(A_j, A_j) - \sum_{i\neq j} R(A_j, A_i)\\ &= \pi(A_j) + R(V, A_j) - S_{\mathcal{C}}(A_j). &(12)\end{aligned}$$

The next lemma shows that, given a sufficient advantage of opinion 1, after one round of voting opinion 1 remains the largest opinion. More precisely, the lemma gives lower bounds on the increase of the size of opinion 1 and on the increase of the advantage of this opinion over the other opinions.

▶ **Lemma 10.** *Assume* $A_1 \leq 2n/3$, $A_1 - A_2 \geq Cn\sqrt{(\log n)/A_1}$ *(requiring* $A_1 \geq C^{2/3}n^{2/3}\log^{1/3} n$*), where* $C = 240\sqrt{2}$*, and* $\lambda \leq (A_1 - A_2)/(32n)$*. Then with probability at least* $1 - 1/n^2$*,*

$$A'_1 \geq A_1\left(1 + \frac{A_1 - A_2}{5n}\right). \quad (13)$$

$$\min_{2\leq j\leq k}\left\{A'_1 - A'_j\right\} \geq (A_1 - A_2)\left(1 + \frac{A_1}{10n}\right), \quad (14)$$

Proof. Several times in this proof we use that $\pi(A_1) \leq 2/3$, which implies that $\pi(A_1^c) \geq 1/3$. Our proof uses the following Chernoff bounds. If X is the sum of independent Bernoulli random variables, then for $\varepsilon \in (0,1)$ and $\delta \geq 1$,

$$\mathbf{Pr}(X \geq (1+\varepsilon)\mathbf{E}(X)),\ \mathbf{Pr}(X \leq (1-\varepsilon)\mathbf{E}(X)) \leq \exp(-\varepsilon^2\mathbf{E}(X)/3), \quad (15)$$

$$\mathbf{Pr}(X \geq (1+\delta)\mathbf{E}(X)) \leq \exp(-\delta\mathbf{E}(X)/3). \quad (16)$$

From Equation (12) and Lemmas 6 and 9, we have the following lower and upper bounds on $\mathbf{E}(\pi(A'_j)|\mathcal{C})$ for any $j \in [k]$.

$$\begin{aligned}\mathbf{E}(\pi(A'_j)|\mathcal{C}) &= \pi(A_j) + R(V, A_j) - S_{\mathcal{C}}(A_j)\\ &\geq \pi(A_j) + \pi(A_j)^2 - \lambda^2\pi(A_j)\pi(A_j^c)\\ &\quad -\pi(A_j)\sum_{i=1}^{k}\pi(A_i)^2 - 2\lambda\pi(A_j)^{1/2}\sum_{i=1}^{k}\pi(A_i)^{3/2} - \lambda^2\\ &\geq \pi(A_j)\left(1 + \pi(A_j) - \sum_{i=1}^{k}\pi(A_i)^2\right) - 2\lambda\pi(A_j)^{1/2}\pi(A_1)^{1/2} - (5/4)\lambda^2. &(17)\end{aligned}$$

$$\begin{aligned}\mathbf{E}(\pi(A'_j)|\mathcal{C}) &= \pi(A_j) + R(V, A_j) - S_{\mathcal{C}}(A_j)\\ &\leq \pi(A_j) + \pi(A_j)^2 + \lambda^2\pi(A_j)\pi(A_j^c) - \pi(A_j)\sum_{i=1}^{k}\pi(A_i)^2 + 2\lambda\pi(A_j)^{1/2}\sum_{i=1}^{k}\pi(A_i)^{3/2}\\ &\leq \pi(A_j)\left(1 + \pi(A_j) - \sum_{i=1}^{k}\pi(A_i)^2\right) + (1/4)\lambda^2 + 2\lambda\pi(A_j)^{1/2}\pi(A_1)^{1/2}. &(18)\end{aligned}$$

We have $\lambda \leq \pi(A_1)/32$ and $\pi(A_1) \leq 2/3$, by assumption, so (17) and (18) imply

$$\pi(A_1)/2 \;\leq\; \mathbf{E}(\pi(A_1')|\mathcal{C}) \;\leq\; 2\pi(A_1). \tag{19}$$

Define $\varepsilon_1 = \sqrt{\frac{9\log n}{\mathbf{E}(A_1'|\mathcal{C})}} \leq \sqrt{\frac{18\log n}{A_1}} < 1$. Using the Chernoff bounds (15), we get

$$\mathbf{Pr}(A_1' \leq (1-\varepsilon_1)\mathbf{E}(A_1'|\mathcal{C})|\mathcal{C}) \leq e^{-3\log(n)} = n^{-3}. \tag{20}$$

For a fixed j, $2 \leq j \leq k$, define $\varepsilon_j = \sqrt{9(\log n)\mathbf{E}(A_1'|\mathcal{C})}/\mathbf{E}(A_j'|\mathcal{C})$ and show the following bound, using (15), for $\varepsilon_j \leq 1$, and (16), for $\varepsilon_j \leq 1$.

$$\mathbf{Pr}(A_j' \geq (1+\varepsilon_j)\mathbf{E}(A_j'|\mathcal{C})|\mathcal{C}) \leq n^{-3}. \tag{21}$$

The bounds (20) and (21) imply that with probability at least $1-n^{-2}$, for all $2 \leq j \leq k$,

$$\begin{aligned} A_1' - A_j' &\geq (1-\varepsilon_1)\mathbf{E}(A_1'|\mathcal{C}) - (1+\varepsilon_j)\mathbf{E}(A_j'|\mathcal{C}) \\ &= \mathbf{E}(A_1' - A_j'|\mathcal{C}) - 2\sqrt{9(\log n)\mathbf{E}(A_1'|\mathcal{C})}, \end{aligned} \tag{22}$$

and thus

$$\pi(A_1') - \pi(A_j') \;\geq\; \mathbf{E}(\pi(A_1') - \pi(A_j')|\mathcal{C}) - 2\sqrt{\frac{9(\log n)\mathbf{E}(\pi(A_1')|\mathcal{C})}{n}}. \tag{23}$$

The right-hand side of (18) is non-increasing with increasing j, so for each $2 \leq j \leq k$,

$$\mathbf{E}(\pi(A_j')|\mathcal{C}) \;\leq\; \pi(A_2)\left(1 + \pi(A_2) - \sum_{i=1}^{k}\pi(A_i)^2\right) + (1/4)\lambda^2 + 2\lambda\pi(A_1). \tag{24}$$

Let $\Delta = \pi(A_1) - \pi(A_2)$. Inequalities (17) and (24) give for each $2 \leq j \leq k$,

$$\begin{aligned} \mathbf{E}(\pi(A_1') - \pi(A_j')|\mathcal{C}) &\geq \Delta\left(1 + \pi(A_1) + \pi(A_2) - \sum_{i=1}^{k}\pi(A_i)^2\right) - 4\lambda\pi(A_1) - (3/2)\lambda^2 \\ &\geq \Delta(1 + \pi(A_1)\pi(A_1^c)) - 4\lambda\pi(A_1) - 2\lambda^2 \qquad (25) \\ &\geq \Delta + \Delta\pi(A_1)/7. \qquad (26) \end{aligned}$$

Inequality (25) holds because $\sum_{i=2}^{k}\pi(A_i)^2 \leq \pi(A_2)$. In the last step we used that $\pi(A_1^c) \geq 1/3$ and $\lambda \leq \Delta/32$. From (23), (26) and (19), with probability at least $1-n^{-2}$,

$$\begin{aligned} \min_{2\leq j\leq k}\{\pi(A_1') - \pi(A_j')\} &\geq \mathbf{E}(\pi(A_1') - \pi(A_j')|\mathcal{C}) - \frac{\varepsilon_1}{n}\mathbf{E}(A_1'|\mathcal{C}) - \frac{\varepsilon_j}{n}\mathbf{E}(A_j'|\mathcal{C}) \\ &\geq \Delta\left(1 + \pi(A_1)/7 - \frac{6}{\Delta}\sqrt{\frac{2\log n}{n}\pi(A_1).}\right). \end{aligned}$$

By assumption, $\Delta \geq 240\sqrt{2\log(n)/A_1}$, so with probability at least $1-n^{-2}$,

$$\min_{2\leq j\leq k}\{\pi(A_1') - \pi(A_j')\} \;\geq\; \Delta(1 + \pi(A_1)/10), \tag{27}$$

and we get we get (14). This also proves that w.h.p. opinion 1 remains the majority opinion. The order between the other opinions might change.

To get information about the increase in the number of vertices with opinion 1, we use Equation (17) with $j = 1$ and the assumption that $\lambda \leq \Delta/32$. We obtain

$$\begin{aligned}\mathbf{E}(\pi(A_1')|\mathcal{C}) &\geq \pi(A_1)(1+\pi(A_1) - \sum_{i=1}^{k}\pi(A_i)^2) - \Delta\pi(A_1)/16 - \Delta^2/(32)^2 \\ &\geq \pi(A_1)(1+\pi(A_1) - \pi(A_1)^2 - \pi(A_2)\pi(A_1^c) - \Delta/16 - \Delta/(32)^2) \\ &> \pi(A_1)(1+\Delta/4). \end{aligned} \tag{28}$$

By using Chernoff bounds (15) with $\varepsilon = \sqrt{\frac{9\log n}{\mathbf{E}(A_1'|\mathcal{C})}}$ and Inequalities (28) and (19), with probability at least $1 - n^{-2}$,

$$\begin{aligned}A_1' &\geq A_1(1+\Delta/4) - \sqrt{\mathbf{E}(A_1'|\mathcal{C})9\log n} \;\geq\; A_1(1+\Delta/4) - \sqrt{18A_1\log n} \\ &= A_1(1+\Delta/4 - 3\sqrt{2}\sqrt{\log n/A_1})). \end{aligned} \tag{29}$$

From the assumptions of the lemma, we have $\Delta/20 = (A_1 - A_2)/(20n) \geq 3\sqrt{2}\sqrt{\log n/A_1}$. Therefore (29) implies $A_1' \geq A_1(1+\Delta/5)$, which is the same as (13). ◀

▶ **Lemma 11.** *Assume $A_1 \leq 2n/3$, $A_1 - A_2 \geq Cn\sqrt{(\log n)/A_1}$ and $\lambda \leq (A_1 - A_2)/(32n)$. With probability at least $1 - 1/n$, after at most $O((n/A_1)\log(A_1/(A_1 - A_2)))$ rounds, the number of vertices with opinion 1 is at least $2n/3$.*

Proof. We apply Lemma 10 to consecutive rounds until the size of opinion 1 reaches $2n/3$. Since w.h.p. the difference between the size of opinion 1 and the size of the second largest opinion increases, our assumption about λ in Lemma 10 is maintained from round to round. If the ordering of the opinions according to size changes at any step t, we relabel the opinions so that $A_1(t) \geq A_2(t) \cdots \geq A_k(t)$. Lemma 10 implies that w.h.p. opinion 1 remains the largest opinion, and thus never relabeled.

Denote by $x(i)$ the fraction of vertices with opinion 1 at the end of round i, and by $y(i)$ the difference between the fraction of vertices with opinion 1 and the fraction of vertices with the second largest opinion. Thus $x(0) = \pi(A_1)$, and $y(0) = \Delta = \pi(A_1) - \pi(A_2) < x(0)$. By (13) and (14) and induction on the number of rounds, with probability at least $1 - 1/n$, for each round $1 \leq i \leq n$, if $x(i) < 2/3$, then

$$x(i) \geq x(i-1)(1 + y(i-1)/5), \tag{30}$$

$$y(i) \geq y(i-1)(1 + x(i-1)/10). \tag{31}$$

Iterating (30) and (31) for $j = \lceil 10/x(0)\rceil < n$ rounds, we get $y(j) \geq 2y(0)$ and $x(j) \geq x(0) + y(0)$, or $x(i) \geq 2/3$ for some $i \leq j$. Repeating this $r = \lceil \log_2(x(0)/y(0))\rceil$ times, we get for round $i_1 = rj < n$, $y(i_1) \geq x(0)$ and $x(i_1) \geq x(0) + y(0) + 2y(0) + 4y(0) \cdots + 2^{r-1}y(0) \geq 2x(0)$, or $x(i) \geq 2/3$ for some $i \leq i_1$.

If for some $q \geq 1$, $y(i_q) \geq 2^{q-1}x(0)$ and $x(i_q) \geq 2^q x(0)$, or $x(i) \geq 2/3$ for some $i \leq i_q$, then at the end of round $i_{q+1} = i_q + \lceil 10/(2^q x(0))\rceil$, either $y(i_{q+1}) \geq 2^q x(0)$ and $x(i_{q+1}) \geq 2^{q+1}x(0)$, or $x(i) \geq 2/3$ for some $i \leq i_{q+1}$, or $i_{q+1} > n$. Taking $q = \lceil \log_2(1/x(0))\rceil$, we have $i_q = O((1/x(0))\log(x(0)/y(0))) = O((n/A_1)\log(A_1/(A_1 - A_2)))$ (observe that $i_q < n$) and $2^q x(0) \geq 1$, so we must have $x(i) \geq 2/3$ for some $i \leq x(i_q)$. ◀

When the largest opinion reaches the size $2n/3$, it will take over the whole graph within additional $O(\log n)$ rounds. The progress of voting in this final stage would be slowest, if all minority opinions were joined together into a single "second" opinion. The next lemma can

be proven in the similar way as it is proven in [14] that two-sample voting finishes in $O(\log n)$ rounds, if there are two opinions, the majority opinion has size at least cn, for a constant $c > 1/2$, and λ is sufficiently small. Theorem 1 follows immediately from Lemmas 11 and 12.

▶ **Lemma 12.** *Let G be a connected regular graph with $\lambda \leq 1/4$. If the majority opinion has size at least $2n/3$, then with probability at least $1 - n^{-2}$, the voting finishes within $\mathcal{O}(\log n)$ rounds.*

5 Reducing Three-sample voting to Two-sample voting

In the three-sample voting process considered in [4], each vertex v samples in each round three random neighbours with replacement, collecting their opinions, say, $Y_{v,1}, Y_{v,2}, Y_{v,3}$. Vertex v changes its opinion to the majority of $\{Y_{v,1}, Y_{v,2}, Y_{v,3}\}$, or, if there is no majority, to $Y_{v,1}$. We show that our analysis of two sampling applies, with small modifications, also to three sampling, giving the same bounds as in Theorem 1. The crucial idea is to view the three sampling process in the following equivalent way.

Suppose in a given round we have k opinions, let $\mathcal{C} = (A_1, A_2, \ldots, A_n)$ be the partition of the vertices into the opinion classes and let A'_j be the vertices with opinion j at the next round. Each verstex v decides on its opinion for the next round in the following way. First, v takes on the opinion $Y_{v,1}$. Let A''_j, $1 \leq j \leq k$, be the opinion classes after this initial updates. Now v obtains its final opinion by the two-sampling decision using opinions $Y_{v,2}, Y_{v,3}$ (taken in the original partition $\mathcal{C}$). Observe that classes A''_j result from choosing only one (random) vertex, that is, they are obtained by one round of the standard (single-sample and slow) pull voting. While the sizes of opinions may change in one round of the standard pull voting, the expected sizes are equal to the initial sizes; see, for example [22]. That is, we have $\mathbf{E}(\pi(A''_j|\mathcal{C})) = \pi(A_j)$.

The following lemma implies that the analysis of two-sample voting can be updated to three-sample voting by putting $\mathbf{E}(S_{\mathcal{C}}(A''_j)|\mathcal{C})$ in place of $S_{\mathcal{C}}(A_j)$ and using the bounds (33) and (34) on $\mathbf{E}(S_{\mathcal{C}}(A''_j)|\mathcal{C})$, which are exactly the same as the bounds on $S_{\mathcal{C}}(A_j)$ in Lemma 9. The proof of the lemma and further details will be included in the full version of the paper.

▶ **Lemma 13.** *Let G be a connected graph and let $\mathcal{C} = (A_1, \ldots, A_k)$ partition V. Then*

$$\mathbf{E}(\pi(A'_j)|\mathcal{C}) = \pi(A_j) + R(V, A_j) - \mathbf{E}(S_{\mathcal{C}}(A''_j)|\mathcal{C}), \tag{32}$$

$$\mathbf{E}(S_{\mathcal{C}}(A''_j)|\mathcal{C}) \geq \pi(A_j)\sum_{i=1}^{k}\pi(A_i)^2 - 2\lambda\pi(A_j)^{1/2}\sum_{i=1}^{k}\pi(A_i)^{3/2}, \tag{33}$$

$$\mathbf{E}(S_{\mathcal{C}}(A''_j)|\mathcal{C}) \leq \pi(A_j)\sum_{i=1}^{k}\pi(A_i)^2 + 2\lambda\pi(A_j)^{1/2}\sum_{i=1}^{k}\pi(A_i)^{3/2} + \lambda^2. \tag{34}$$

If $\mathcal{C} = (A, B)$, then $E(S_{\mathcal{C}}(B'')|\mathcal{C}) \geq \pi(B)/2$.

6 Algorithm Pushy and Proof of Theorem 3

We describe a push-voting algorithm for reaching consensus on a graph G.

Algorithm Pushy
Repeat L times
begin
Each vertex activates with probability p and sends its opinion to all neighbours

Each vertex keeps a tally of how many opinions of each type it has received
Let $M(v)$ be the set of opinions received by v which have the maximal count
If $|M(v)| \geq 1$, vertex v picks an opinion u.a.r. from $M(v)$
If $|M(v)| = 0$ (i.e. v has not received any opinion), v keeps its current opinion
end

The following lemma rephrases Theorem 3.

▶ **Lemma 14.** *Let $G = K_n$ and $p = \alpha \log n/n$, for a sufficiently large constant $\alpha > 0$. Then with probability $1 - o(1)$, after $L = 4$ rounds there is a unique remaining opinion, irrespective of the initial number and distribution of opinions.*

Proof. On the complete graph K_n, for each round, the sets $M(v) \equiv M$ are the same for all vertices. Consider one round and let $m = |M|$. The expected number of activated vertices is $\alpha \log n$ and the actual number is concentrated around this value (see (15)), so w.h.p. at least one but fewer than $\beta \log n$ vertices activate in this round, where $\beta = \beta(\alpha)$ is a constant sufficiently larger than α. Thus w.h.p. $1 \leq m < \beta \log n$. If n vertices pick u.a.r. from the set M, then w.h.p. each opinion i is chosen $N_i = (1 + o(1))n/m$ times (use again (15)). Thus w.h.p. the number of opinions in the next round is $k = m < \beta \log n$.

Consider now a round $j \geq 2$ with the number of opinions $k \geq 2$ (no consensus yet). We have w.h.p. $2 \leq k < \beta \log n$ and each opinion i is represented by $N_i = (1 + o(1))n/k$ vertices. We will upper bound m, which is the number of opinions with the maximal number of activations, giving the number of opinions for the next round. The number of activations of opinion i is a binomial random variable $X_i \sim Bin(N_i, p)$ independent of any other opinion. Henceforth we use X and $N = (1 + o(1))n/k$ to denote X_i and N_i. We use the inequalities for binomial probabilities given below, which hold as follows: (35) for any h, (36) for $h > Np$, (37) for $h > Np$, $h = o(\sqrt{N})$, $hp = o(1)$ and $\theta = (1 + O(1/h) + O(h^2/N))/\sqrt{2\pi}$.

$$\mathbf{Pr}(X = h) \leq 1/\sqrt{Np(1-p)}, \tag{35}$$

$$\mathbf{Pr}(X = h) \leq \mathbf{Pr}(X \geq h) \leq \mathbf{Pr}(X = h) \cdot \left(1 - \frac{N-h}{h+1}\frac{p}{1-p}\right)^{-1}, \tag{36}$$

$$\mathbf{Pr}(X = h) = \binom{N}{h} p^h (1-p)^{N-h} = \frac{\theta}{\sqrt{h}} \left(\frac{eNp}{h}\right)^h e^{-Np}. \tag{37}$$

We consider the following four cases, where $\omega = (\log\log n)/\log\log\log n$.

C1: $(\beta \log n)/\omega \leq k < \beta \log n$,

C2: $(\log n)/\log\log n \leq k < (\beta \log n)/\omega$,

C3: $\beta(\log n)^{1/6} \leq k < (\log n)/\log\log n$,

C4: $2 \leq k < \beta(\log n)^{1/6}$.

Case C1: $(\beta \log n)/\omega \leq k < \beta \log n$. Let $h = (11/12)\omega = (11/12)(\log\log n)/\log\log\log n$. Referring to (36) and recalling that β is sufficiently larger than α,

$$\frac{N-h}{h+1}\frac{p}{1-p} \leq (1 + o(1))\frac{Np}{h} \leq (1 + o(1))\frac{\alpha}{(11/12)\beta} < 1/2.$$

Thus from (36), $\mathbf{Pr}(X \geq h) \leq 2 \cdot \mathbf{Pr}(X = h)$.

Let $N = (1+o(1))n/k$, $p = f/n$ where $f = \alpha \log n$. Let $k = (\log n)/\sigma$ where $1/\beta \le \sigma \le \omega/\beta$. Using (37),

$$\begin{aligned}
\mathbf{Pr}(X=h) = \; & \frac{\theta}{\sqrt{h}}\left(\frac{ne}{kh}p\right)^h e^{-np/k} \quad = \quad \frac{\theta}{\sqrt{h}}\left(\frac{ef}{hk}\right)^h e^{-f/k} \quad = \quad \frac{\theta}{\sqrt{h}}\left(\frac{e\alpha\sigma}{h}\right)^h e^{-\alpha\sigma} \\
= \; & \theta \exp\left(-\alpha\sigma - h\log\left(\frac{h}{e\alpha\sigma}\right) - (1/2)\log h\right) \\
= \; & \theta \exp\left(-h\log h\left(1 + \frac{\alpha\sigma}{h\log h} + \frac{1}{2h} - \frac{\log(e\alpha\sigma)}{h\log h}\right)\right) \\
= \; & \exp\left(-h\log h\,(1+o(1))\right) \\
= \; & (\log n)^{-(11/12)(1+o(1))} \quad = \quad \rho.
\end{aligned}$$

The probability that no opinion is represented at least h times is $(1-\rho)^k = o(1)$. The expected number of opinions represented at least h times is $k\mathbf{Pr}(X \ge h)$, and

$$k\mathbf{Pr}(X \ge h) \;\le\; 2k\mathbf{Pr}(X = h) \;=\; O(\log n)^{(1/12)(1+o(1))} = o((\log n)^{1/6}).$$

Thus with probability $1-o(1)$, there is an opinion activated at least h times and the number m of opinions with the maximal number of activations (that is, the number of opinions for the next round) is less than $\beta(\log n)^{1/6}$. That is, in the next round we will have Case C4 or $k = 1$.

Case C2: $(\log n)/\log\log n \le k < (\beta \log n)/\omega$. For any h, using (35),

$$\mathbf{Pr}(X = h) \le \frac{1}{\sqrt{np/k}} = \frac{1}{\sqrt{\omega\alpha/\beta}}.$$

Let $j = k\gamma/\sqrt{\omega}$, where $\gamma \ge 3e\sqrt{B/A}$. The probability that more than j opinions are transmitted the same (maximum) number of times is at most

$$\binom{k}{j}\left(\frac{1}{\sqrt{\omega\alpha/\beta}}\right)^{j-1} \le O(\sqrt{\omega})\left(\frac{ke}{j}\right)^j\left(\frac{1}{\sqrt{\omega\alpha/\beta}}\right)^j = O(\sqrt{\omega})\,e^{-j} = o(1).$$

Thus the number of opinions in the next round will be $j = o(\log n/\log\log n)$, with probability $1-o(1)$. That is, in the next round we will have Case C3 or Case C4 or $k = 1$.

Case C3: $\beta(\log n)^{1/6} \le k < (\log n)/(\log\log n)$. For $Np \ge \alpha \log\log n$, using (36) and (37),

$$\mathbf{Pr}(X \ge eNp) \le \frac{O(1)}{\sqrt{Np}}e^{-Np} = \frac{o(1)}{(\log n)^\alpha},$$

so with probability $1-o(1)$ all opinions are transmitted at most eNp times. For $h = DNp$, $D > 1$, by direct calculation

$$\frac{\mathbf{Pr}(X = h+1)}{\mathbf{Pr}(X = h)} = \frac{p}{1-p}\frac{N-h}{h+1} = (1+o(1))\frac{1}{D}.$$

As $\mathbf{Pr}(X \ge \mathbf{E}(X)) = \gamma$ for some $\gamma > 0$ constant, it follows that with probability $1-o(1)$ there exists $1 < D < e$ such that and for $h = DNp$, the expected number of opinions activated at least h times is

$$k\mathbf{Pr}(X \ge h) = \Theta((\log n)^{1/7}).$$

Thus with probability $1-o(1)$, the number of opinions with the maximal number of activations is at most $\beta(\log n)^{1/7}$. That is, in the next round we will have Case C4 or $k = 1$.

Case C4: $2 \leq k < \beta(\log n)^{1/6}$. Using (35) with $Np = \Omega((\log n)^{5/6})$, we get

$$\mathbf{Pr}(X = h) = O\left(1/(\log n)^{5/12}\right).$$

Let X_i be the number of activated vertices of opinion $i = 1, \ldots, k$, then

$$\mathbf{Pr}(\exists (i,j)\ s.t.\ X_i = X_j) = O\left(\frac{k^2}{(\log n)^{5/12}}\right) = O\left(\frac{1}{(\log n)^{1/12}}\right) = o(1),$$

so, with probability $1 - o(1)$, a unique maximum opinion remains, that is, $k = 1$ in the next round.

The probability that no vertex activates in one round is $(1-p)^n = O(n^{-\alpha})$. Thus after 4 rounds (if Case C2, then no C3 or C4) a unique opinion remains with probability $1 - o(1)$. ◀

References

1 D. Aldous. Meeting times for independent Markov chains. *Stochastic Processes and their Applications* 38(2):185–193, (1991).

2 D. Aldous and J. Fill. *Reversible Markov Chains and Random Walks on Graphs*, `http://stat-www.berkeley.edu/pub/users/aldous/RWG/book.html`.

3 L. Becchetti, A. Clementi, E. Natale, F. Pasquale and R. Silvestri. Plurality consensus in the gossip model. *SODA 2015*, pages 371–390.

4 L. Becchetti, A. Clementi, E. Natale, F. Pasquale, R. Silvestri and L. Trevisan. Simple dynamics for plurality consensus. In *Proceedings of the 26th ACM symposium on Parallelism in Algorithms and Architectures (SPAA 2014)*, pages 247–256, (2014).

5 L. Becchetti, A.E.F. Clementi, E. Natale, F. Pasquale and L. Trevisan. Stabilizing Consensus with Many Opinions. *SODA 2016*, pages 620–635.

6 P. Berenbrink, A.E.F. Clementi, R. Elsässer, P. Kling, F. Mallmann-Trenn, E. Natale. Ignore or Comply?: On Breaking Symmetry in Consensus. *PODC 2017*, pages 335–344.

7 P. Berenbrink, Tom .Friedetzky, G. Giakkoupis, P. Kling. Efficient Plurality Consensus, Or: the Benefits of Cleaning up from Time to Time. *ICALP 2016*, pages 136:1-136:14.

8 P. Berenbrink, T. Friedetzky, P. Kling, F. Mallmann-Trenn and C. Wastell. Plurality Consensus in Arbitrary Graphs: Lessons Learned from Load Balancing. In *Proceedings of the 24th Annual European Symposium on Algorithms, (ESA 2016)*, pages 10:1–10:18, (2016).

9 S. Brahma, S. Macharla, S. P. Pal, S. R. Singh. Fair Leader Election by Randomized Voting. In *ICDCIT 2004*, pages 22–31, (2004).

10 A. Broder, A. Frieze, S. Suen and E. Upfal. Optimal construction of edge disjoint paths in random graphs. *SIAM Journal on Computing*, 28(2), pages 541–573, (1999).

11 C. Cooper, M.E. Dyer, A.M. Frieze, N. Rivera. Discordant Voting Processes on Finite Graphs. *ICALP 2016*, pages 145:1–145:13.

12 C. Cooper, R. Elsässer, H. Ono, T. Radzik. Coalescing Random Walks and Voting on Connected Graphs. *SIAM Journal of Discrete Math*, 27, pages 1748–1758, (2013).

13 C. Cooper, R. Elsässer and T. Radzik. The power of two choices in distributed voting, *ICALP 2014*, pages 435–446, (2014).

14 C. Cooper, R Elsässer, T. Radzik, N. Rivera and T. Shiraga. Fast consensus for voting on general expander graphs. In *DISC 2015 – 29th International Symposium on Distributed Computing*, Springer-Verlag LNCS 9363, pages 248–262. (2015).

15 J. T. Cox. Coalescing random walks and voter model consensus times on the torus in $\mathbb{Z}^d$. *The Annals of Probability* 17(4):1333–1366, (1989).

16 X. Deng and C. Papadimitriou. On the Complexity of Cooperative Solution Concepts. *Mathematics of Operations Research* 19, pages 257–266, (1994).

17 R. Elsässer, T. Friedetzky, D. Kaaser, F. Mallmann-Trenn and H. Trinker. *Rapid Asynchronous Plurality Consensus.* arXiv:1602.04667, February 2017.

18 J. Friedman. A proof of Alon's second eigenvalue conjecture. In *STOC 2003: Proc. 35th Annual ACM Symposium on Theory of Computing*, pages 720–724, (2003).

19 M. Ghaffari and J. Lengler. *Tight Analysis for the 3-Majority Consensus Dynamics.* arXiv:1705.05583, May 2017.

20 M. Ghaffari and M. 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, (2016).

21 D. Gifford. Weighted Voting for Replicated Data. In *SOSP 1979: Proceedings of the 7th ACM Symposium on Operating Systems Principles*, pages 150–162, (1979).

22 Y. Hassin and D. Peleg. Distributed probabilistic polling and applications to proportionate agreement. *Information & Computation*, 171, pages 248–268, (2001).

23 B. Johnson, *Design and Analysis of Fault Tolerant Digital Systems*, Addison-Wesley, (1989).

Meeting in a Polygon by Anonymous Oblivious Robots*

Giuseppe A. Di Luna[1], Paola Flocchini[2], Nicola Santoro[3], Giovanni Viglietta[4], and Masafumi Yamashita[5]

1 **University of Ottawa, Ottawa, Canada**
gdiluna@uottawa.ca
2 **University of Ottawa, Ottawa, Canada**
paola.flocchini@uottawa.ca
3 **Carleton University, Ottawa, Canada**
santoro@scs.carleton.ca
4 **University of Ottawa, Ottawa, Canada**
gvigliet@uottawa.ca
5 **Kyushu University, Fukuoka, Japan**
mak@inf.kyushu-u.ac.jp

Abstract

The *Meeting problem* for $k \geq 2$ searchers in a polygon P (possibly with holes) consists in making the searchers move within P, according to a distributed algorithm, in such a way that at least two of them eventually come to see each other, regardless of their initial positions. The polygon is initially unknown to the searchers, and its edges obstruct both movement and vision. Depending on the shape of P, we minimize the number of searchers k for which the Meeting problem is solvable. Specifically, if P has a rotational symmetry of order σ (where $\sigma = 1$ corresponds to no rotational symmetry), we prove that $k = \sigma + 1$ searchers are sufficient, and the bound is tight. Furthermore, we give an improved algorithm that optimally solves the Meeting problem with $k = 2$ searchers in all polygons whose barycenter is not in a hole (which includes the polygons with no holes). Our algorithms can be implemented in a variety of standard models of mobile robots operating in Look-Compute-Move cycles. For instance, if the searchers have memory but are anonymous, asynchronous, and have no agreement on a coordinate system or a notion of clockwise direction, then our algorithms work even if the initial memory contents of the searchers are arbitrary and possibly misleading. Moreover, oblivious searchers can execute our algorithms as well, encoding information by carefully positioning themselves within the polygon. This code is computable with basic arithmetic operations (provided that the coordinates of the polygon's vertices are algebraic real numbers in some global coordinate system), and each searcher can geometrically construct its own destination point at each cycle using only a compass. We stress that such memoryless searchers may be located anywhere in the polygon when the execution begins, and hence the information they initially encode is arbitrary. Our algorithms use a self-stabilizing map construction subroutine which is of independent interest.

1998 ACM Subject Classification I.2.11 Distributed Artificial Intelligence – multiagent systems

Keywords and phrases Meeting problem, Oblivious robots, Polygon, Self-stabilization

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.14

* Full version available at https://arxiv.org/abs/1705.00324.

© Giuseppe A. Di Luna, Paola Flocchini, Nicola Santoro, Giovanni Viglietta, and Masafumi Yamashita;
licensed under Creative Commons License CC-BY
31st International Symposium on Distributed Computing (DISC 2017).
Editor: Andréa W. Richa; Article No. 14; pp. 14:1–14:15

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

1 Introduction

1.1 Framework

Consider a set of $k \geq 2$ autonomous mobile robots, modeled as geometric points located in a polygonal enclosure P, which may contain holes. The boundary of P limits both visibility and mobility, in that robots cannot move or see through the edges of P. Each robot observes the visible portion of P (taking an instantaneous snapshot of it), executes an algorithm to compute a visible destination point, and then moves to that point. Such a *Look-Compute-Move cycle* is repeated forever by every robot, each time taking a new snapshot and moving to a newly computed point. In this paper we study the *Meeting problem*, which prescribes the k robots to move in such a way that eventually at least two of them come to see each other and become "mutually aware". We will refer to these robots as *P-searchers*, or simply *searchers*.

Our searchers are severely limited, which makes the Meeting problem harder to solve. They do not know the shape of P in advance, nor their whereabouts within P. They are anonymous, implying that they all execute the same algorithm to determine their destination points. They are oblivious, meaning that each destination point is computed based only on the last snapshot taken, while older snapshots are forgotten, and no memory is retained between cycles. They are deterministic, meaning that they cannot resort to randomness in their computations. They are asynchronous, in the sense that we make no assumptions on how fast each searcher completes a Look-Compute-Move cycle compared to the others (these parameters are dynamic and are controlled by an adversarial *scheduler*). They are disoriented, which means that they have no magnetic compasses, GPS devices, or agreements of any kind. Each searcher has its own independent local orientation, unit of length, and handedness. They are silent, in that they cannot communicate with one another in any way. They have arbitrary initial locations within P.

In real-life applications, being in line of sight may allow robots to communicate in environments where non-optical means of communication are unavailable or impractical [22]. Solving the Meeting problem is also a necessary preliminary step to more complex tasks, such as space coverage [23] or the extensively studied *Gathering problem*, where all k robots have to physically reach the same point and stop there. In the special case of $k = 2$ robots, the Gathering problem is also called *Rendezvous problem*. Clearly, the terminating condition of the Meeting problem is more relaxed than that of Gathering; hence, any solution to the Gathering problem would also solve Meeting. Unfortunately, no solution to the Gathering problem in the setting considered here exists in the literature (see Section 1.4), and to the best of our knowledge there are no previous results on the Meeting problem.

In fact, given our searchers' many handicaps, and especially their lack of memory and orientation, it is hard to see how they could solve any non-trivial problem at all. Nonetheless, in this paper we will present the surprising result that the Meeting problem is solvable in almost every polygon, even for $k = 2$ searchers. Moreover, with the addition of a simple synchronization phase, our Meeting algorithms can be turned into Rendezvous algorithms, as we will discuss in Section 5.

1.2 Techniques

Since our searchers are disoriented and have no kind of *a-priori* agreement, they must use the geometric features of P to implicitly agree on some "landmarks" which can help them in their task. In order to identify such landmarks, each searcher has to visit P and construct a

map of it. But this cannot be done straightforwardly, because searchers are oblivious, and they forget everything as soon as they move. To cope with this handicap, they carefully move within P in such a way as to implicitly encode information as their distance from the closest vertex.

This positional encoding technique poses some obvious difficulties. First, it greatly limits the freedom of the searchers: they have to do precise movements to encode the correct information, and still manage to visit all of P and update the map as they go. Second, since searchers can be located anywhere in P when the execution starts, they could be implicitly encoding anything. This includes misleading information, such as a false map of P that happens to be locally coherent with the surroundings of the searcher. Therefore, a searcher can never rely on the information it is implicitly encoding, but it must constantly re-visit the entire polygon to make sure that the map it is encoding is correct.

Hence, searchers cannot simply agree on a landmark and sit on it waiting for one another, because that would prevent them from re-visiting P. This inconvenience drastically complicates the Meeting problem, and forces the searchers to follow relatively complicated movement patterns that make at least two of them necessarily meet.

There is also a subtle problem with the actual encoding of complex data as the distance from a point, which is a single real number. One could naively pack several real numbers into one by interleaving their digits, but this encoding would not be computable by real random-access machines [3]. Hence, we propose a more sophisticated technique, which only requires basic arithmetic operations. Such a technique can substitute the naive one under the reasonable assumption that the vertices of P be points with algebraic coordinates (as expressed in some global coordinate system, which is not necessarily the local one of any searcher).

1.3 Our Contributions

We prove that the Meeting problem in a polygon P can be solved by $k = \sigma + 1$ searchers, where σ is the order of the rotation group of P (which is also called the *symmetricity* of P). We also give a matching lower bound, showing that there are polygons of symmetricty σ where σ searchers cannot solve the Meeting problem.

Then, since all our lower-bound examples are polygons with a hole around the center, we wonder if the Meeting problem can be solved by fewer searchers if we exclude this small class of polygons. Surprisingly, it turns out that in all the remaining polygons only two searchers are sufficient to solve the Meeting problem. In particular, these include all the polygons with no holes.

Additionally, searchers can geometrically construct their destination points with a compass, provided that the vertices of P are algebraic points. Equivalently, searchers only have to compute combinations of basic arithmetic operations and square root extractions on the coordinates of the visible vertices of P. This is done via an encoding technique of independent interest, which we apply to mobile robots for the first time.

As a subroutine of our algorithms, we employ a self-stabilizing map construction algorithm that is of independent interest, as well.

In Section 2, we formally define all the elements of the Meeting problem. In Section 3, we consider the Meeting problem for searchers equipped with an unlimited amount of persistent internal memory whose initial contents can be arbitrary (hence also "incorrect"). This simplification allows us to present "cleaner" versions of our algorithms, which are not burdened by the technicalities of our positional encoding method. In Section 4, we present our encoding technique and we show how to apply it to the Meeting algorithms of Section 3, thus

extending our results to oblivious searchers. Finally, in Section 5 we discuss complementary results and directions for further research. Among other things, we briefly explain how to convert our Meeting algorithms into Rendezvous algorithms.

1.4 Related Work

The Gathering problem has been extensively studied in several contexts [1, 16]. The literature can be divided into works considering robots in a geometric space, and works considering agents on a graph [1, 11, 14]. The works on Gathering in the plane, which are more related to our setting, pertain to robots that inhabit an unbounded plane where no extraneous objects can block visibility or movement [2, 7, 12, 15, 18, 19]. In particular, none of these results considers robots in a polygon.

The work that is most relevant to ours is represented by a series of papers on Rendezvous and approximate Rendezvous by two robots in polygons or more general planar enclosures [9, 10, 11, 13]. The authors show how to guarantee that the two robots' trajectories will intersect (or get arbitrary close to each other in case of approximate rendezvous) within finite time, in spite of a powerful adversary that controls the speed and the movements of the robots on their trajectories. However, termination happens implicitly: the robots are not necessarily aware of each other's presence, and Rendezvous is considered solved even if they are both moving. Moreover, none of these papers considers oblivious robots, and none of them allows the initial memory contents of the robots to be arbitrary. Both are simplifications of the problem, because they allow robots to implicitly agree on a single landmark and just move there.

Other mildly related works consider robots searching for an intruder in a polygon [27, 28], robots in an empty plane that obstruct each other's view, whose task is to become all mutually visible [20, 24, 25], robots tasked with constructing a map of the visibility graph of a polygon [4, 5, 6], and a static version of the Meeting problem called the *hidden set problem* [26].

The issue of defining a model of computation for mobile robots has hardly ever been addressed in the relevant literature. It is nonetheless interesting to establish what destination points are computable by mobile robots, and what it means for them to compute a point. To the best of our knowledge, only two papers deal with this problem [7, 17], explicitly rejecting transcendental functions and deeming them not intuitively computable. Interestingly, other papers, such as [9], allow robots to compute transcendental functions.

Another contribution of this paper is a formal definition of the concept of computability for mobile robots (see the beginning of Section 4.2). Accordingly, all of our geometric constructions can be performed with a compass.

2 Definitions

A *polygon* in the Euclidean plane $\mathbb{R}^2$ is a non-empty, bounded, connected, and topologically closed 2-manifold whose boundary is a finite collection of line segments. The *vertices*, *edges*, and *diagonals* of a polygon are defined in the standard way, as well as the notion of *adjacency* between vertices. One connected component of a polygon's boundary, called the *external boundary*, encloses all others, which are called *holes*. We say that a point $p \in P$ *sees* a point $q \in P$ (or, equivalently, that q is *visible* to p) if the line segment pq lies in P.

If a polygon has an axis of symmetry, we say that it is *axially symmetric*. The largest integer σ such that rotating a polygon around its barycenter by $2\pi/\sigma$ radians leaves it unchanged is called the *symmetricity* of the polygon. In other words, the symmetricity is the order of the rotation group of the polygon. If $\sigma > 1$, the polygon is said to be *rotationally symmetric*.

If P is a polygon, by *P-searcher* we mean an anonymous *robot* represented by a point in P, which, informally, can *observe* its surroundings and *move* within P. The technical specifications of our searchers have been listed in Section 1.1. We remark that each searcher's snapshots are expressed in its own *local reference system*, which is a Cartesian system of coordinates with the searcher's current location as the origin. A searcher's local coordinate system translates as the searcher moves (to keep the searcher's location at the origin), but it retains its orientation, scale, and handedness.

We say that two P-searchers are *mutually aware* at some point in time if they have seen each other during their most recent Look phases. That is, if searcher s_1 sees searcher s_2 during a Look phase at time t_1, s_2 sees s_1 during a Look phase at time $t_2 \geq t_1$, and neither s_1 nor s_2 performs another Look phase in the time interval (t_1, t_2), then s_1 and s_2 are mutually aware at time t_2 (and they remain mutually aware until s_1 performs a Look phase without seeing s_2, or vice versa). A very similar notion of mutual awareness has been defined in [21].

Given a team of P-searchers, the *Meeting problem* prescribes that at least two of them become mutually aware. More precisely, the Meeting problem for k searchers in P is *solvable* if there exists an algorithm A such that, if all k searchers execute A during all their Compute phases, at least two of them eventually become mutually aware, regardless of how the searchers are initially laid out in P, and regardless of how the scheduler decides to control their behavior. Occasionally, we will say that two searchers *meet*, as a synonym of becoming mutually aware.

In Section 3, we are going to assume that each searcher has an unlimited amount of *persistent internal memory*, which can be read and updated by the searcher during each Compute phase, and is retained for use in later Compute phases. The initial contents of the internal memory of each searcher are arbitrary, and possibly "incorrect". In Section 4, we will drop the persistent memory requirements, and we will extend our algorithms to *oblivious* searchers, whose computations only rely on the single snapshot taken in the most recent Look phase, and whose internal memory is erased during each Move phase.

3 Algorithms and Correctness

3.1 General Algorithm

First we give a lower bound on the minimum number of searchers required to solve the Meeting problem in a polygon. Our bound is in terms of the polygon's symmetricity.

▶ **Theorem 1.** *For every integer $\sigma > 0$, there exists a polygon with symmetricity σ in which σ (or fewer) searchers cannot solve the Meeting problem.*

Proof. If $\sigma = 1$, the statement is trivial. If $\sigma > 1$, we construct a polygon with symmetricity σ shaped as a σ-pointed star with one large hole almost touching the external boundary, as shown in Figure 1. We then arrange $\sigma' \leq \sigma$ searchers and orient their local coordinate systems in a symmetric fashion, as in Figure 1. Now, let the initial memory contents of all the searchers be equal, and suppose that the scheduler always activates them synchronously. By the rotational symmetry of our construction, each searcher gets an identical snapshot of the polygon, and therefore all searchers compute symmetric destination points and modify their memory in the same way. This holds true at every cycle, and so, by induction, the searchers will always be found at σ' symmetric locations throughout the execution. Note that our polygon has the property that no two of its points whose angular distance (with respect to the barycenter) is a multiple of $2\pi/\sigma$ can see each other. Hence, no matter what algorithm the searchers are executing, no two of them will ever be mutually aware. ◀

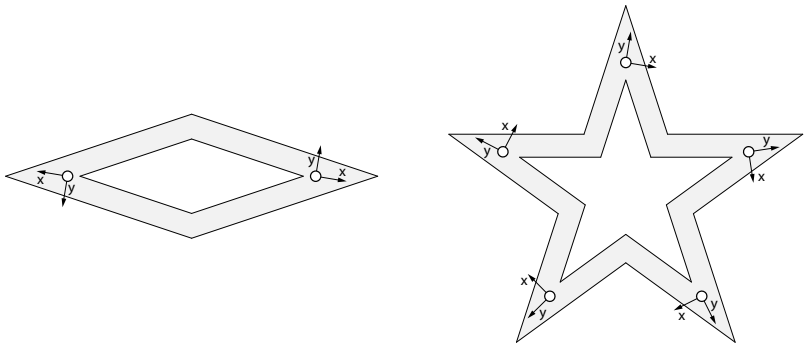


Figure 1 Constructions used in Theorem 1 for $\sigma = 2$ and $\sigma = 5$.

Algorithm 1 Meeting algorithm for general polygons.

```
Persistent variables
SnapshotList
Action
Direction
Polygon
PivotPoint

Procedure Compute (Snapshot)
if Snapshot contains no other searcher then
    Append Snapshot to SnapshotList
    if SnapshotList is inconsistent or (Action = PATROL and PivotPoint is not consistent with Polygon) then
        SnapshotList := Snapshot
        Action := EXPLORE
    if Action = EXPLORE then
        U := Extract unvisited vertices from SnapshotList
        if U ≠ ∅ then
            v := First vertex of U
            Move to the next vertex in a shortest path to v
        else
            Action := PATROL
            Direction := CLOCKWISE
            Polygon := Extract polygon from SnapshotList
            S := Set of axes of symmetry of Polygon
            if S = ∅ then
                C := Select a rotation class of vertices of Polygon in a similarity-invariant way
                PivotPoint := Select any vertex in C
            else
                S' := Select a class of equivalent axes in S in a similarity-invariant way
                ℓ := Select any axis in S'
                C := Select a class of points of ℓ on the boundary of Polygon in a similarity-invariant way
                PivotPoint := Select any point in C
            Augment Polygon using PivotPoint as pivot in a similarity-invariant way to make it simply connected
    if Action = PATROL then
        if I am in PivotPoint then
            Invert Direction
        Move to the next vertex of Polygon, following its boundary in the direction stored in variable Direction
```

Next we will prove that the bound of Theorem 1 is tight, and hence $\sigma + 1$ searchers are optimal. We do so by giving a Meeting algorithm called Algorithm 1, which works by constructing a map of the polygon in a self-stabilizing way. This algorithm assumes that searchers have unlimited memory, and hence they can store the entire history of the snapshots they have taken since the beginning of the execution. In Section 4, we will show how to drop this requirement and apply our algorithms to oblivious searchers.

By definition, the Meeting problem is solved when two searchers become mutually aware. So, in our algorithm, whenever a searcher sees another searcher, it stays idle for a cycle and waits to be noticed (which does not necessarily happen, since the second searcher may be in the middle of a Move phase and disappear shortly after). Let P be the polygon in which

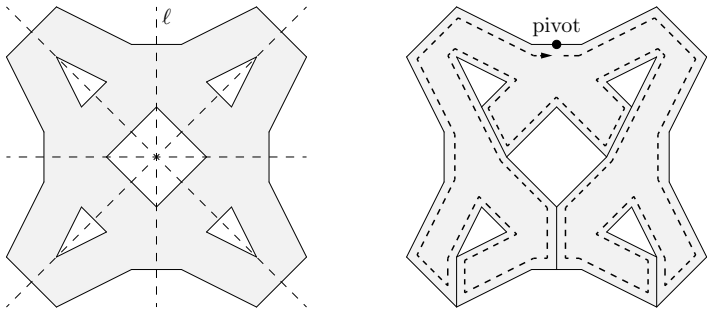


Figure 2 Augmenting an axially symmetric polygon and defining a tour of its boundary.

the searchers are located. Since the initial memory contents may be incorrect, if a searcher notices a discrepancy between the current snapshot of P and the history of snapshots stored in memory, it forgets everything and restarts the execution from wherever it is.

As observed in Section 1.2, each searcher must keep re-visiting every part of the boundary of P. Hence, our main algorithm is divided into two phases: EXPLORE and PATROL. Roughly speaking, in the EXPLORE phase, a searcher visits all vertices of P; in the PATROL phase, it moves back and forth along the boundary of P, searching for a companion. The EXPLORE phase is relatively simple: as the searcher explores new vertices, it keeps track of the ones that it has seen but not visited. Then it picks the first of such vertices and moves to it along a shortest path.

For the PATROL phase, the searcher must first choose a *pivot point* of P, which is the point where the searcher changes direction as it patrols P's boundary. It also has to cope with the fact that the boundary of P may not be connected, since P may have holes. The pivot point is always chosen on the boundary of P and on a symmetry axis of P, if one exists. It is also chosen in a *similarity-invariant* way, meaning that the selection algorithm should not depend on the scale, rotation, position, and handedness of P, but it should be a deterministic algorithm that only looks at angles between vertices and ratios between segment lengths. This is to guarantee that all searchers that have a correct picture of P in memory (expressed in their respective local coordinates systems) will select pivot points that are equivalent up to similarity.

Once a searcher has selected a pivot point, it adds some "artificial" edges to P in order to make it simply connected, i.e., remove all its holes. This may be impossible to do in a similarity-invariant way, so the pivot point is used to determine how symmetries are broken. An example of this construction is illustrated in Figure 2, where the polygon is symmetric and therefore the additional edges are symmetric, as well.

As a result of cutting P along these segments, we obtain a *degenerate* simply connected polygon $\widetilde{P}$. It is now possible to perform a *tour* of the boundary of $\widetilde{P}$, as shown in Figure 2. The PATROL phase of our algorithm consists in taking a tour of $\widetilde{P}$ and switching direction (from clockwise to counterclockwise and vice versa) every time the pivot point is reached. So, all vertices of P are perpetually visited in some fixed order, then in the opposite order, and so on. At any time, the searcher can always determine its next destination vertex based on the history of snapshots stored in memory.

▶ **Theorem 2.** *There is an algorithm that, for every integer $\sigma > 0$, solves the Meeting problem with $\sigma + 1$ searchers (regardless of their initial memory contents) in every polygon with symmetricity σ.*

Proof. We will show that Algorithm 1 correctly solves the Meeting problem for $\sigma+1$ searchers in any polygon P with symmetricity σ. Since the initial memory contents of a searcher may be incorrect, when a searcher notices a discrepancy between the current observation and a previous observation, it erases its own memory and restarts the execution. The same happens if it realizes that the pivot point it has chosen does not match the polygon. From that point onward, the searcher's memory will only contain correct information, and the execution will never be restarted again. Hence, in the following, we will assume that no such discrepancy is ever discovered, and therefore the execution is never restarted.

In the EXPLORE phase, a searcher moves to all the discovered but not yet visited vertices of P, until there are none left. This indeed lets the searcher discover all vertices of P, due to the connectedness of its *visibility graph* (which is the graph on the set of vertices of P whose edges are the edges and diagonals of P). This means that eventually the searcher will have a complete and coherent representation of *some* polygon in memory. If this representation is incorrect, the searcher will eventually find out during the PATROL phase, when it re-visits all vertices of P.

We can therefore assume without loss of generality that, at some point, all searchers are in the PATROL phase, and all have a correct representation of P in memory. Since the pivot point is chosen by each searcher in a similarity-invariant way, there are only σ points that can possibly be elected as pivots. By the pigeonhole principle, there are two searchers that have the same pivot point. So, both searchers will perform a clockwise tour of the boundary of $\widetilde{P}$, touching all of its vertices is some fixed order, followed by a counterclockwise tour, touching all vertices in the opposite order, and so on. Since they both turn around at the same pivot point, they do the same tour. As a consequence, they become mutually aware by the time one of them has completed a full tour, thus solving the Meeting problem. ◀

We emphasize that, if a searcher were tasked to construct a map of P, it could do so by simply executing the above algorithm indefinitely (i.e., ignoring the presence of other searchers). Since the algorithm eventually discovers and corrects any possible inconsistency in the initial memory state of the searcher, it is self-stabilizing.

3.2 Improved Algorithm for Polygons with Barycenter not in a Hole

The worst-case examples given in Theorem 1 are polygons with a hole around the barycenter. It is natural to wonder if the Meeting problem can be solved with fewer searchers if we exclude this special type of polygons. It turns out that in all other cases Algorithm 1 can be drastically improved: only two searchers are needed whenever the polygon's barycenter is not in a hole. Notably, this includes all polygons with no holes.

It is not hard to construct counterexamples where simply making the searchers patrol the boundary of the polygon as in the previous algorithm may not solve the Meeting problem, even if the polygon has no holes. Hence a new strategy has to be devised: our improved Meeting algorithm is called Algorithm 2. It begins by testing for the presence of another searcher, followed by some consistency tests, and an EXPLORE phase, which are essentially the same as in the previous algorithm. It then proceeds with a PATROL phase, which is more complex than the old one. Note that Algorithm 1 already solves the Meeting problem with two searchers if the polygon is not rotationally symmetric (i.e., for $\sigma = 1$). So, in this special case, our improved algorithm works exactly as the previous one. In the following, we will therefore assume that the polygon is rotationally symmetric, and we will discuss only the new PATROL phase.

Algorithm 2 Improved Meeting algorithm for polygons with barycenter not in a hole.

```
Persistent variables
SnapshotList
Action
Stage
Polygon
PivotVertex
PolygonTriangles
PolygonLevels

Procedure Compute (Snapshot)
if Snapshot contains no other searcher then
    Append Snapshot to SnapshotList
    if Persistent variables are inconsistent then
        SnapshotList := Snapshot
        Action := EXPLORE
    if Action = EXPLORE then
        U := Extract unvisited vertices from SnapshotList
        if U ≠ ∅ then
            v := First vertex of U
            Move to the next vertex in a shortest path to v
        else
            Action := PATROL
            Stage := −1
            Polygon := Extract polygon from SnapshotList
            if Polygon is rotationally symmetric then
                C := Select a class of vertices of Polygon closest to the center in a similarity-invariant way
                PivotVertex := Select any vertex in C
                Augment Polygon in a similarity-invariant way to make it simply connected
                Triangulate each branch of augmented Polygon in a similarity-invariant way
                PolygonTriangles := Total number of triangles in the triangulation of augmented Polygon
                PolygonLevels := Height of the dual tree of the triangulation of each branch of augmented Polygon
            else
                PivotVertex := Select a vertex of Polygon in a similarity-invariant way
    if Action = PATROL then
        if Polygon is rotationally symmetric then
            if I am in PivotVertex then
                Stage := Stage + 1
                if Stage ≥ 2 · PolygonLevels + 2 · PolygonTriangles^2 then
                    Stage := 0
            if Stage = −1 then
                Move to the next vertex in a shortest path to PivotVertex
            else if Stage < PolygonLevels then
                j := Stage
                Move to the next vertex of a clockwise j-tour of Polygon
            else
                j := 2 · PolygonLevels + 2 · PolygonTriangles^2 − Stage
                if j > PolygonLevels then
                    j := PolygonLevels
                Move to the next vertex of a counterclockwise j-tour of Polygon
        else
            if I am in PivotVertex then
                Stage := Stage + 1
            if Stage is odd then
                Move to the next vertex of Polygon, following its boundary in the clockwise direction
            else
                Move to the next vertex of Polygon, following its boundary in the counterclockwise direction
```

Upon ending the EXPLORE phase, a searcher does some pre-processing on the polygon P. First it identifies a polygon Q formed by vertices of P that are closest to the center of rotation and equivalent up to similarity. So, Q is a convex polygon that is completely contained in P. Each connected component of $P \setminus Q$ is called a *branch* of P. Each branch is then augmented by some extra edges to make it simply connected, obtaining a degenerate polygon $\widetilde{P}$. Then, $\widetilde{P} \setminus Q$ is triangulated in a similarity-invariant way. This guarantees that all searchers compute the same triangulation. Figure 3 shows an example of this construction.

Let P_j be the union of Q and the triangles of the triangulation whose corresponding nodes of the dual graph have distance at most j from the root Q. Let m be the smallest integer such that $P_m = P$. The PATROL phase has a "primitive" operation called j-*tour*,

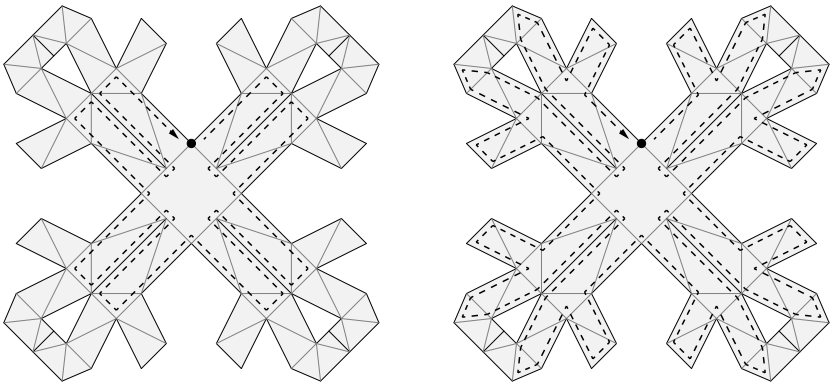


Figure 3 Augmented and triangulated axially symmetric polygon with a 3-tour and a 6-tour.

with $0 \leq j \leq m$, which is a tour of the boundary of P_j, starting and ending at the pivot point, following the edges of $\widetilde{P}$. For example, a 0-tour is simply a tour of the boundary of Q, an m-tour is a tour of the boundary of $\widetilde{P}$ (much like the tours of Algorithm 1), and Figure 3 shows a 3-tour and a 6-tour.

The PATROL phase consists of several *stages*, and in each stage the searcher performs a j-tour, for some j. The j-tours are performed according to the following list, which is repeated until the Meeting problem is solved: a clockwise 0-tour, a clockwise 1-tour, a clockwise 2-tour, ..., a clockwise $(m-1)$-tour, a sufficiently large number of counterclockwise m-tours (twice the square of the total number of triangles in the triangulations of all the branches of P is abundantly enough), a counterclockwise $(m-1)$-tour, a counterclockwise $(m-2)$-tour, ..., a counterclockwise 1-tour. The first m stages, where the searcher performs clockwise j-tours, are called *ascending stages*. All the other stages are called *descending stages*. Moreover, the first stage is called the *central stage*, and the stages in which an m-tour is performed are called *perimeter stages*.

Note that, when we say "clockwise" and "counterclockwise", we mean it in the local reference system of the executing searcher. Recall that two different searchers executing the algorithm may not have the same notion of clockwise direction, and therefore in their respective ascending stages they may actually perform tours in opposite directions. If two searchers have the same notion of clockwise direction, they are said to be *concordant*; otherwise, they are *discordant*.

We can now proceed with the proof of correctness of Algorithm 2. The proof of the following technical lemmas is found in the appendix.

▶ **Lemma 3.** *Let two P-searchers be executing Algorithm 2, let both be in the PATROL phase, and let both have a correct representation of the polygon P in memory, which is rotationally symmetric. Then, they will either become mutually aware or be in a perimeter stage at the same time, with at least one full perimeter stage still to perform.*

▶ **Lemma 4.** *Assume the hypotheses of Lemma 3, and let the searchers be concordant. If one searcher begins a j-tour in an ascending (respectively, descending) stage while the other searcher is performing a $(j+1)$-tour (respectively, $(j-1)$-tour) in a descending (respectively, ascending) stage, with $0 \leq j < m$ (respectively, $0 < j < m$), they eventually become mutually aware.*

▶ **Theorem 5.** *There is an algorithm that solves the Meeting problem with two searchers (regardless of their initial memory contents) in every polygon whose barycenter does not lie in a hole.*

Proof. We will show that Algorithm 2 correctly solves the Meeting problem for two searchers in any polygon P whose barycenter does not lie in a hole. The proof of correctness is the same as that of Theorem 2, except for the PATROL phase. So, in the following, we will assume that both searchers already have a correct picture of P in memory, and are both in the PATROL phase. Moreover, since the new algorithm works in the same way as the old one if P is not rotationally symmetric (and the proof of correctness is the same as in Theorem 2), we will assume that P is rotationally symmetric.

Suppose for a contradiction that the searchers never become mutually aware. By Lemma 3, at some point they are both in a perimeter stage. If they are discordant, they proceed along the boundary of $\widetilde{P}$ in opposite directions, and therefore they become mutually aware. If they are concordant, they will perform all the remaining descending stages, followed by the ascending stages, starting with the central stage. If they start the central stage at the same time, they necessarily become mutually aware, because they are on the boundary of the central polygon Q, which is convex and empty. So, one searcher must begin the central stage while the other is still in a descending stage. Then, as one searcher ascends and the other descends, the hypotheses of Lemma 4 are going to be satisfied, which means that the searchers eventually become mutually aware. ◀

4 Memoryless Implementations

4.1 Encoding Persistent Variables

In order to re-implement our algorithms without using any memory, we first want to be able to encode all persistent variables (which roughly amount to a list of snapshots) as a single real number. Since a snapshot is the visible sub-polygon of P expressed in the coordinate system of the observing P-searcher, it can be easily expressed as the array of the visible sub-segments of P's edges, or as the array of the coordinates of these segments' endpoints. Hence, encoding a history of snapshots amounts to encoding a finite array of real numbers $(r_1, \ldots, r_n)$ as a single real number r.

As a model of computation, we choose the *Blum-Shub-Smale machine* [3], which is a random-access machine whose registers can store arbitrary real numbers. Its computational primitives are the four basic arithmetic operations, but it is customary to extend the basic model with additional primitives, which should be somewhat well-behaved and intuitively computable. This immediately rules out the naive approach of constructing the binary representation of r by interleaving the binary digits of the r_i's. Indeed, such an encoding function has a set of discontinuities that is dense in its entire domain, and is therefore not intuitively computable.

We propose a more sophisticated encoding strategy, which is computable even on a basic Blum-Shub-Smale machine (i.e., the one with the four basic arithmetic operations only). A small drawback is that we can only apply this method if the vertices of the polygon P have algebraic coordinates (i.e., they are *algebraic points*) in some global coordinate system. In practice, we are not imposing a big limitation on our inputs, in that basically all the polygons we can reasonably think of fall into this class. In particular, we can construct polygons of any symmetricity (by contrast, using rational points would only allow us to construct polygons with symmetricity 1, 2, and 4).

First of all, we observe that we can represent an algebraic number using finitely many bits: if the algebraic number α is the ith real root of the polynomial with integer coefficients $a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$, we can encode it as $(n, i, a_n, a_{n-1}, \ldots, a_0)$, which in turn requires only finitely many bits. Given α, a suitable polynomial can be found by simply generating all possible polynomials in some order (they are a countable set) and evaluating them in α.

Once we have some algebraic numbers expressed in this implicit finite form, we can use our basic Blum-Shub-Smale machine to do Turing-computable bit manipulations to compute all kinds of common functions on them. In particular, there are standard ways of computing the basic arithmetic operations, as well as root extractions of any degree. A comprehensive exposition of these techniques, along with their theoretical background, is found in [8]. Essentially, this is also one of the several ways in which mathematical software such as Sage, Mathematica, and CGAL handles algebraic numbers and does exact computations with then.

The key point to keep in mind is that, once a number is encoded in this form, we cannot necessarily retrieve it in finite time; we can only approximate it arbitrarily well, for instance via Sturm's theorem [8]. However, we can still evaluate computable predicates on these numbers exactly, and have them influence the flow of our algorithms.

Note that, even if the vertices of P are algebraic points in some global coordinate system, they may not be algebraic in the coordinate system of a searcher. So, the idea is to let the searcher use two visible vertices v and v' as a basis to construct a new coordinate system, and then compute the coordinates of all other visible vertices of P in this system. The resulting coordinates are guaranteed to be algebraic numbers, and can be encoded with the above technique. Note that some vertices of the snapshot are not necessarily vertices of P, and may not be algebraic even in the new coordinate system. These vertices are easily identified and discarded. This causes some loss of information, which is not going to matter in our application to the Meeting problem.

4.2 Adapting the Algorithms

Next we are going to apply our encoding method to the Meeting algorithms of Section 3, and we will show how oblivious searchers can solve the Meeting problem, as well. We will need searchers to be able to compute only basic arithmetic operations on real numbers, as well as extract square roots. Hence, internally, each searcher will run a Blum-Shub-Smale machine extended with a square-root primitive. Only the four basic arithmetic operations are required for our encoding method, but square roots are needed in the geometric computations. It is well known that the points whose coordinates can be computed by composing these five operations are precisely the ones that can be constructed with a compass and a straightedge. In turn, the Mohr-Mascheroni theorem states that these points can also be constructed with a compass alone.

A searcher can simulate memory by encoding data as its distance d from the closest visible vertex of P, which we call the *virtual vertex* of the searcher. It is not hard to design the code in such a way that d and $d/2$ have the same meaning, for every d. This gives a searcher the ability to get arbitrarily close to its virtual vertex without losing memory. After decoding d, a searcher will execute one of the old Meeting algorithms pretending to be located on the virtual vertex. This approach poses several technical problems, which we examine next.

Recall that snapshots are encoded in a special coordinate system defined by two visible vertices v and v'. As the searcher moves to a different virtual vertex, it has to re-express all the history of snapshots into a new coordinate system, choosing a v and a v' that will be visible from its destination point. This can be done by letting v be the current virtual vertex and v' be the destination virtual vertex, and adding enough information to the encoded data to retrieve v once the virtual vertex is v'. This way, the searcher can transport snapshots around P and is still able to decode them.

In the EXPLORE phase, a searcher completely explores one connected component of the boundary of P before moving to the next component. To determine the position of the next vertex, it has to move close enough to the current virtual vertex, and possibly around it if it

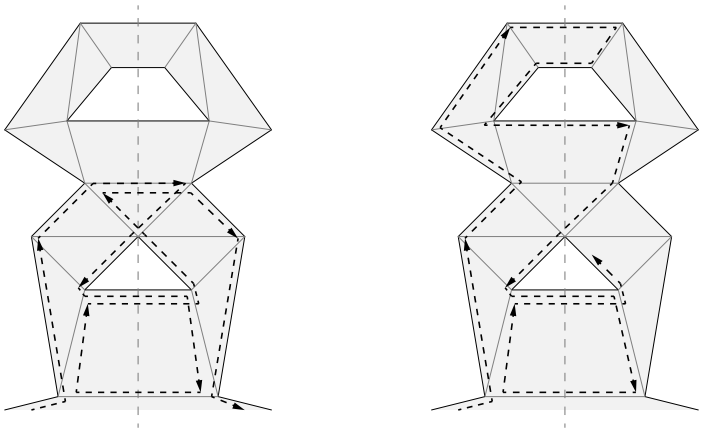


Figure 4 Symmetric partition of a branch with a 5-tour and part of a perimeter tour.

is reflex. The searcher makes sure to stop on the angle bisector of each vertex it visits and take a snapshot from there. If implemented properly, this strategy guarantees that, when the list of discovered but not visited vertices is empty, all vertices of P have actually been visited, and the map of P is faithful.

For the PATROL phase, recall that a tour turns at the vertices of the augmented polygon $\widetilde{P}$, which are not necessarily vertices of P. Unfortunately, our oblivious searchers cannot approach generic points without losing information. Our solution is to modify the tours so that they turn only at vertices of P. These modifications should retain the axes of symmetry of the tours whenever they are required by the original algorithms. This is especially tricky for the j-tours of Algorithm 2, where we need to redesign the partition of $\widetilde{P}$ including not only triangles but also isosceles trapezoids, as shown in Figure 4. Note that our new j-tours may self-overlap and touch the same vertices multiple times, but this is not an issue. These new j-tours have the relevant properties that are required by Lemmas 3 and 4: the pieces of the partition are convex, and each point on a $(j+1)$-tour is visible to at least an entire edge of a j-tour.

Of course, a j-tour defines a curve that is followed only approximately by our oblivious searchers, because they have to keep encoding information as they go. Still, they follow this curve closely enough and without ever properly crossing it, in such a way as to preserves the aforementioned relevant properties. Also, while they do so, they have to stop on the angle bisector of each vertex whenever they get the opportunity: this is to guarantee that they will discover any discrepancies between the encoded map and the real polygon.

Finally, if two searchers are doing the same tour in opposite directions, and they stay within a thin-enough "band" around the correct curve, they are guaranteed to become mutually aware when they cross each other.

Therefore, Theorems 2 and 5 can be extended to oblivious searchers.

▶ **Theorem 6.** *There is an algorithm that, for every integer $\sigma > 0$, solves the Meeting problem with $\sigma + 1$ oblivious searchers in every polygon with symmetricity σ. There is an algorithm that solves the Meeting problem with two oblivious searchers in every polygon whose barycenter does not lie in a hole. If the polygon's vertices are algebraic points, these algorithms are implementable on a real random-access machine that can compute basic arithmetic operations and extract square roots.*

5 Conclusions and Further Work

Our Meeting algorithms can be converted into Rendezvous algorithms for two searchers. Indeed, once two searchers realize that they have become mutually aware, they can implicitly agree on a rendezvous point using the boundary of the polygon as a static reference. Note that this "rendezvous phase" should only be performed if the searchers are in fact mutually aware, or else they may behave incorrectly and compromise the algorithms. To detect mutual awareness, a searcher that sees another searcher performs a preliminary "synchronization phase", in which it moves very slightly a few times and sees if the other searcher does the same or disappears. If the pattern of these "synchronization moves" is unique to this phase, i.e., it cannot be normally performed during an EXPLORE or a PATROL phase, then the searchers know that they are mutually aware, and they can proceed to the rendezvous phase.

In some cases, we can also extend our algorithms to weaker models of searchers. Namely, searchers with *limited visibility*, which can only see up to a fixed distance, and the *non-rigid* setting, in which a searcher can be stopped by the scheduler during each Move phase before reaching its destination point, but not before having moved by at least a constant δ (for details on this model, refer to [16]). However, a complete solution for these and other searcher models remains an open problem.

References

1 S. Alpern and S. Gal. *The theory of search games and rendezvous*. Springer, 2003.

2 H. Ando, Y. Oasa, I. Suzuki, and M. Yamashita. Distributed memoryless point convergence algorithm for mobile robots with limited visibility. *IEEE Transactions on Robotics and Automation*, 15(5):818–828, 1999.

3 L. Blum, F. Cucker, M. Shub, and S. Smale. *Complexity and real computation*. Springer-Verlag New York, 1998.

4 J. Chalopin, S. Das, Y. Disser, M. Mihalák, and P. Widmayer. Mapping simple polygons: how robots benefit from looking back. *Algorithmica*, 65(1):43–59, 2013.

5 J. Chalopin, S. Das, Y. Disser, M. Mihalák, and P. Widmayer. Simple agents learn to find their way: an introduction on mapping polygons. *Discrete Applied Mathematics*, 161(10–11):1287–1307, 2013.

6 J. Chalopin, S. Das, Y. Disser, M. Mihalák, and P. Widmayer. Mapping simple polygons: the power of telling convex from reflex. *ACM Transactions on Algorithms*, 11(4):33:1–33:16, 2015.

7 M. Cieliebak, P. Flocchini, G. Prencipe, and N. Santoro. Distributed computing by mobile robots: Gathering. *SIAM Journal on Computing*, 41(4):829–879, 2012.

8 H. Cohen. *A course in computational algebraic number theory*. Springer, 1993.

9 J. Czyzowicz, D. Ilcinkas, A. Labourel, and A. Pelc. Asynchronous deterministic rendezvous in bounded terrains. *Theoretical Computer Science*, 412(50):6926–6937, 2011.

10 J. Czyzowicz, A. Kosowski, and A. Pelc. Deterministic rendezvous of asynchronous bounded-memory agents in polygonal terrains. *Theory of Computing Systems*, 52(2):179–199, 2013.

11 J. Czyzowicz, A. Labourel, and A. Pelc. How to meet asynchronously (almost) everywhere. *ACM Transactions on Algorithms*, 8(4):37:1–37:14, 2012.

12 X. Défago, M. Gradinariu, S. Messika, and P. Raïpin-Parvédy. Fault-tolerant and self-stabilizing mobile robots gathering. In *Proceedings of the 20th International Conference on Distributed Computing, (DISC)*, pages 46–60, 2006.

13 Y. Dieudonné and A. Pelc. Deterministic polynomial approach in the plane. *Distributed Computing*, 28(2):111–129, 2015.

14 Y. Dieudonné, A. Pelc, and V. Villain. How to meet asynchronously at polynomial cost. *SIAM Journal on Computing*, 44(3):844–867, 2015.

15 Y. Dieudonné and F. Petit. Self-stabilizing gathering with strong multiplicity detection. *Theoretical Computer Science*, 428:47–57, 2012.

16 P. Flocchini, G. Prencipe, and N. Santoro. *Distributed computing by oblivious mobile robots*. Morgan & Claypool, 2012.

17 P. Flocchini, G. Prencipe, N. Santoro, and G. Viglietta. Distributed computing by mobile robots: Uniform Circle Formation. *Distributed Computing*, in press.

18 P. Flocchini, G. Prencipe, N. Santoro, and P. Widmayer. Gathering of asynchronous robots with limited visibility. *Theoretical Computer Science*, 337(1–3):147–168, 2005.

19 P. Flocchini, N. Santoro, G. Viglietta, and M. Yamashita. Rendezvous with constant memory. *Theoretical Computer Science*, 621:57–72, 2016.

20 G. A. Di Luna, P. Flocchini, S. Gan Chaudhuri, F. Poloni, N. Santoro, and G. Viglietta. Mutual visibility by luminous robots without collisions. *Information and Computation*, 254(3):392–418, 2017.

21 L. Pagli, G. Prencipe, and G. Viglietta. Getting close without touching: Near-Gathering for autonomous mobile robots. *Distributed Computing*, 28(5):333–349, 2015.

22 W. Rabinovich, J. Murphy, M. Suite, M. Ferraro, R. Mahon, P. Goetz, K. Hacker, W. Freeman, E. Saint Georges, S. Uecke, and J. Sender. Free-space optical data link to a small robot using modulating retroreflectors. In *Proceedings of SPIE 7464, Free-Space Laser Communications IX*, volume 7464, 2009.

23 I. Rekleitis, V. Lee-Shue, A. Peng New, and H. Choset. Limited communication, multi-robot team based coverage. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 3462–3468, 2004.

24 G. Sharma, C. Busch, and S. Mukhopadhyay. Mutual visibility with an optimal number of colors. In *Proceedings of the 11th International Symposium on Algorithms and Experiments for Wireless Sensor Networks (ALGOSENSORS)*, pages 196–210, 2016.

25 G. Sharma, R. Vaidyanathan, J. L. Trahan, C. Busch, and S. Rai. Complete visibility for robots with lights in O(1) time. In *Proceedings of the 18th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 327–345, 2016.

26 T. Shermer. Hiding people in polygons. *Computing*, 42(2):109–131, 1989.

27 I. Suzuki and M. Yamashita. Searching for a mobile intruder in a polygonal region. *SIAM Journal on Computing*, 21(5):863–888, 1992.

28 M. Yamashita, H. Umemoto, I. Suzuki, and T. Kameda. Searching for mobile intruders in a polygonal region by a group of mobile searchers. *Algorithmica*, 31(2):208–236, 2001.

Three Notes on Distributed Property Testing*

Guy Even[†1], Orr Fischer[2], Pierre Fraigniaud[‡3], Tzlil Gonen[4], Reut Levi[5], Moti Medina[6], Pedro Montealegre[§7], Dennis Olivetti[8], Rotem Oshman[¶9], Ivan Rapaport[‖10], and Ioan Todinca[11]

1 **Tel Aviv University, School of Electrical Engineering, Tel Aviv, Israel**
guy@eng.tau.ac.il
2 **Tel Aviv University, Computer Science Department, Tel Aviv, Israel**
orrfischer@mail.tau.ac.il
3 **CNRS and University Paris Diderot, Paris, France**
pierre.fraigniaud@irif.fr
4 **Tel Aviv University, Computer Science Department, Tel Aviv, Israel**
tzlilgon@mail.tau.ac.il
5 **Max Planck Institute for Informatics, Saarland Informatics Campus, Saarbrücken, Germany**
rlevi@mpi-inf.mpg.de
6 **Max Planck Institute for Informatics, Saarland Informatics Campus, Saarbrücken, Germany**
mmedina@mpi-inf.mpg.de
7 **Facultad de Ingeniería y Ciencias, Universidad Adolfo Ibáñez, Santiago de Chile, Chile**
p.montealegre@uai.cl
8 **Gran Sasso Science Institute, L'Aquila, Italy**
dennis.olivetti@gssi.infn.it
9 **Tel Aviv University, Computer Science Department, Tel Aviv, Israel**
roshman@tau.ac.il
10 **DIM-CMM (UMI 2807 CNRS), Universidad de Chile, Santiago de Chile, Chile**
rapaport@dim.uchile.cl
11 **Université d'Orléans, INSA Centre Val de Loire, LIFO EA 4022, Orléans, France**
ioan.todinca@univ-orleans.fr

Abstract

In this paper we present distributed property-testing algorithms for graph properties in the CONGEST model, with emphasis on testing subgraph-freeness. Testing a graph property $\mathcal{P}$ means distinguishing graphs $G = (V, E)$ having property $\mathcal{P}$ from graphs that are ε-far from having it, meaning that $\varepsilon|E|$ edges must be added or removed from G to obtain a graph satisfying $\mathcal{P}$.

We present a series of results, including:

- Testing H-freeness in $O(1/\varepsilon)$ rounds, for any constant-sized graph H containing an edge (u, v) such that any cycle in H contain either u or v (or both). This includes all connected graphs over five vertices except K_5. For triangles, we can do even better when ε is not too small.

* Full versions related to the paper are available at https://arxiv.org/abs/1705.04898 and http://arxiv.org/abs/1705.04033 [16, 17].

† Work done while visiting Max Planck Institute for Informatics.
‡ Additional support from ANR Project DESCARTES, and from INRIA Project GANG.
§ This work was partially supported by CONICYT via Basal in Applied Mathematics.
¶ Orr Fischer, Tzlil Gonen and Rotem Oshman are supported by the Israeli Centers of Research Excellence (I-CORE) program, (Center No. 4/11) and by BSF Grant No. 2014256.
‖ This work was partially supported by Fondecyt 1170021, Núcleo Milenio Información y Coordinación en Redes ICM/FIC RC130003.

© Guy Even, Orr Fischer, Pierre Fraigniaud, Tzlil Gonen, Reut Levi, Moti Medina, Pedro Montealegre, Dennis Olivetti, and Rotem Oshman;
licensed under Creative Commons License CC-BY

31st International Symposium on Distributed Computing (DISC 2017).
Editor: Andréa W. Richa; Article No. 15; pp. 15:1–15:30

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

- A deterministic CONGEST protocol determining whether a graph contains a given tree as a subgraph in constant time.
- For cliques K_s with $s \geq 5$, we show that K_s-freeness can be tested in $O(m^{\frac{1}{2}-\frac{1}{s-2}} \cdot \varepsilon^{-\frac{1}{2}-\frac{1}{s-2}})$ rounds, where m is the number of edges in the network graph.
- We describe a general procedure for converting ε-testers with $f(D)$ rounds, where D denotes the diameter of the graph, to work in $O((\log n)/\varepsilon) + f((\log n)/\varepsilon)$ rounds, where n is the number of processors of the network. We then apply this procedure to obtain an ε-tester for testing whether a graph is bipartite and testing whether a graph is cycle-free. Moreover, for cycle-freeness, we obtain a *corrector* of the graph that locally corrects the graph so that the corrected graph is acyclic. Note that, unlike a tester, a corrector needs to mend the graph in many places in the case that the graph is far from having the property.

These protocols extend and improve previous results of [Censor-Hillel et al. 2016] and [Fraigniaud et al. 2016].

1998 ACM Subject Classification F.2 Analysis of Algorithms and Problem Complexity

Keywords and phrases Property testing, Property correcting, Distributed algorithms, CONGEST model

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.15

1 General Introduction

Distributed decision refers to tasks in which the computing elements of a distributed system have to collectively decide whether the system satisfies some given boolean predicate on system states. If the system state is legal, i.e., it satisfies the given predicate, then all computing elements must accept; if the system state is illegal, then at least one computing element must reject. Distributed property testing is a relaxed variant of distributed decision, which only requires distinguishing legal states from states that are "far from" being legal. (The notion of "farness" depends on the context.)

In the context of distributed network computing, one is interested in deciding or testing whether the actual network, modeled as a simple connected graph, satisfies some given predicate on graphs; e.g., bipartiteness, cycle-freeness, subgraph-freeness, etc. For a positive distance parameter $\varepsilon \leq 1$, a graph G with m edges is said to be ε-far from satisfying a given property P if removing and/or adding up to $\varepsilon\, m$ edges from/to G cannot result in a graph satisfying P.

In this paper we study distributed decision in general, and distributed property testing in particular, in the framework of distributed network computing, under the standard CONGEST model. This paper is the result of merging the three papers [16], [17], and [18] that were concurrently submitted to the 31st International Symposium on Distributed Computing (DISC 2017), which independently showed overlapping results, using different methods and ideas. To highlight the different approaches to the problem, we chose to present a short version of each of the three papers in the form of three notes.

The Subgraph-Freeness Problem. Each of the three notes presented here gives results on subgraph-freeness: we are given a constant-size graph H, and we wish to determine whether the network graph contains H as a subgraph or not. In the *property testing* relaxation of the problem, we only need to distinguish the case where the network graph is H-free from

the case where it is ε-*far* from H-free, in the sense that at least an ε-fraction of the graph's edges must be removed to eliminate all copies of H.

Hereafter, we provide a summary of the results and methods in each paper.

1.1 Summary of the Results and Techniques

Note #1: Color-Coding Based Algorithms for Testing Subgraph-Freeness

This note uses a technique called *color-coding* [4] to design randomized algorithms for property-testing subgraph-freeness in $O(1/\varepsilon)$ rounds, for any subgraph H that contains an edge (u, v) such that any cycle in H contains at least one of u and v. In the case of trees, the color-coding technique yields an $O(1)$-round algorithm for testing exactly whether the graph contains the given tree or not. In addition, for cliques K_s with $s \geq 3$, we show that K_s-freeness can be tested in $O(m^{\frac{1}{2}-\frac{1}{s-2}} \cdot \varepsilon^{-\frac{1}{2}-\frac{1}{s-2}})$ rounds, where m is the number of edges in the network graph. In the special case of triangles, K_3-freeness can actually be solved in $O(1)$ rounds in n-node networks, i.e., in a number of rounds independent from ε, assuming $\varepsilon \geq \min\{m^{-\frac{1}{3}}, \frac{n}{m}\}$.

Note #2: Deterministic Tree Detection and Applications in Distributed Property Testing

In this note, we propose a generic construction for designing deterministic distributed algorithms detecting the presence of any given tree T as a subgraph of the input network, performing in a constant number of rounds in the CONGEST model. Our construction also provides randomized algorithms for testing H-freeness for every graph pattern H that can be decomposed into an edge and a tree, with arbitrary connections between them, also running in $O(1)$ rounds in the CONGEST model. This generalizes the results in [9, 19, 20], where algorithms for testing K_3, K_4, and C_k-freeness for every $k \geq 3$ were provided.

Note #3: Algorithms for Testing and Correcting Graph Properties in the CONGEST Model

In Section 4, we design and analyze distributed testers in the distributed CONGEST model all of which work in the general model. Stating that our testers work in the general model means that our measure of distance between two graphs is measured by adding or removing $\varepsilon \cdot m$ edges.

Diameter dependency reduction and its Applications. In Section 4.2 we describe a general procedure for converting ε-testers with $f(D)$ rounds, where D denotes the diameter of the graph, to work in $O((\log n)/\varepsilon) + f((\log n)/\varepsilon)$ rounds, where n is the number of processors of the network. We then apply this procedure to obtain an ε-tester for testing whether a graph is bipartite. The improvement of this tester over state of the art is twofold: (a) the round complexity is $O(\varepsilon^{-1} \log n)$, which improves over the $\text{Poly}(\varepsilon^{-1} \log n)$-round algorithm by Censor-Hillel et al. [9, Thm. 5.2], and (b) our tester works in the general model while [9] works in the more restrictive bounded degree model. Moreover, the number of rounds of our bipartiteness tester meets the $\Omega(\log n)$ lower bound by [9, Thm. 7.3], hence our tester is asymptotically optimal in terms of n. We then apply this "compiler" to obtain a cycle-free tester with number of rounds of $O(\varepsilon^{-1} \cdot \log n)$, thus revisiting the result by [9, Thm. 6.3]. The last application that we consider is "how to obtain a *corrector* of the graph by using

this machinery?" Namely, how to design an algorithm that locally corrects the graph so that the corrected graph satisfies the property. For cycle-freeness, we are able to obtain also an ε-corrector. Note that, unlike a tester, a corrector needs to mend the graph in many places in the case that the graph is far from having the property.

Testers for H-freeness for $|V(H)| \leq 4$. In Section 4.3 we design algorithms for testing (in the general model) whether the network is H-free for any connected H of size up to four with round complexity of $O(\varepsilon^{-1})$.

Testers for tree-freeness. In Section 4.4, we first generalize the global tester by Iwama and Yoshida [25] of testing k-path freeness to testing the exclusion of any tree, T, of order k. Our tester has a one sided error and it works in the general graph model with random edge queries. This algorithm can be simulated in the CONGEST model in $O(k^{k^2+1} \cdot \varepsilon^{-k})$ rounds.

2 Note #1: Color-Coding Based Algorithms for Testing Subgraph-Freeness

2.1 Introduction

The aim of this part of the paper is to improve our understanding of the question: "which types of excluded subgraphs can be tested in constant time?". We also explore several related questions, such as whether limiting the maximum degree in the graphs helps (by analogy to the bounded-degree model in sequential property testing), whether we can test H-freeness in *sublinear* time for some subgraphs H for which no constant-time algorithm is known, and whether there are cases where we can test H-freeness with *no dependence on the distance parameter* ε, even when ε is sub-constant (e.g., $\varepsilon = O(1/\sqrt{n})$). Using new ideas and combining them with previous techniques, we are able to simplify and extend prior work, and point out some surprising answers to the questions above, which point to several aspects where distributed property testing for subgraph-freeness differs from the sequential analogue.

Our Results. First we give simple randomized algorithms for testing subgraph-freeness, using the *color-coding* technique from [4], originally used to find cycles of fixed constant size sequentially.

We begin by giving a color-coding based algorithm which tests k-cycle freeness in $O(1/\varepsilon)$ rounds (for constant k). Next we show that for any tree T, we can test T-freeness exactly (without the property-testing relaxation) in constant time. Both of the results extend to directed graphs in the directed version of the CONGESTmodel. Combining the two algorithms, we give a class of graphs $\mathcal{H}$ such that for any constant-sized $H \in \mathcal{H}$, we can test H-freeness in $O(1/\varepsilon)$ rounds. The class $\mathcal{H}$ consists of all graphs H containing an edge $\{u, v\}$ such that each cycle in H includes either u or v (or both). This includes all graphs of size 5 except for the 5-clique, K_5.

We then turn our attention to the special case of cliques. We present a different approach for detecting triangles, which allows us to *eliminate* the dependence of the running time on ε when ε is not too small: if n is the number of nodes and m is the number of edges, and we are promised that $\varepsilon \geq \min\left\{m^{-1/3}, n/m\right\}$, then we can test triangle-freeness in $O(1)$ rounds (independent of the actual value of ε). We extend this approach to cliques of any size $s \geq 3$, and show that K_s-freeness can be tested in $O\left(\varepsilon^{-1/2-1/(s-2)} m^{1/2-1/(s-2)}\right)$ rounds. In particular, for constant ε and $s = 5$, we can test K_5-freeness in $O(m^{1/6})$ rounds. We also

modify the algorithm to work in *constant time* in graphs whose maximum degree Δ is not too large with respect to the total number of edges, $\Delta = O((\varepsilon m)^{1/(s-2)})$.

2.2 Preliminaries

We generally work with undirected graphs, unless indicated otherwise. We let $N(v)$ denote the neighbors of v, and $d(v)$ the degree of v. Note that throughout the paper, when we use the term *subgraph*, we do not mean *induced subgraph*; we say that $G' = (V', E')$ is a subgraph of $G = (V, E)$ if $V' \subseteq V, E' \subseteq E$.

We say that a graph $G = (V, E)$ is *ε-far from property $\mathcal{P}$* if at least $\varepsilon|E|$ edges need to be added to or removed from E to obtain a graph satisfying $\mathcal{P}$.

The goal in distributed property testing for H-freeness is to solve the following problem: if the network graph G is H-free, then with probability $2/3$, all nodes should accept. On the other hand, if G is ε-far from being H-free, then with probability $2/3$, some node should reject.

We rely on the following fundamental property, which serves as the basis for most sequential property testers for H-freeness:

▶ **Property 1.** *Let G be ε-far from being H-free, then G has $\varepsilon m/|E(H)|$ edge-disjoint copies of H.*

Our algorithms assign random colors to vertices of the graph, and then look for a copy of the forbidden subgraph H which received the "correct colors". Formally we define:

▶ **Definition 2** (Properly-colored subgraphs)**.** Let $G = (V, E)$ and $H = ([k], F)$ be graphs, and let $G' = (V', E')$ be a subgraph of G that is isomorphic to H. We say that G' is *properly colored* with respect to a mapping $color_V : V \to [k]$ if there is an isomorphism $\varphi : V' \to [k]$ from G' to H such that for each $u \in V'$ we have $color_V(v) = \varphi(v)$.

2.3 Detecting Constant-Size Cycles

In this section we show that C_k-freeness can be tested in $O(1/\varepsilon)$ rounds in the CONGEST model for any constant integer $k > 2$.

▶ **Theorem 3.** *For any constant $k > 2$, there is a 1-sided error distributed algorithm for testing C_k-freeness which uses $O(1/\varepsilon)$ rounds.*

The key idea of the algorithm is to assign each node u of the graph a random color $color(u) \in [k]$. The node colors induce a coloring of both orientations of each edge, where $color(u, v) = (color(u), color(v))$. We discard all edges that are not colored $(i, (i+1) \bmod k)$ for some $i \in [k]$; this eliminates all cycles of size less than k, while preserving a constant fraction of k-cycles with high probability.

Next, we look for a properly-colored k-cycle by choosing a random directed edge (u_0, u_1)[1] and carrying out a k-round *color-coded BFS* from node u_0: in each round $r = 0, \ldots, k-1$, the BFS only explores edges colored $(r, (r+1) \bmod k)$. After k rounds, if the BFS reaches node u_0 again, then we have found a k-cycle.

Next we describe the implementation of the algorithm in more detail. We do not attempt to optimize the constants. To simplify the analysis, fix a set $\mathcal{C}$ of $\varepsilon m/k$ edge-disjoint k-cycles

[1] What we really want to do is choose a random node with probability proportional to its degree; choosing random edge is a simple way to do that.

(which we know exist if the graph is ε-far from being C_k-free). We abuse notation by also treating $\mathcal{C}$ as the set of edges participating in the cycles in $\mathcal{C}$.

For the analysis, it is helpful to think of the algorithm as first choosing a random edge and then choosing random colors, and this is the way we describe it below.

Choosing a Random Edge. It is not possible to get all nodes of the graph to explicitly agree on a uniformly random directed edge in constant time (unless the graph has constant diameter), but we can emulate the effect as follows: each node $u \in V$ chooses a uniformly random weight $w(e) \in [n^4]$ for each of its edges e. (Note that each edge has two weights, one for each of its orientations.) Implicitly, the directed edge we selected is the edge with the smallest weight in the graph, assuming that no two directed edges have the same weight.

▶ **Observation 4.** *With probability at least* $1 - 1/n^2$*, all weights in the graph are unique.*

Let $\mathcal{E}_U$ be the event that all edge weights are unique. Conditioned on $\mathcal{E}_U$, the directed edge with the smallest weight is uniformly random. Let e_0 be this edge; implicitly, e_0 is the edge we select. (However, nodes do not initially know which edge was selected, or even if a single edge was selected.)

Since the set $\mathcal{C}$ contains $\varepsilon m/k$ edge-disjoint k-cycles, and the graph has a total of m edges, we have:

▶ **Observation 5.** *We have* $\Pr[e_0 \in \mathcal{C} \mid \mathcal{E}_U] = \varepsilon$.

Let $\mathcal{E}_{Cyc}$ be the event that $e_0 \in \mathcal{C}$, and let $C_0 = \{u_0, u_1, \ldots, u_{k-1}\}$ be the cycle to which e_0 belongs given $\mathcal{E}_C$, where $e_0 = (u_0, u_1)$.

Color Coding. In order to eliminate cycles of length less than k, we assign to each node u a uniformly random color $color(u) \in [k]$. Node u then broadcasts $color(u)$ to its neighbors.

Since the colors are independent of the edge weights, we have:

▶ **Observation 6.** $\Pr[\forall i \in [k] \ : \ color(u_i) = i \mid \mathcal{E}_C, \mathcal{E}_U] = \frac{1}{k^k}$.

Let $\mathcal{E}_{Col}$ be the event that each u_i received color i. Combining our observations above yields:

▶ **Corollary 7.** $\Pr[\mathcal{E}_U \cap \mathcal{E}_{Cyc} \cap \mathcal{E}_{Col}] > \frac{\varepsilon}{2k^k}$.

Next we show thatn when $\mathcal{E}_U, \mathcal{E}_{Cyc}$ and $\mathcal{E}_{Col}$ all occur, we find a k-cycle.

Color-Coded BFS. Each node u stores the weight wgt_u associated with the lightest edge it has heard of so far, and the root $root_u$ of the BFS tree to which it currently belongs. Initially, wgt_u is set to the weight of the lightest of u's outgoing edges, and $root_u$ is set to u.

In each round $r = 0, \ldots, k-1$ of the BFS, nodes u with color r send $(u, wgt_u, root_u)$ to their neighbors, and nodes v with color $r+1$ update their state: if they received a message $(u, wgt_u, root_u)$ from a neighbor u, they set wgt_v to the lightest weight they received, and $root_v$ to the root associated with that weight.

After k rounds, if some node colored 0 receives a message $(v, wgt_v, root_v)$ where $root_v = u$, then it has found a k-cycle, and it rejects.

In Section 2.5, we will use the same C_k-freeness algorithms, but some nodes will be prohibited from taking certain colors. We incorporate this in Algorithm 1 by having some nodes whose state is abort. These nodes do not forward BFS messages and do not participate in the algorithm.

Algorithm 1: Procedure `ColorCodedBFS`, code for node u.

```
1  root ← u;
2  wgt ← min {w(u, v) | v ∈ N(u)};
3  for r = 0, ..., k − 1 do
4      if color = r and state ≠ abort then send (wgt, root) to neighbors ;
5      receive (w_1, r_1), ..., (w_t, r_t) from neighbors;
6      if color = (r + 1) mod k then
7          i ← argmin {w_1, ..., w_t};
8          if w_i < wgt then
9              root ← r_i, wgt ← w_i ;
10     if r = k − 1 and u ∈ {r_1, ..., r_t} then return 1 ;
11 return 0;
```

▶ **Lemma 8.** *If $\mathcal{E}_U, \mathcal{E}_{Cyc}$ and $\mathcal{E}_{Col}$ all occur, and if in addition the cycle C_0 contains no nodes whose state is* abort*, then u_0 returns 1 and Algorithm 1 finds a k-cycle (i.e., returns 1).*

Proof of Theorem 3. Suppose that G is ε-far from being C_k-free. We have no nodes whose state is abort (as we said, the abort state will be used in Section 2.5). Drawing random colors and weights, the probability that $\mathcal{E}_U, \mathcal{E}_{Cyc}$ and $\mathcal{E}_{Col}$ all occur is at least $\frac{\varepsilon}{2k^k}$; therefore, the probability that we fail to detect a k-cycle after $\lceil 20k^k/\varepsilon \rceil$ attempts is at most $1/10$. ◀

2.4 Detecting Constant-Size Trees

In this section we show that for any constant-size tree T, we can test T-freeness *exactly* (that is, without the property-testing relaxation) in $O(1)$ rounds. Let the nodes of T be $0, \ldots, k-1$. We arbitrarily assign node 0 to be the root of T, and orient the edges of the tree upwards toward node 0. Let R be the depth of the tree, that is, the maximum number of hops from any leaf of T to node 0. Finally, let $children(x)$ be the children of node x in the tree.

In the algorithm, we map each node of the network graph G onto a random node of T by assigning it a random color from $[k]$. Then we check if there is a copy of T in G that was mapped "correctly", with each node receiving the color of the vertex in T it corresponds to.

Initially the state of each node is "open" if it is an inner node of T, and "closed" if it is a leaf. The algorithm has R rounds, in each of which all nodes broadcast their state and their color to their neighbors. When a node with color j hears "closed" messages from nodes with colors matching all the children of node j in T, it changes its status to "closed". After R rounds, if node 0's state is "closed", we reject.

Let $x \in \{0, \ldots, k-1\}$ be a node of T, let T' be the sub-tree rooted at x, and let $G' = (U, E')$ be a subgraph of $G = (V, E)$ isomorphic to T'. We say that G' is *properly colored* if there is an isomorphism φ from G' to T', such that $color(u) = \varphi(u)$. (There may be more than one possible isomorphism from G' to T'.)

▶ **Lemma 9.** *Let u be a node with color $color(u) = x$, and let T' be the sub-tree of T rooted at x. Let h_x be the height of x, that is, the length of the longest path from a leaf of T' to x. Then at any time $t \geq h_x$ in the execution of Algorithm 2, we have $state_u(t) =$* closed *iff there is a subgraph G' containing u, which is isomorphic to T' and properly colored.*

Algorithm 2: Procedure `CheckTree`, code for node u.

```
1  if children(color) = ∅ then
2      state ← closed;
3  else
4      state ← open;
5  missing ← children(color);
6  for r = 1, ..., R do
7      send (color, state) to neighbors ;
8      receive (c_1, s_1), ..., (c_t, s_t) from neighbors;
9      foreach i = 1, ..., t do
10         if c_i ∈ missing and s_i = closed then
11             missing ← missing \ {c_i};
12             if missing = ∅ then state ← closed ;
13 if color = 0 and state = closed then
14     return 1;
15 else
16     return 0;
```

▶ **Corollary 10.** *For any node $u \in V$, at time h we have $\mathsf{state}_u(h) = \mathsf{closed}$ iff u is the root of a properly-colored copy of T.*

▶ **Corollary 11.** *If G contains a copy of T, then repeating Algorithm 2 yields an constant probability one sided-error algorithm for detecting a copy of T.*

Proof. Fix a subgraph G' which is isomorphic to T. Each time we pick a random coloring, the probability that G' is properly colored is at least $1/k^k$ (perhaps more, if there is more than one isomorphism mapping the nodes of G' to T). By Corollary 10, if G' is properly colored, the root of the tree will discover this and return 1. Therefore the probability that we fail $\lceil 10k^k \rceil$ times is at most $1/10$. ◀

2.5 Detecting Constant-Size Complex Graphs

In this section we define a class $\mathcal{H}$ of graphs, and give an algorithm for detecting those graphs in constant number of round (taking the size of the graph as a constant). The class $\mathcal{H}$ includes all graphs of size 5 except K_5 (see full paper [17]).

Definition of the Class $\mathcal{H}$. The class $\mathcal{H}$ contains all graphs that have the following property: there exists an edge (u, v) such that any cycle in the graph contain at least one of u and v. Equivalently, the class $\mathcal{H}$ contains all connected graphs that can be constructed using the following procedure:

1. We start with two nodes, 0 and 1, with an edge between them
2. Add any number of cycles $C_1, \ldots, C_\ell$ using new nodes, such that:
   - Each cycle C_i contains either node 0 or node 1 or both; and
   - With the exception of nodes $0, 1$, the cycles are node-disjoint.
3. Select a subset R of the nodes added so far, and for each node x selected, attach a tree T_x rooted at x using "fresh" nodes (that is, with the exception of node x, each tree T_x that

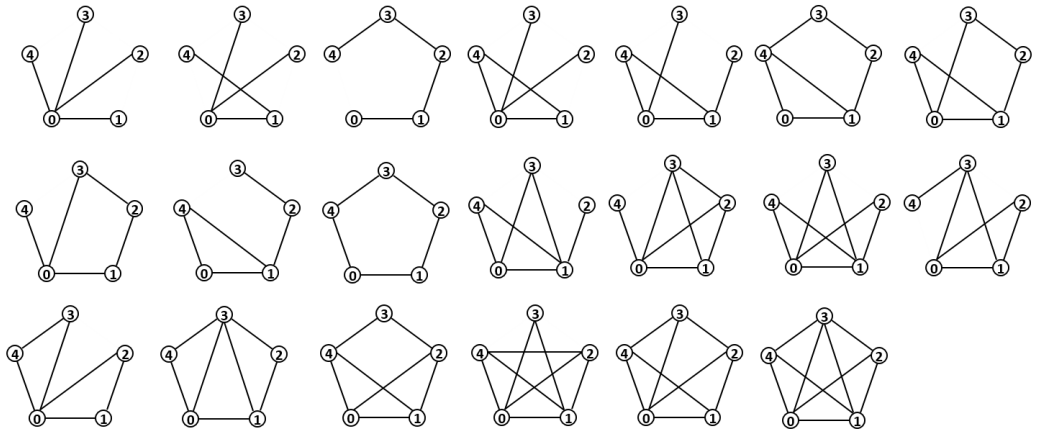


Figure 1 A good labeling for each connected graph over five vertices, except K_5.

we attach is node-disjoint from the graph constructed so far, including trees T_y added for other nodes $y \neq x$).

4. For each $x \in \{0,1\}$, add edges E_x between node x and some subset of nodes added in the previous steps.

The class $\mathcal{H}$ includes all connected nodes over five vertices, except the clique K_5. In Figure 1 we show a labeling consistent with the construction above (once the "correct" edge to label as $(0,1)$ is identified, the rest is easy to verify).

Our algorithm for testing H-freeness for $H \in \mathcal{H}$ combines the ideas from the previous sections. We begin by color-coding the nodes of G, mapping each node onto a random node of H. Next, we choose a random directed edge (u_0, u_1) from among the edges mapped to $(0,1)$, and begin the task of verifying that the various components of H are present and attached properly.

For simplicity, below we describe the verification process assuming that we really do choose a unique random edge, and all nodes know what it is; however, we cannot really do this, so we substitute using random edge weights as in Section 2.3.

1. Nodes u_0 and u_1 broadcast the chosen edge (u_0, u_1) for $\mathrm{diam}(H)$ rounds.
2. Any node whose color is 0 or 1 but which is not u_0 or u_1 (resp.) sets its state to abort.
3. For each edge $\{b, x\} \in E_b$, where $b \in \{0,1\}$, nodes colored x verify that they have an edge to node u_b; if they do not, they set their state to abort.
4. For each tree T_x added in stage 3 of the construction, we verify that a properly-colored copy of T_x is present, by having nodes colored x call Algorithm 2, with the colors replaced by the names of the nodes in T_x. We denote this by `CheckTree`(T_x).
   If a node colored x fails to detect a copy of T_x for which it is the root, it sets its state to abort for the rest of the current attempt.
5. For each $i = 1, \ldots, \ell$, we test for a properly-colored copy of C_i. We define the *owner* of C_i, denoted $owner(C_i)$, to be node 0 if C_0 contains 0, and otherwise node 1. To verify the presence of C_i, we call Algorithm 1, using the names of the nodes in C_i as colors: instead of color 0 we use $owner(C_i)$, and the remaining colors are mapped to the other nodes of C_i in order (in a arbitrary orientation of C_i). We denote this call by `ColorBFS`(C_i). (As indicated in Alg. 1, nodes whose state is abort do not participate.)
6. If both u_0 and u_1 are not in state abort, u_0 rejects, otherwise it accepts. All other nodes accept.

The analysis and pseudo-code of this algorithm appear in the full paper [17].

2.6 Testing K_s-Freeness

In previous sections it was shown how to test K_3 and K_4 freeness in $O(1/\varepsilon)$ rounds of communication. In this section we describe how to test K_s-freeness for cliques of any constant size s, in a sublinear number of rounds. Moreover, we show that triangle-freeness can be tested in $O(1)$ rounds, with no dependence on ε, when $\min\left(\frac{n}{m}, m^{-1/3}\right) \leq \varepsilon \leq 1$. Finally, we show that if the maximal degree is bounded by $O((\varepsilon m)^{\frac{1}{s-2}})$ then K_s-freeness can be tested in $O(1)$ rounds, but due to lack of space, this appears in the full paper version only [17].

2.6.1 Algorithm Overview

The basic idea is the following simple observation: suppose that each node u could learn the entire subgraph induced by $N(u)$, that is, node u knew for any two $v_1, v_2 \in N(u)$ whether they are neighbors or not. Then u could check if there is a set of s neighbors in $N(u)$ that are all connected to each other, and thus know if it participates in an s-clique or not. How can we leverage this observation?

For nodes u with high degree, we cannot afford to have u learn the entire subgraph induced by $N(u)$, as this requires of $N(u)^2$ bits of information. But fortunately, if G is ε-far from K_s-free, then there are many copies of K_s that contain some fairly low-degree nodes, as observed in [20]:

▶ **Lemma 12** ([20]). *Let $I(G)$ be the set of edges in some maximum set of edge-disjoint copies of H, and let $g(G) = \{(u,v) \mid d(u)d(v) \leq 2m|E(H)|/\varepsilon\}$. Then $|I(G) \cap g(G)| \geq \varepsilon m/(4|E(H)|)$.*

▶ Remark. [20] considers only subgraphs H with 4 vertices and constant ε, but their proof works for any subgraph H and any $0 < \varepsilon \leq 1$.

The focus in [20] is on *good edges*, which are edges satisfying the condition in Lemma 12, but here we need to focus on the *endpoints* of such edges. We call $u \in V$ a *good vertex* if its degree is at most $\sqrt{2m|E(H)|/\varepsilon}$, and we say that a copy of H in G is a *good copy* if it contains a good vertex. Since each copy of H in $I(G)$ contributes at most $|E(H)|$ edges to $g(G)$,

▶ **Corollary 13.** *If G is ε-far from H-free, then G contains at least $\varepsilon m/(4|E(H)|^2)$ edge-disjoint good copies of H.*

Because there are many good edge-disjoint copies of K_s, we can *sparsify* the graph and still retain at least one good copy of K_s.

We partition G into many edge-disjoint sparse subgraphs, by having each vertex u choose for each neighbor $v \in N(u)$ a random color $color(v) \in \{1, \ldots, C(u)\}$, where the size of the color range, $C(u)$, will be fixed later. This induces a partition of G's edges into $C(v)$ color classes; let $N_c(u)$ denote the set of neighbors $v \in N(u)$ with $color(v) = c$. The expected size of $N_c(u)$ is $d(v)/C(v)$.

With this partition in place, we begin by showing how to solve triangle-freeness in constant time, and then extend the algorithm to other cliques K_s with $s > 3$.

2.6.2 Triangle-Freeness for $\varepsilon \in [\min\left\{m^{-1/3}, n/m\right\}, 1]$ in $O(1)$ rounds

Assume that ε is not too small with respect to n and m: $\varepsilon \geq \min\left\{m^{-1/3}, n/m\right\}$. Then we can improve the algorithm from Section 2.3 and test triangle-freeness in *constant* time that does not depend on ε.

To test triangle-freeness, we set $C(v) = \lceil d(v)/200 \rceil$. Each node chooses a random color for each neighbor from the range $\{1, \ldots, C(v)\}$. Then, we go through the color classes $c = 1, \ldots, C(v)$ *in parallel*, and for each color class c, we look for a triangle containing two edges from $N_c(u)$: let $N_c(u) = \{v_1, \ldots, v_{t_c}\}$. for $R = 202e^2$ rounds $r = 1, \ldots, R$, node u sends v_r to all neighbors $v_1, \ldots, v_r$ in $N_c(u)$, and each neighbor v_i responds by telling u whether it is also connected to v_r, that is, whether $v_r \in N(v_i)$ (note that we do not insist on the edge (v_r, v_i) having color c). If $v_r \in N(v_i)$, then node u has found a triangle, and it rejects. If after $202e^2$ attempts node u has not found a triangle in any color class, it accepts.

▶ **Lemma 14.** *If G is ε-far from being triangle-free, then with probability $2/3$, at least one vertex detects a triangle.*

Proof. Let $\mathcal{T}$ be a set of edge-disjoint good triangles in G, of size $|\mathcal{T}| \geq \varepsilon m/(4|E(T)|^2) = \varepsilon m/36$. By Corollary 13 we know that there is such a set.

Assume that $\mathcal{T} = \{T_1, \ldots, T_t\}$. By definition, each good triangle has a *good vertex*; let v_i be a good vertex from the i'th triangle T_i.

For each $i \in \{1, \ldots, t\}$, let A_i be the event that v_i assigned the same color, c_i, to the other two vertices in T_i, and let X_i be an indicator for A_i. We have $\Pr[X_i = 1] = 1/C(v_i) = 200/d(v_i)$. Also, since the triangles in $\mathcal{T}$ are edge-disjoint, $X_1, \ldots, X_t$ are independent. Now let $X = \sum_{i=1}^{t} X_i$ be their sum; then

$$\Pr[X = 0] = \Pr[\bigcap_{i=1}^{t}(X_i = 0)] = \prod_{i=1}^{t}\left(1 - \frac{1}{C(v_i)}\right) = \prod_{i=1}^{t}\left(1 - \frac{200}{d(v_i)}\right).$$

We divide into two cases:

1. $m < n^{3/2}$: then $\varepsilon \geq \min\left(\frac{n}{m}, m^{-1/3}\right) = m^{-1/3}$. Recall v_i is a *good vertex*, which means $d(v_i) \leq \sqrt{6m/\varepsilon}$, and therefore

$$\prod_{i=1}^{t}\left(1 - \frac{200}{d(v_i)}\right) \leq \left(1 - \frac{200}{\sqrt{6m/\varepsilon}}\right)^t \leq e^{-\frac{200t}{\sqrt{6m/\varepsilon}}} \leq e^{-\frac{200\varepsilon m}{36\sqrt{6m/\varepsilon}}} = e^{-\frac{200\varepsilon^{3/2}\sqrt{m}}{36\sqrt{6}}} \leq e^{-2}.$$

2. $m \geq n^{3/2}$: then $\varepsilon \geq \min\left(\frac{n}{m}, m^{-1/3}\right) = \frac{n}{m}$. The degree of each vertex is no more than n, and hence

$$\prod_{i=1}^{t}\left(1 - \frac{200}{d(v_i)}\right) \leq \left(1 - \frac{200}{n}\right)^t \leq e^{-\frac{200t}{n}} \leq e^{-\frac{200\varepsilon m}{36n}} \leq e^{-2}.$$

So in any case we get $Pr[X = 0] \leq e^{-2}$.

Conditioned on $X \geq 1$, there is at least one vertex v_i which put two of its triangle neighbors in the same color class c_i, which means that if $N_c(v_i)$ is no larger than $200e^2$, node v_i will go through all neighbors in $N_c(v_i)$ and find the triangle. Because the colors of the edges are independent of each other, conditioning on A_i does not change the expected size of $N_{c_i}(v_i)$ by much: we know that the other two vertices in T_i received color c_i, but the remaining neighbors are assigned to a color class independently. The expected size of $|N_{c_i}(v_i)|$ is therefore $(d(v_i) - 2)/C(v_i) + 2 < 202 = R/e^2$, and by Markov, $\Pr[|N_{c_i}(v_i)| > R] \leq 1/e^2$.

To conclude, by union bound, the probability that no node v_i has $X_i = 1$, or that the smallest node v_i with $X_i = 1$ has $|N_c(v_j)| > 200e^2$ for the smallest color class c containing two triangle neighbors, is at most $1/e^2 + 1/e^2 < 1/3$. ◀

2.6.3 General Tester for K_s-Freeness

Use the same algorithm but with a different setting of the parameters, we can test K_s-freeness for any $s \geq 3$.

▶ **Theorem 15.** *There is a 1-sided error distributed property-testing algorithm for K_s-freeness, for any constant $s \geq 4$, with running time $O(\varepsilon^{\frac{-s}{2(s-2)}} m^{\frac{s-4}{2(s-2)}})$.*

▶ **Corollary 16.** *There is a 1-sided error distributed property-testing algorithm for K_5-freeness, with running time $O(m^{1/6})$.*

We set

$$C(u) = \left\lceil \left(\frac{1}{2s^4} \varepsilon m \right)^{\frac{1}{s-2}} \right\rceil$$

to be the number of color classes at node u, and

$$R = 2s^4 e^2 \left[\varepsilon^{-1/2-1/(s-2)} m^{1/2-1/(s-2)} + s - 1 \right]$$

to be the timeout. For R rounds, each node u sends the next node v_r from each color class to all neighbors $v_1, \ldots, v_{t_c}$ in that color class, and each neighbor v_i responds by telling u whether v_r is its neighbor or not. Node u remembers this information; if at any point it knows of a subset $S \subseteq N_c(u)$ of $|S| = s$ nodes that are all neighbors of each other, then it has found an s-clique, and it rejects. After R rounds u gives up and accepts.

▶ **Lemma 17.** *If G is ε-far from K_s-free, then with probability at least $2/3$, at least one vertex detects a copy of K_s.*

▶ Remark. For $s \geq 5$, the algorithm requires a linear estimate of m to get good running time. If m is unknown, then the vertices may run the algorithm $\log n$ times for exponentially-increasing guesses $m = [n, 2n, ...n^2]$, and as the protocol has one sided error, correctness is maintained; however, the running time increases to $O(\varepsilon^{\frac{-s}{2(s-2)}} n^{\frac{s-4}{(s-2)}})$ rounds.

3 Note #2: Deterministic Tree Detection and Applications in Distributed Property Testing

3.1 Context and Objective

Given a fixed graph H (e.g., a triangle, a clique on four nodes, etc.), a graph G is H-free if it does not contain H as a subgraph[2]. Detecting copies of H or deciding H-freeness has been investigated in many algorithmic frameworks, including classical sequential computing [2], parametrized complexity [32], streaming [8], property-testing [3], communication complexity [27], quantum computing [5], etc. In the context of distributed network computing, deciding H-freeness refers to the task in which the processing nodes of a network G must collectively detect whether H is a subgraph of G, according to the following decision rule:

[2] Recall that H is a subgraph of G if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$.

- if G is H-free then every node outputs accept;
- otherwise, at least one node outputs reject.

In other words, G is H-free if and only if all nodes output accept.

Recently, deciding H-freeness for various types of graph patterns H has received lots of attention (see, e.g., [9, 10, 12, 13, 26, 19, 20]) in the CONGEST model [36], and in variants of this model. (Recall that the CONGEST model is a popular model for analyzing the impact of limited link bandwidth on the ability to solve tasks efficiently in the context of distributed network computing). In particular, it has been observed that deciding H-freeness may require nodes to consume a lot of bandwidth, even for very simple graph patterns H. For instance, it has been shown in [13] that deciding C_4-freeness requires $\widetilde{\Omega}(\sqrt{n})$ rounds in n-node networks in the CONGEST model. Intuitively, the reason why so many rounds of computation are required to decide C_4-freeness is that the limited bandwidth capacity of the links prevents every node with high degree from sending the entire list of its neighbors through one link, unless consuming a lot of rounds. The lower bound $\widetilde{\Omega}(\sqrt{n})$ rounds for C_4-freeness can be extended to larger cycles C_k, $k \geq 4$, obtaining a lower bound of $\Omega(\text{poly}(n))$ rounds, where the exponent of the polynomial in n depends on k [13]. Hence, not only "global" tasks such as minimum-weight spanning tree [11, 28, 34], diameter [1, 21], and all-pairs shortest paths [24, 30, 33] are bandwidth demanding, but also "local" tasks such as deciding H-freeness are bandwidth demanding, at least for some graph patterns H.

In this note, we focus on a generic set of H-freeness decision tasks which includes several instances deserving full interest on their own right. In particular, deciding P_k-freeness, where P_k denotes the k-node path, is directly related to the NP-hard problem of computing the longest path in a graph. Also, detecting the presence of large complete binary trees, or of large binomial trees, is of interest for implementing classical techniques used in the design of efficient parallel algorithms (see, e.g., [29]). Similarly, detecting large Polytrees in a Bayesian network might be used to check fast belief propagation [35]. Finally, as it will be shown in this note, detecting the presence of various forms of trees can be used to tests the presence of graph patterns of interest in the framework of distributed property-testing [9]. Hence, this note addresses the following question:

> For which tree T is it possible to decide T-freeness efficiently in the CONGEST model, that is, in a number of rounds independent from the size n of the underlying network?

At a first glance, deciding T-freeness for some given tree T may look simpler than detecting cycles, or even just deciding C_4-freeness. Indeed, the absence of cycles enables to ignore the issue of checking that a path starts and ends at the *same* node, which is bandwidth consuming because it requires maintaining all possible partial solutions corresponding to growing paths from all starting nodes. When detecting cycles, discarding even just a few starting nodes may result in missing the unique cycle including these nodes. However, even deciding P_k-freeness requires to overcome many obstacles. First, as mentioned before, finding a longest simple path in a graph is NP-hard, which implies that it is unlikely that an algorithm deciding P_k-freeness exists in the CONGEST model, with running time polynomial in k at every node. Second, and more importantly, there exists potentially up to $\Theta(n^k)$ paths of length k in a network, which makes impossible to maintain all of them in partial solutions, as the overall bandwidth of n-node networks is at most $O(n^2 \log n)$ in the CONGEST model.

3.2 Our Results

We show that, in contrast to C_k-freeness, P_k-freeness can be decided in a constant number of rounds, for any $k \geq 1$. In fact, our main result is far more general, as it applies to *any* tree. Stated informally, we prove the following:

Theorem A. *For every tree T, there exists a deterministic algorithm for* deciding *T-freeness performing in a constant number of rounds under the* CONGEST *model.*

For establishing Theorem A, we present a distributed implementation of a pruning technique based on a combinatorial result due to Erdős et al. [15] that roughly states the following. Let $k > t > 0$. For any set V of n elements, and any collection F of subsets of V, all with cardinality at most t, let us define a *witness* of F as a collection $\widehat{F} \subseteq F$ of subsets of V such that, for any $X \subseteq V$ with $|X| \leq k - t$, the following holds:

$$\big(\exists Y \in F : Y \cap X = \emptyset\big) \implies \big(\exists \widehat{Y} \in \widehat{F} : \widehat{Y} \cap X = \emptyset\big).$$

Of course, every F is a witness of itself. However, Erdős et al. have shown that, for every k, t, and F, there exists a *compact* witness $\widehat{F}$ of F, that is, a witness whose cardinality depends on k and t only, and hence is independent of n. To see why this result is important for detecting a tree T in a network G, consider V as the set of nodes of G, k as the number of nodes in T, and F as a collection of subtrees Y of size at most t, each isomorphic to some subtree of T. The existence of compact witnesses allows an algorithm to keep track of only a small subset $\widehat{F}$ of F. Indeed, if F contains a partial solution Y that can be extended into a global solution isomorphic to T using a set of nodes X, then there is a *representative* $\widehat{Y} \in \widehat{F}$ of the partial solution $Y \in F$ that can also be extended into a global solution isomorphic to T using the same set X of nodes. Therefore, there is no need to keep track of all partial solutions $Y \in F$, it is sufficient to keep track of just the partial solutions $\widehat{Y} \in \widehat{F}$. This pruning technique has been successfully used for designing fixed-parameter tractable (FPT) algorithms for the longest path problem [32], as well as, recently, for searching cycles in the context of distributed property-testing [19]. Using this technique for detecting the presence of a given tree however requires to push the recent results in [19] much further. First, the detection algorithm in [19] is anchored at a fixed node, i.e., the question addressed in [19] is whether there is a cycle C_k passing through a given node. Instead, we address the detection problem in its full generality, and we do not restrict ourselves to detecting a copy of T including some specific node. Second, detecting trees requires to handle partial solutions that are not only composed of sets of nodes, but that offer various shapes, depending on the structure of the tree T, representing all possible combinations of subtrees of T.

Theorem A, which establishes the existence of distributed algorithms for detecting the presence of trees, has important consequences on the ability to *test* the presence of more complex graph patterns in the context of *distributed property-testing*. Recall that, for $\varepsilon \in (0, 1)$, a graph G is ε-far from being H-free if removing less than a fraction ε of its edges cannot result in an H-free graph. A (randomized) distributed algorithm *tests* H-freeness if it decides H-freeness according to the following decision rule:

- if G is H-free then Pr[every node outputs accept] $\geq 2/3$;
- if G is ε-far from being H-free then Pr[at least one node outputs reject] $\geq 2/3$.

That is, a testing algorithm separates graphs that are H-free from graphs that are far from being H-free. So far, the only non-trivial graph patterns H for which distributed algorithms testing H-freeness are known are:

- the complete graphs K_3 and K_4 (see [9, 20]), and
- the cycles C_k, $k \geq 3$ (see [19]).

Using our algorithm for detecting the presence of trees, we show the following (stated informally):

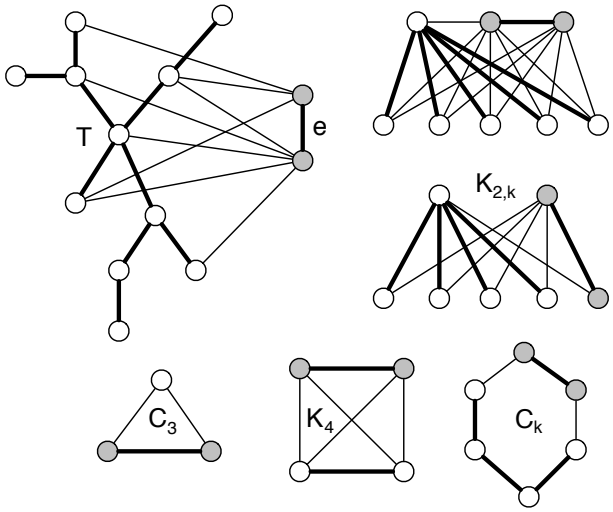


Figure 2 All these graphs are composed of a tree T and edge e with arbitrary connections between them.

Theorem B. *For every graph pattern H composed of an edge and a tree with arbitrary connections between them, there exists a (randomized) distributed algorithm for* testing *H-freeness performing in a constant number of rounds under the* CONGEST *model.*

At a first glance, the family of graph patterns H composed of an edge and a tree with arbitrary connections between them (like, e.g., the graph depicted on the top-left corner of Fig. 2) may look quite specific and artificial. This is not the case. For instance, every cycle C_k for $k \geq 3$ is a "tree plus one edge". This also holds for 4-node complete graph K_4. In fact, all known results about testing H-freeness for some graph H in [9, 19, 20] are just direct consequence of Theorem B. Moreover, Theorem B enables to test the presence of other graph patterns, like the complete bipartite graph $K_{2,k}$ with $k+2$ nodes, for every $k \geq 1$, or the graph pattern depicted on the top-right corner of Fig. 2, in $O(1)$ rounds. It also enables to test the presence of connected 1-factors as a subgraph in $O(1)$ rounds. (Recall that a graph H is a 1-factor if its edges can be directed so that every node has out-degree 1).

In fact, our algorithm is 1-sided, that is, if G is H-free, then all nodes output accept with probability 1.

3.3 Detecting the Presence of Trees

In this section, we establish our main result:

▶ **Theorem 18.** *For every tree T, there exists an algorithm performing in $O(1)$ rounds in the* CONGEST *model for detecting whether the given input network contains T as a subgraph.*

Proof. Let k be the number of nodes in tree T. The nodes of T are labeled arbitrarily by k distinct integers in $[1, k]$. We arbitrarily choose a vertex $r \in [1, k]$ of T, and view T as rooted in r. For any vertex $\ell \in V(T)$, let T_ℓ be the subtree of T rooted in ℓ. We say that T_ℓ is a *shape* of T. Our algorithm deciding T-freeness proceeds in $\text{depth}(T_r) + 1$ rounds. At round t, every node u of G constructs, for each shape T_ℓ of depth at most t, a set of subtrees of G all rooted at u, denoted by $\text{SOS}_u(T_\ell)$, such that each subtree in $\text{SOS}_u(T_\ell)$ is isomorphic to the shape T_ℓ. The isomorphism is considered in the sense of rooted trees, i.e., it maps u to ℓ. If

we were in the LOCAL model[3], we could afford to construct the set of all such subtrees of G. However, we cannot do that in the CONGEST model because there are too many such subtrees. Therefore, the algorithm acts in a way which guarantees that:

1. the set $\text{SOS}_u(T_\ell)$ is of constant size, for every node u of G, and every node ℓ of T;
2. for every set $C \subseteq V$ of size at most $k - |V(T_\ell)|$, if there is some subtree W of G rooted at u that is isomorphic to T_ℓ, and that is not intersecting C, then $\text{SOS}_u(T_l)$ contains at least one such subtree W' not intersecting C. (Note that W' might be different from W).

The intuition for the second condition is the following. Assume that there exists some subtree W of G rooted at u, corresponding to some shape T_ℓ, which can be extended into a subtree isomorphic to T by adding the vertices of a set C. The algorithm may well not keep the subtree W in $\text{SOS}_u(T_l)$. However, we systematically keep at least one subtree W' of G, also rooted at u and isomorphic to T_ℓ, that is also extendable to T by adding the vertices of C. Therefore the sets $\text{SOS}_u(T_\ell)$, over all shapes T_ℓ of depth at most t, are sufficient to ensure that the algorithm can detect a copy of T in G, if it exists. Our approach is described in Algorithm 3. (Observe that, in this algorithm, if we omit Lines 13 to 15, which prune the set $\text{SOS}_u(T_\ell)$, we obtain a trivial algorithm detecting T in the LOCAL model). Implementing the pruning of the sets $\text{SOS}_u(T_\ell)$ for keeping them compact, we make use of the following combinatorial lemma, which has been rediscovered several times, under various forms (see, e.g., [19, 32]).

▶ **Lemma 19** (Erdős, Hajnal, Moon [15]). *Let V be a set of size n, and consider two integer parameters p and q. For any set $F \subseteq \mathcal{P}(V)$ of subsets of size at most p of V, there exists a* compact (p, q)-representation *of F, i.e., a subset $\hat{F}$ of F satisfying:*

1. *For each set $C \subseteq V$ of size at most q, if there is a set $L \in F$ such that $L \cap C = \emptyset$, then there also exists $\hat{L} \in \hat{F}$ such that $\hat{L} \cap C = \emptyset$;*
2. *The cardinality of $\hat{F}$ is at most $\binom{p+q}{p}$, for any $n \geq p + q$.*

By Lemma 19, the sets $\text{SOS}_u(T_\ell)$ can be reduced to constant size (i.e., independent of n), for every shape T_ℓ and every node u of G. Moreover, the number of shapes is at most k, and, for each shape T_ℓ, each element of $\text{SOS}_u(T_\ell)$ can be encoded on $k \log n$ bits. Therefore each vertex communicates only $O(\log n)$ bits per round along each of its incident edges. So, the algorithm does perform in $O(1)$ rounds in the CONGEST model[4].

Proof of correctness. First, observe that if $\text{SOS}_u(T_\ell)$ contains a graph W, then W is indeed a tree rooted at u, and isomorphic to T_ℓ. This is indeed the case at round $t = 0$, and we can proceed by induction on t. Let T_ℓ be a shape of depth ℓ. Each graph W added to $\text{SOS}_u(T_\ell)$ is obtained by gluing vertex-disjoint trees at the root u. These latter trees are isomorphic to the shapes $T_{j_1}, \ldots, T_{j_s}$, where $j_1, \ldots, j_s$ are the children of node j in T. Therefore W is isomorphic to T_ℓ. In particular, if the algorithm rejects at some node u, it means that there exists a subtree of G isomorphic to T.

We now show that if G contains a subgraph W isomorphic to T, then the algorithm rejects in at least one node. For this purpose, we prove a stronger statement:

---

[3] The LOCAL model is similar to the CONGEST model, but it has no restriction on the size of the messages [36].

[4] We may assume that, for compacting a set $\text{SOS}_u(T_\ell)$ in Lines 13-15, every node u applies Lemma 19 by brute force (e.g., by testing all candidates $\hat{F}$). In [32], an algorithmic version of Lemma 19 is proposed, producing a set $\hat{F}$ of size at most $\sum_{i=1}^{q} p^i$ in time $O((p+q)! \cdot n^3)$, i.e., in time poly(n) for fixed p and q.

Algorithm 3: Tree-detection, for a given tree T. Algorithm executed by node u.

```
1  for each leaf ℓ of T do
2      let SOS_u(T_ℓ) be the unique tree with single vertex u
3      exchange the sets SOS with all neighbors
4  for t = 1 to depth(T) do
5      for each node ℓ of T with depth(T_ℓ) = t do
6          SOS_u(T_ℓ) ← ∅
7          let j_1, ..., j_s be the children of ℓ in T
8          for every s-uple (v_1, ..., v_s) of nodes in N(u) do
9              for every (W_1, ..., W_s) ∈ SOS_{v_1}(T_{j_1}) × ··· × SOS_{v_s}(T_{j_s}) do
10                 if {u} and W_1, ..., W_s are pairwise disjoint then
11                     let W be the tree with root u, and subtrees W_1, ..., W_s
12                     add W to SOS_u(T_ℓ) ;  // each W_i is glued to u by its root
13         let F = {V(W) | W ∈ SOS_u(T_l)}
           ;                  // collection of vertex sets for trees in SOS_u(T_l)
14         construct a (|V(T_ℓ)|, k − |V(T_ℓ)|)-compact representation F̂ ⊆ F
           ;                                                          // cf. Lemma 19
15         remove from SOS_u(T_ℓ) all trees W with vertex set not in F̂
16         exchange SOS_u(T_ℓ) with all neighbors
17 if SOS_u(T_r) = ∅ then ;                          // r denotes the root of T
18
19     return accept
20 else
21     return reject
```

▶ **Lemma 20.** *Let u be a node of G, T_ℓ be a shape of T, and C be a subset of vertices of G, with $|C| \leq k - |V(T_u)|$. Let us assume that there exists a subgraph W_u of G, satisfying the following two conditions:*
(1) *W_u is isomorphic to T_ℓ, and the isomorphism maps u on ℓ, and*
(2) *W_u does not contain any vertex of C.*
Then $\text{SOS}_u(T_\ell)$ contains a tree W'_u satisfying these two conditions.

We prove the lemma by induction on the depth of T_ℓ. If $\text{depth}(T_\ell) = 0$ then ℓ is a leaf of T_ℓ, and $\text{SOS}_u(T_\ell)$ just contains the tree formed by the unique vertex u. Il particular, it satisfies the claim. Assume now that the claim is true for any node of T whose subtree has depth at most $t-1$, and let ℓ be a node of depth t. Let $j_1, \ldots, j_s$ be the children of ℓ in T. For every i, $1 \leq i \leq s$, let v_i be the vertex of W_u mapped on j_i. By induction hypothesis, $\text{SOS}_{v_1}(T_{j_1})$ contains some tree W'_{v_1} isomorphic to T_{j_1} and avoiding the nodes in $C \cup \{u\}$, as well as all the nodes of $W_{v_2}, \ldots W_{v_s}$. Using the same arguments, we proceed by increasing values of $i = 2, \ldots, s$, and we choose a tree $W'_{v_i} \in \text{SOS}_{v_i}(T_{j_i})$ isomorphic to T_{j_i} that avoids $C \cup \{u\}$, as well as all the nodes in $W'_{v_1}, \ldots, W'_{v_{i-1}}$ and the nodes of $W_{v_{i+1}}, \ldots, W_{v_s}$. Now, observe that the tree W'' obtained from gluing u to $W'_{v_1}, \ldots, W'_{v_s}$ has been added to $\text{SOS}_u(T_\ell)$ before compacting this set, by Line 11 of Algorithm 3. Since W'' does not intersect C, we get that, by compacting the set $\text{SOS}_u(T_\ell)$ using Lemma 19, the algorithm keeps a representative subtree W' of G that is isomorphic to T_l and not intersecting C. This completes the proof of the lemma. ◇

To complete the proof of Theorem 18, let us assume there exists a subtree W of G isomorphic to T, and let u be the vertex that is mapped to the root r of T by this isomorphism. By Lemma 20, $\text{SOS}_u(T_r) \neq \emptyset$, and thus the algorithm rejects at node u. ◀

3.4 Distributed Property Testing

In this section, we show how to construct a distributed tester for H-freeness in the sparse model, based on Algorithm 3. This tester is able to test the presence of every graph pattern H composed of an edge e and a tree T connected in an arbitrary manner, by distinguishing graphs that include H from graphs that are ε-far[5] from being H-free.

Specifically, we consider the set $\mathcal{H}$ of all graph patterns H with node-set $V(H) = \{x, y, z_1, \ldots, z_k\}$ for $k \geq 1$, and edge-set $E(H) = \{f\} \cup E(T) \cup \mathcal{E}$, where $f = \{x, y\}$, T is a tree with node set $\{z_1, \ldots, z_k\}$, and $\mathcal{E}$ is some set of edges with one end-point equal to x or y, and the other end-point z_i for $i \in \{1, \ldots, k\}$. Hence, a graph $H \in \mathcal{H}$ can be described by a triple $(f, T, \mathcal{E})$ where $\mathcal{E}$ is a set of edges connecting a node in T with a node in f.

We now establish our second main result, i.e., Theorem B, stated formally below as follows:

▶ **Theorem 21.** *For every graph pattern $H \in \mathcal{H}$, i.e., composed of an edge and a tree connected in an arbitrary manner, there exists a randomized* 1*-sided error distributed property testing algorithm for H-freeness performing in $O(1/\varepsilon)$ rounds in the* CONGEST *model.*

Proof. Let $H = (f, T, \mathcal{E})$, with $f = \{x, y\}$. Let us assume that there are ν copies of H in G, and let us call these copies $H_1 = (f_1, T_1, \mathcal{E}_1), \ldots, H_\nu = (f_\nu, T_\nu, \mathcal{E}_\nu))$. Let $\boldsymbol{E} = \{f_1, \ldots, f_\nu\}$. Our tester algorithm for H-freeness is composed by the following two phases:

1. determine a candidate edge e susceptible to belong to $\boldsymbol{E}$;
2. checking the existence of a tree T connected to e in the desired way.

In order to find the candidate edge, we exploit the following lemma:

▶ **Lemma 22** ([20])**.** *Let H be any graph. Let G be an m-edge graph that is ε-far from being H-free. Then G contains at least $\varepsilon m/|E(H)|$ edge-disjoint copies of H.*

Hence, if the actual m-edge graph G is ε-far from being H-free, we have $|\boldsymbol{E}| \geq \varepsilon m/|E(H)|$. Thus, by randomly choosing an edge e and applying Lemma 22, $e \in \boldsymbol{E}$ with probability at least $\varepsilon/|E(H)|$.

As shown in [19], the first phase can be computed in the following way. First, every edge is assigned to the endpoint having the smallest identifier. Then, every node picks a random integer $r(e) \in [1, m^2]$ for each edge e assigned to it. The candidate edge of Phase 1 is the edge e_{min} with minimum rank, and indeed $\Pr[e_{min} \in \boldsymbol{E}] \geq \varepsilon/|E(H)|$.

It might be the case that e_{min} is not unique though. However: $\Pr[e_{min} \text{ is unique}] \geq 1/e^2$ where e denotes here the basis of the natural logarithm. Also, every node picks, for every edge $e = \{v_1, v_2\}$ assigned to it, a random bit b. Assume, w.l.o.g., that $\text{ID}(v_1) < \text{ID}(v_2)$. If $b = 0$, then the algorithm will start Phase 2 for testing the presence of H with $(x, y) = (v_1, v_2)$, and if $b = 1$, then the algorithm will start Phase 2 for testing the presence of H with $(x, y) = (v_2, v_1)$. We have $\Pr[e_{min} \text{ is considered in the right order}] \geq 1/2$. It follows that the probability e_{min} is unique, considered in the right order, and part of $\boldsymbol{E}$ is at least $\frac{\varepsilon}{2|E(H)|e^2}$.

[5] For $\varepsilon \in (0, 1)$, a graph G is ε-far from being H-free if removing less than a fraction ε of its edges cannot result in an H-free graph.

Using a deterministic search based on Algorithm 3, H will be found with probability at least $\frac{\varepsilon}{2|E(H)|e^2}$. To boost the probability of detecting H in a graph that is ε-far from being H-free, we repeat the search $2e^2|E(H)|\ln 3/\varepsilon$ times. In this way, the probability that H is detected in at least one search is at least $2/3$ as desired.

During the second phase, the ideal scenario would be that all the nodes of G search for $H = (f, T, \mathcal{E})$ by considering only the edge e_{min} as candidate for f, to avoid congestion. Obviously, making all nodes aware of e_{min} would require diameter time. However, there is no needs to do so. Indeed, the tree-detection algorithm used in the proof of Theorem 18 runs in $depth(T)$ rounds. Hence, since only the nodes at distance at most $depth(T)+1$ from the endpoints of e_{min} are able to detect T, it is enough to broadcast e_{min} at distance up to $2\,(depth(T)+1)$ rounds. This guarantees that all nodes participating to the execution of the algorithm for e_{min} will see the same messages, and will perform the same operations that they would perform by executing the algorithm for e_{min} on the full graph. So, every node broadcasts its candidate edge with the minimum rank, at distance $2\,(depth(T)+1)$. Two contending broadcasts, for two candidate edges e and e' for f, resolve contention by discarding the broadcast corresponding to the edge e or e' with largest rank. (If e and e' have the same rank, then both broadcast are discarded). After this is done, every node is assigned to one specific candidate edge, and starts searching for T. Similarly to the broadcast phase, two contending searches, for two candidate edges e and e', resolve contention by aborting the search corresponding to the edge e or e' with largest rank. From now on, one can assume that a single search in running, for the candidate edge e_{min}.

It remains to show how to adapt Algorithm 3 for checking the presence of a tree T connected to a *fixed* edge $e = \{x, y\} \in E(G)$ as specified in $\mathcal{E}$. Let us consider Instruction 5 of Algorithm 3, that is: "**for** each node ℓ of T with $\text{depth}(T_\ell) = t$ **do**". At each step of this for-loop, node u tries to construct a tree W that is isomorphic to the subtree of T rooted at ℓ. In order for u to add W to $\text{SOS}_u(T_\ell)$, we add the condition that:

- if $\{\ell, x\} \in E(H)$ then $\{u, x\} \in E(G)$, and
- if $\{\ell, y\} \in E(H)$ then $\{u, y\} \in E(G)$.

Note that this condition can be checked by every node u. If this condition is not satisfied, then u sets $\text{SOS}_u(T_\ell) = \emptyset$.

This modification enables to test H-freeness. Indeed, if the actual graph G is H-free, then, since at each step of the modified algorithm, the set $\text{SOS}_u(T_\ell)$ is a subset of the set $\text{SOS}_u(T_\ell)$ generated by the original algorithm, the acceptance of the modified algorithm is guaranteed from the correctness of the original algorithm.

Conversely, let us show that, in a graph G that is ε far of being H-free, the algorithm rejects G as desired. In the first phase of the algorithm, it holds that $e_{min} \in \boldsymbol{E}$ happens in at least one search whenever G is ε-far from being H-free, with probability at least $2/3$. Following the same reasoning of the proof of Lemma 20, since the images of the isomorphism satisfy the condition of being linked to nodes $\{x, y\}$ in the desired way, the node of G that is mapped to the root of T correctly detects T, and rejects, as desired. ◀

3.5 Conclusion

In this note, we have proposed a generic construction for designing deterministic distributed algorithms detecting the presence of any given tree T as a subgraph of the input network, performing in a constant number of rounds in the CONGEST model. Therefore, there is a clear dichotomy between cycles and trees, as far as efficiently solving H-freeness is concerned: while every cycle of at least four nodes requires at least a polynomial number of rounds to be detected, every tree can be detected in a constant number of rounds. It is not clear whether

one can provide a simple characterization of the graph patterns H for which H-freeness can be decided in $O(1)$ rounds in the CONGEST model. Indeed, the lower bound $\widetilde{\Omega}(\sqrt{n})$ for C_4-freeness can be extended to some graph patterns containing C_4 as induced subgraphs. However, the proof does not seem to be easily extendable to all such graph patterns as, in particular, the patterns containing many overlapping C_4 like, e.g., the 3-dimensional hypercube Q_3, since this case seems to require non-trivial extensions of the proof techniques in [13]. An intriguing question is to determine the round-complexity of deciding K_k-freeness in the CONGEST model for $k \geq 3$, and in particular to determine the exact round-complexity of deciding C_3-freeness.

Our construction also provides randomized algorithms for testing H-freeness (i.e., for distinguishing H-free graphs from graphs that are far from being H-free), for every graph pattern H that can be decomposed into an edge and a tree, with arbitrary connections between them, also running in $O(1)$ rounds in the CONGEST model. This generalizes the results in [9, 19, 20], where algorithms for testing K_3, K_4, and C_k-freeness for every $k \geq 3$ were provided. Interestingly, K_5 is the smallest graph pattern H for which it is not known whether testing H-freeness can be done in $O(1)$ rounds, and this is also the smallest graph pattern that cannot be decomposed into a tree plus an edge. We do not know whether this is just coincidental or not.

4 Note #3: Algorithms for Testing and Correcting Graph Properties in the CONGEST Model

4.1 Computational Models

Notations. Let $G = (V, E)$ denote a graph, were V is the set of vertices and E is the set of edges. Let $n \triangleq |V(G)|$, and let $m \triangleq |E(G)|$. For every $v \in V$, let $N_G(v) \triangleq \{u \in V \mid \{u, v\} \in E\}$ denote the neighborhood of v in G. For every $v \in V$, let $d_G(v) \triangleq |N_G(v)|$ denote the degree of v. When the graph at hand is clear from the context we omit the subscript G.

4.1.1 Distributed CONGEST Model

Computation in the distributed CONGEST [36] model is done as follows. Let $G = (V, E)$ denote a network where each vertex is a processor and each edge is a communication link between its two endpoints. Each processor is given a local input. Each processor v has a distinct ID - for brevity we say that the ID of processor v is simply v.[6] The computation is synchronized and is measured in terms of *rounds*. In each round, each processor performs the following steps:

1. Receive the messages that were sent by its neighbors in the previous round.
2. Execute a local (randomized) computation.
3. Sends (different) messages of $O(\log n)$ bits to every neighbor neighbors (or a possible "empty message").

In the last round all the processors stop and output a local output.

[6] In this paper we focus on randomized algorithms. Note that, with high probability, distinct IDs can be randomly generated using $O(\log n)$ bits.

4.1.2 (Global) Testing Model

Graph property testing [22, 23] is defined as follows. A graph property $\mathcal{P}$ is a subset of all (undirected and unlabeled) graphs e.g., the graph is cycle-free, the graph is bipartite, etc. We focus on edge monotone (with respect to deletions) properties.

▶ **Definition 23.** A graph property $\mathcal{P}$ is *edge-monotone* if $G \in \mathcal{P}$ and G' is obtained from G by the removal of edges, then $G' \in \mathcal{P}$.

We define the *edge-distance* between two graphs $G = (V, E)$ and $G' = (V, E')$ as the number edges in the symmetric difference $E \triangle E'$. We say that a graph G is *ε-far (in the general model)* from having the property $\mathcal{P}$ if $|E \triangle E'| \geq \varepsilon \cdot |E|$, for every $G' = (V, E') \in \mathcal{P}$.

The tester accesses the graph via queries. The type of queries we consider are: (1) what is the degree of v for $v \in V$? (2) who is the ith neighbor of $v \in V$?

We say that an algorithm is a *one sided ε-tester for property $\mathcal{P}$ in the general model* if given query access to the graph G the algorithm ACCEPTS the graph G if G has the property $\mathcal{P}$, i.e, completeness, and REJECTS the graph G with probability at least $2/3$ if G is ε-far from having the property $\mathcal{P}$, i.e., soundness.

We note that since the tester must accept graphs $G \in \mathcal{P}$, a reject occurs if only if the tester has a proof that that $G \notin \mathcal{P}$. Such a proof is called a *witness* against $G \in \mathcal{P}$. In fact, in [9], it is required that the witness is an induced proper subgraph of G.

The complexity measure of this model is the number of queries made to G. The goal is to design an ε-tester with as few as possible queries. In particular, the number of queries should be sublinear in the size of the graph.

In Section 4.4 an additional query type is allowed; this query is called a *random edge query*, and enables on to pick an edge e u.a.r. from E.

4.1.3 Distributed Testing in the CONGEST model

Let $G = (V, E)$ be a graph and let $\mathcal{P}$ denote a graph property. We say that a randomized distributed CONGEST algorithm is an *ε-tester for property $\mathcal{P}$ in the general model* [9] if when G has the property $\mathcal{P}$ then *all* the processors $v \in V$ output ACCEPT, and if G is ε-far from having the property $\mathcal{P}$, then there is a processor $v \in V$ that outputs REJECT with probability at least $2/3$.

4.1.4 Distributed Correcting

In this section we define correction in the distributed setting. We then explain how to obtain correction for the property of cycle-freeness. We focus here on edge-monotone properties, and therefore, consider only corrections that delete edges. One can view an ε-corrector as an approximation algorithm to the distance to property $\mathcal{P}$, where the approximation is additive.

▶ **Definition 24.** In the distributed CONGEST model, we say that an algorithm is an ε-corrector for an edge-monotone property $\mathcal{P}$ if the following holds.

1. Let $G = (V, E)$ denote the network's graph. When the algorithm terminates, each processor v knows which edges in E that intersect with v are in the set of *deleted edges* $E' \subseteq E$.
2. $G(V, E \setminus E')$ is in $\mathcal{P}$.
3. $|E'| \leq dist(G, \mathcal{P}) + \varepsilon|E|$, where $dist(G, \mathcal{P})$ denotes the minimum number of edges that should be removed from G in order to obtain the property $\mathcal{P}$.

4.2 Reducing the Dependency on the Diameter and Applications

In this section we present a general technique that reduces the dependency of the round complexity on the diameter. The technique is based on graph decompositions defined below.

▶ **Definition 25** ([31]). Let $G = (V, E)$ denote an undirected graph. A (β, d)-decomposition of G is a partition of V into disjoint subsets $V_1, \ldots, V_k$ such that

1. For all $1 \leq i \leq k$, $\mathrm{diam}(G[V_i]) \leq d$, where $G[V_i]$ is the vertex induced subgraph of of G that is induced by V_i.
2. The number of edges with endpoints belonging to different subsets is at most $\beta \cdot |E|$. We refer to these as *cut-edges* of the decomposition.

Note that the diameter constraint refers to strong diameter, in particular, each induced subgraph $G[V_i]$ must be connected.

Algorithms for $(\varepsilon, (\log n)/\varepsilon)$-decompositions were developed in many contexts (e.g., parallel algorithms [6, 7, 31]). An implementation in the CONGEST-model is presented in [14].

▶ **Theorem 26** ([14]). *A $(\varepsilon, O(\log n/\varepsilon))$-decomposition can be computed in the randomized* CONGEST*-model in $O((\log n)/\varepsilon)$ rounds with probability at least $1 - 1/\mathrm{Poly}(n)$.*

A nice feature of the algorithm based on random exponential shifts is that at the end of the algorithm, there is a spanning BFS-like rooted tree T_i for each subset V_i in the decomposition. Moreover, each vertex $v \in V_i$ knows the center of T_i as well as its parent in T_i. In addition, every vertex knows which of the edges incident to it are cut-edges.

The following definition captures the notion of connected witnesses against a graph satisfying a property.

▶ **Definition 27** ([9]). [7] A graph property $\mathcal{P}$ is *non-disjointed* if for every witness G' against $G \in \mathcal{P}$, there exists an induced subgraph G'' of G' that is connected such that G'' is also a witness against $G \in \mathcal{P}$.

The main result of this section is formulated in the following theorem. We refer to a distributed algorithm in which all vertices accept iff $G \in \mathcal{P}$ as a *verifier* for $\mathcal{P}$.

▶ **Theorem 28.** *Let $\mathcal{P}$ be an edge-monotone non-disjointed graph property that can be verified in the* CONGEST*-model in $O(\mathrm{diam}(G))$ rounds, where G is the input graph. Then there is an ε-tester for $\mathcal{P}$ in the randomized* CONGEST*-model with $O((\log n)/\varepsilon)$ rounds.*

Proof. The algorithm tries to "fix" the input graph G so that it satisfies $\mathcal{P}$ by removing less than $\varepsilon \cdot m$ edges. The algorithm consists of two phases. In the first phase, an $(\varepsilon', O((\log n)/\varepsilon'))$ decomposition is computed in $O((\log n)/\varepsilon')$ rounds, for $\varepsilon' = \varepsilon/2$. The algorithm removes all the cut-edges of the decomposition. (There are at most $\varepsilon \cdot m/2$ such edges.) In the second phase, in each subgraph $G[V_i]$, an independent execution of the verifier algorithm for $\mathcal{P}$ is executed. The number of rounds of the verifier in $G[V_i]$ is $O(\mathrm{diam}(G[V_i])) = O((\log n)/\varepsilon)$.

We first prove completeness. Assume that $G \in \mathcal{P}$. Since $\mathcal{P}$ is an edge-monotone property, the deletion of the cut-edges does not introduce a witness against $\mathcal{P}$. This implies that each

[7] An alternative (nonequivalent) definition which suffices for proving Theorem 28 is as follows. A property $\mathcal{P}$ is non-disjointed if, for every nonconnected graph G, the following holds:

$$G \in \mathcal{P} \iff \text{for every connected component } G' \text{ of } G\text{: } G' \in \mathcal{P}.$$

induced subgraph $G[V_i]$ does not contain a witness against $\mathcal{P}$, and hence the verifiers do not reject, and every vertex accepts.

We now prove soundness. If G is ε-far from $\mathcal{P}$, then after the removal of the cut-edges (at most $\varepsilon m/2$ edges) property $\mathcal{P}$ is still not satisfied. Let G' be a witness against the remaining graph satisfying $\mathcal{P}$. Since property $\mathcal{P}$ is non-disjointed, there exists a connected witness G'' in the remaining graph. This witness is contained in one of the subgraphs $G[V_i]$, and therefore, the verifier that is executed in $G[V_i]$ will reject, hence at least one vertex rejects, as required. ◀

We remark that if the round complexity of the verifier is $f(\text{diam}(G), n)$ (e.g., $f(\Delta, n) = \Delta + \log n$), then the round complexity of the ε-tester is $O((\log n)/\varepsilon) + f((\log n)/\varepsilon, n)$. This follows directly from the proof.

Extensions to ε-Testers

The following "bootstrapping" technique can be applied. If there exists an ε-tester in the CONGEST-model with round complexity $O(\text{diam}(G))$, then there exists an ε-tester with round complexity $O((\log n)/\varepsilon)$. The proof is along the same lines, expect that instead of a verifier, an ε'-tester is executed in each subgraph $G[V_i]$. Indeed, if G is ε-far from $\mathcal{P}$, then there must exist a subset V_i such that $G[V_i]$ is ε'-far from $\mathcal{P}$. Otherwise, we could "fix" all the parts by deleting at most $\varepsilon' \cdot m$ edges, and thus "fix" G by deleting at most $2\varepsilon' \cdot m = \varepsilon m$ edges, a contradiction.

4.2.1 Testing Bipartiteness

Theorem 28 can be used to test whether a graph is bipartite or ε-far from being bipartite. A verifier for bipartiteness can be obtained by attempting to 2-color the vertices (e.g., BFS that assigns alternating colors to layers). In our special case, each subgraph $G[V_i]$ has a root which is the only vertex that initiates the BFS. In the general case, one would need to deal with "collisions" between searches, and how one search "kills" the other searches initiated by vertices of lower ID.

4.2.2 Testing Cycle-Freeness

Theorem 28 can be used to test whether a graph is acyclic or ε-far from being acyclic. As in the case of bipartiteness, any scan (e.g., DFS, BFS) can be applied. A second visit to a vertex indicates a cycle, in which case the vertex rejects.

▶ **Corollary 29.** *There exists an ε-tester in the randomized* CONGEST*-model for bipartiteness and cycle-freeness with round complexity $O((\log n)/\varepsilon)$.*

4.2.3 Corrector for Cycle-Freeness

Our ε-testers for testing cycle freeness can be easily converted into ε-correctors as follows:: (1) All the cut-edges are removed. (2) In each $G[V_i]$, all the edges which are not in the BFS-like spanning tree T_i are removed.

▶ **Theorem 30.** *There exists an ε-corrector for cycle-freeness in the randomized* CONGEST*-model with round complexity $O((\log n)/\varepsilon)$.*

Proof sketch. The remaining edges form a forest of disjoint trees, and are therefore acyclic. The proof that the number of deleted edges is at most $\text{dist}(G, \mathcal{P}) + \varepsilon \cdot |\mathcal{E}|$ is based on the following two observations. Let G' denote the graph obtained from G by deleting all the cut-edges. $\text{dist}(G', \mathcal{P}) \leq \text{dist}(G, \mathcal{P})$ and $\text{dist}(G'[V_i], \mathcal{P}) = |E(G[V_i]) \setminus E(T_i)|$. ◀

4.3 Testing H-Freeness in $\Theta(1/\varepsilon)$ Rounds for $|V(H)| \leq 4$

4.3.1 Testing Triangle-Freeness

In this section we present an ε-tester for triangle-freeness that works in the CONGEST-model. The number of rounds is $O(1/\varepsilon)$.

Consider a triangle ABC in the input graph $G = (V, E)$. This triangle can be detected if A tells B about a neighbor $C \in N(A)$ with the hope that C is also a neighbor of B. Vertex B checks that C is also its neighbor, and if it is then the triangle ABC is detected. Hence, A would like to send to B the name of a vertex C such that $C \in N(A) \cap N(B)$. Since A can discover $N(A)$ in a single round, it proceeds by telling B about a neighbor $C \in N(A) \setminus \{B\}$ chosen uniformly at random. Let $M_{A \to B}$ denote the random neighbor that A reports to B. A listing of the distributed ε-tester for triangle-freeness appears as Algorithm 4. Note that all the messages $\{M_{A \to B}\}_{(A,B) \in E}$ are independent, and that the messages are rechosen for each iteration.

▶ **Claim 31.** *For every triangle x, the probability that triangle x is detected is at least $1/m$.*

Proof. Label the vertices of x arbitrarily by A, B, C. The event that triangle x is detected is contained in the event that $M_{A \to B} \in N(B)$. Since ABC is a triangle, $C \in N(A) \cap N(B)$, and $\Pr[M_{A \to B} \in N(B)] \geq 1/d(A) \geq 1/m$. ◀

▶ **Claim 32.** *If a graph G is ε-far from being triangle-free, then it contains at least $\varepsilon \cdot m/3$ edge-disjoint triangles.*

Proof. Consider the following procedure for "covering" all the triangles: while the graph contains a triangle, delete all three edges of the triangle. When the procedure ends, the remaining graph is triangle-free, hence at least εm edges were removed. The set of deleted triangles is edge disjoint and hence contains at least $\varepsilon m/3$ triangles. ◀

▶ **Theorem 33.** *Algorithm 4 is an ε-tester for triangle-freeness.*

Proof. *Completeness:* If G is triangle free then Line 4 is never satisfied, hence for every v Algorithm 4 terminates at Line 5.

Soundness: Let $G = (V, E)$ be a graph which is ε-far from being triangle free. By Claim 32 there are $\varepsilon \cdot m/3$ edge disjoint triangles in G. Edge disjointness implies that the detection of these triangles are independent events[8]. Hence, the probability of not detecting any of these triangles in a single iteration is at most $(1 - 1/m)^{\varepsilon m/3}$. The reject probability is amplified to $2/3$ by setting the number of iterations to be $t = \Theta(1/\varepsilon)$. ◀

4.3.2 Testing C_4-Freeness in $\Theta(1/\varepsilon)$ Rounds

In this section we present an ε-tester in the CONGEST-model for C_4-freeness that runs in $O(1/\varepsilon)$ rounds.

[8] In fact, the events are independent even for triangles which are not edge disjoint.

Algorithm 4: Triangle-free-test(v).

```
1 Send v to all u ∈ N(v)                          // 1st round: each v learns N(v)
2 for t ≜ Θ(1/ε) times do
3     For all u ∈ N(v), simultaneously: send u the message M_{v→u} ~ U(N(v) \ {u}).
4     If ∃w ∈ N(v) such that M_{w→v} ∈ N(v) then return REJECT
5 return ACCEPT
```

Uniform Sampling of 2-paths

Let $P_2(v)$ denote the set of all paths of length 2 that start at v. The algorithm is based on the ability of each vertex v to uniformly sample a path from $P_2(v)$. How many paths in $P_2(v)$ start with the edge (v, w)? Clearly, there are $(d(w) - 1)$ such paths. Hence the first edge should be chosen according to the degree distribution over $N(v)$ defined by $\pi^v(w) \triangleq (d(w) - 1)/\sum_{x \in N(v)}(d(x) - 1)$. Moreover, for each $x \in N(w) \setminus \{v\}$, the (directed) edge (w, x) appears exactly once as the second edge of a path in $P_2(v)$. Hence, given the first edge, the second edge is chosen uniformly.

This implies that v can pick a random path $p \in P_2(v)$ as follows: (1) Each neighbor $w \in N(v)$ sends v its degree and a uniformly randomly chosen neighbor $B_v(w) \in N(w) \setminus \{v\}$. The edge $(w, B_v(w))$ is a candidate edge for the second edge of p. (2) v picks a neighbor $A(v) \in N(v)$ where $A(v) \sim \pi^v$. The random path p is $p = \langle v, A(v), B_v(A(v))\rangle$, and it is uniformly distributed over $P_2(v)$.

In the algorithm, vertex v reports a path to each neighbor. We denote by $p_u(v)$ the path in $P_2(v)$ that v reports to $u \in N(v)$. This is done by independently picking neighbors $A_u(v) \in N(v)$, where each $A_u(v) \sim \pi^v$. Hence, the path that v reports to u is $p_u(v) \triangleq \langle v, A_u(v), B_v(A_u(v))\rangle$ Algorithm 5 uses this process for reporting paths of length 2. Interestingly, these paths are not independent, however for the case of edge disjoint copies of C_4, their "usefulness" in detecting copies of C_4 turns out to be independent (see Lemma 35).

Detecting a Cycle

Consider a copy $C = (v, w, x, u)$ of C_4 in G. If the 2-path $p_u(v)$ that v reports to u is $p_u(v) = (v, w, x)$, then u can check whether the last vertex x in $p_u(v)$ is also in $N(u)$. If $x \in N(u)$, then the copy C in G of C_4 is detected. (The vertex u also needs to verify that $w \neq u$.)

Description of the Algorithm

The ε-tester for C_4-freeness is listed as Algorithm 5. In the first round, each vertex v learns its neighborhood $N(v)$ and the degree of each neighbor. The for-loop repeats $t = O(1/\varepsilon)$ times. Each iteration consists of three rounds. In the first round, v independently draws fresh values for $A_u(v)$ and $B_u(v)$ for each of its neighbors $u \in N(v)$, and sends $B_u(v)$ to u. In the second round, for each neighbor $u \in N(v)$, v sends the path $\langle v, A_u(v), B_v(A_u(v))\rangle$. In the third round, v checks if it received a path $\langle w, a, b\rangle$ for a neighbor $w \in N(v)$ where $a \neq v$ and $b \in N(v)$. If this occurs, then (v, w, a, b) is a copy of C_4, and vertex v rejects. If v did not reject in all the iterations, then it finally accepts.

Analysis of the Algorithm

▶ **Definition 34.** We say that $p_u(v)$ is a *success* (wrt $C = (v, w, x, u)$) if $p_u(v) = (v, w, x)$. Let $I_{v,u}$ denote the indicator variable of the event that $p_u(v)$ is a success.

▶ **Lemma 35.** *Let $\{C^j(v_j, w_j, x_j, u_j)\}_{j\in J}$ denote a set of edge-disjoint copies of C_4 in G. Then the random variables I_{v_j,u_j} are independent.*

Proof. The event $I_{v,u} = 1$ occurs iff $A_u(v) = w$ and $B_v(w) = x$. Both $A_u(v)$ and $B_v(w)$ are random variables assigned to (directed) edges. By construction, all the random variables $\{A_u(v)\}_{(u,v\in E} \cup \{B_v(w)\}_{(v,w)\in E}$ are independent. Since the cycles are edge-disjoint, the lemma follows. ◀

▶ **Claim 36.** $\Pr[I_{v,u} = 1 \mid C] \geq 1/(2m)$.

Proof. The path $p_u(v)$ equals (v, w, x) iff $A_u(v) = w$ and $B_v(w) = x$. As $A_u(v)$ and $B_v(w)$ are independent, we obtain

$$\begin{aligned}\Pr\left[I_{v,u} = 1 \mid C\right] &= \Pr\left[A_u(v) = w|C\right] \cdot \Pr\left[B_v(w) = x \mid C\right] \\ &= \frac{d(w) - 1}{\sum_{x\in N(v)}(d(x) - 1)} \cdot \frac{1}{d(w) - 1} \geq \frac{1}{2m}.\end{aligned}$$ ◀

▶ **Claim 37.** *If a graph G is ε-far from being C_4-free, then it contains at least $\varepsilon \cdot m/4$ edge-disjoint copies of C_4.*

▶ **Theorem 38.** *Algorithm 5 is an ε-tester for C_4-freeness. The round complexity of the algorithm is $\Theta(1/\varepsilon)$ and in each round no more than $O(\log n)$ bits are communicated along each edge.*

Proof. *Completeness:* If G is C_4-free then Line 7 is never satisfied, hence for every v Algorithm 5 terminates at Line 8.

Soundness: Let $G = (V, E)$ be a graph which is ε-far from being C_4-free. Therefore, there exist $\ell \triangleq \varepsilon m/4$ edge disjoint copies of C_4 in G. Denote these copies by $\{C^1, \ldots, C^\ell\}$, where $C^j = (v_j, w_j, w_j, u_j)$. In each iteration, the cycle C^j is detected if $I_{v_j,u_j} = 1$, which (by Claim 36) occurs with probability at least $1/(2m)$. The cycles $\{C^j\}_j$ are edge-disjoint, hence, by Lemma 35, the probability that none of these cycles is detected is at at most $(1-1/(2m))^\ell$. The iterations are independent, and hence the probability that all the iterations fail to detect one of these cycles is at most $(1 - 1/(2m))^{\ell\cdot t}$. Since $\ell = \varepsilon m/4$, setting $t = \Theta(1/\varepsilon)$ reduces the probability of false accept to at most $1/3$, as required. ◀

Extending Algorithm 5

The algorithm can be easily extended to test H-freeness for any connected H over four nodes. If H is a $K_{1,3}$ then clearly H-freeness can be tested in one round. Otherwise, H is Hamiltonian and can be tested by simply sending an additional bit in the message sent in Line 5 of the algorithm. The additional bit indicates whether v is connected to $B_v(A_u(v))$. Given this information, u can determine the subgraph induced on $\{u, v, A_u(v), B_v(A_u(v))\}$, and hence rejects if H is a subgraph of this induced subgraph. Therefore we obtain the following theorem.

▶ **Theorem 39.** *There is an algorithm which is an ε-tester for H-freeness for any connected H over 4 vertices. The round complexity of the algorithm is $\Theta(1/\varepsilon)$ and in each round no more than $O(\log n)$ bits are communicated along each edge.*

Algorithm 5: C_4-free-test(v).

```
1 Send v and d(v) to all u ∈ N(v)          // v learns N(v) and d(u) for every u ∈ N(v)
2 Define the following distribution π^v over N(v): For every w ∈ N(v),
    π^v(w) ≜ d(w)/∑_{x∈N(v)} d(x) .
3 for t ≜ Θ(1/ε) times do
4     For every neighbor u ∈ N(v) independently draw A_u(v) ∼ π^v and
        B_u(v) ∼ U(N(v) \ {u}, send B_u(v) to u.
5     For every neighbor u ∈ N(v) send the path ⟨v, A_u(v), B_v(A_u(v))⟩ to u.
6     if ∃w ∈ N(v) s.t. v received the path ⟨w, a, b⟩ from w, where v ≠ a and b ∈ N(v) then
7         return REJECT                    // A cycle C = (v, w, a, b) was found.
8 return ACCEPT
```

4.4 Testing T-Freeness for any Tree T

In this section we generalize the tester by Iwama and Yoshida [25] of testing k-path freeness to testing the exclusion of any tree, T, of order k. We assume that the vertices of T are labeled by $v_0, \ldots, v_{k-1}$. Our tester has a one sided error and it works in the general graph model with random edge queries. This algorithm can be simulated in the CONGEST model. The complexities of the algorithms are stated in the next theorems.

▶ **Theorem 40.** *Algorithm 6 is a global ε-tester, one-sided error for T-freeness. The query complexity of the algorithm is $O\left(k^{k^2+1} \cdot \varepsilon^{-k}\right)$. The algorithm works in the general graph model augmented with random edge samples.*

▶ **Theorem 41.** *There is an ε-tester in the CONGEST model that on input T, where T is a tree, tests T-freeness. The round complexity of the tester is $O\left(k^{k^2+1} \cdot \varepsilon^{-k}\right)$ where k is the order of T.*

Global Algorithm Description

The algorithm by Iwama and Yoshida [25] for testing k-path freeness proceeds as follows. An edge is picked u.a.r. and an endpoint, v, of the selected edge,is picked u.a.r. A random walk of length k is performed from v, if a simple path of length k is found then the algorithm rejects. The analysis in [25] shows that this process has a constant probability (depends only on k and ε) to find a k-path in an ε-far from k-path freeness graph.

We generalize this tester in the following straightforward manner. We pick a random vertex v as in the above-mentioned algorithm. The vertex v is a candidate for being the root of a copy of T. For the sake of brevity we denote the (possible) root of the copy of T also by v_0. From v we start a "DFS-like" revealing of a tree which is a possible copy of T with the first random vertex acting as its root. DFS-like means that we scan a subgraph of G starting from v as follows: the algorithm independently and randomly selects $d_T(v_0)$ neighbors (out of the possible $d_G(v)$) and recursively scans the graph from each of these randomly chosen neighbors. While scanning, if we encounter any vertex more than once then we abort the process (we did not find a copy of T). If the process terminates, then this implies that the algorithm found a copy of T. In order to obtain probability of success of 2/3 the above process is repeated $t = f(\varepsilon, k)$ times. The listing of this algorithm appears in Algorithm 6. The algorithm can be simulated in the CONGEST model in a straight-forward way. The proofs of Theorems 40 and 41 appear in the full version of this paper [16].

Algorithm 6: Global-tree-free-test(T, v).

for $t \triangleq \Theta(k^{k^2}/\varepsilon^k)$ *times* **do**
  Pick an edge u.a.r. and an endpoint, v, of the selected edge u.a.r.
  Initialize all the vertices in G to be un-labeled.
  Call Recursive-tree-exclusion$(T, 0, v)$ and **return** *REJECT* if it returned 1.
return *ACCEPT*.

Procedure Recursive-tree-exclusion(T, i, v).

If v was already labeled then return 0, otherwise, label v by i. `// The recursion returns 0 if the revealed labeled subgraph is not` T.
Define $\ell = d_T(v_i) - 1$ if $i > 0$ and $\ell = d_T(v_i)$ otherwise.
Let $v_{i_1}, \ldots, v_{i_\ell}$ denote the labels of the children of v_i in T (in which v_0 is the root).
Pick u.a.r. ℓ vertices $u_1, \ldots, u_\ell$ from $N_G(v)$ and recursively call Recursive-tree-exclusion(T, i_j, u_j) for each $j \in [\ell]$
If one of the calls returned 0, then return 0, otherwise return 1.

References

1 Amir Abboud, Keren Censor-Hillel, and Seri Khoury. Near-linear lower bounds for distributed distance computations, even in sparse networks. In *30th International Symposium in Distributed Computing (DISC)*, pages 29–42, 2016. doi:10.1007/978-3-662-53426-7_3.

2 Noga Alon, Sonny Ben-Shimon, and Michael Krivelevich. A note on regular ramsey graphs. *Journal of Graph Theory*, 64(3):244–249, 2010.

3 Noga Alon, Eldar Fischer, Michael Krivelevich, and Mario Szegedy. Efficient testing of large graphs. *Combinatorica*, 20(4):451–476, 2000.

4 Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *J. ACM*, 42(4):844–856, 1995.

5 Noga Alon, Raphael Yuster, and Uri Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997.

6 Baruch Awerbuch, Bonnie Berger, Lenore Cowen, and David Peleg. Low-diameter graph decomposition is in nc. In *Scandinavian Workshop on Algorithm Theory*, pages 83–93. Springer, 1992.

7 Guy E Blelloch, Anupam Gupta, Ioannis Koutis, Gary L Miller, Richard Peng, and Kanat Tangwongsan. Nearly-linear work parallel sdd solvers, low-diameter decomposition, and low-stretch subgraphs. *Theory of Computing Systems*, 55(3):521–554, 2014.

8 Luciana Buriol, Gereon Frahling, Stefano Leonardi, Alberto Marchetti-Spaccamela, and Christian Sohler. Counting triangles in data streams. In *25th ACM Symposium on Principles of Database Systems (PODS)*, pages 253–262, 2006.

9 Keren Censor-Hillel, Eldar Fischer, Gregory Schwartzman, and Yadu Vasudev. Fast distributed algorithms for testing graph properties. In *30th Int. Symposium on Distributed Computing (DISC)*, volume 9888 of *LNCS*, pages 43–56. Springer, 2016.

10 Keren Censor-Hillel, Petteri Kaski, Janne H. Korhonen, Christoph Lenzen, Ami Paz, and Jukka Suomela. Algebraic methods in the congested clique. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 143–152, 2015.

11 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. In *43rd ACM Symposium on Theory of Computing (STOC)*, pages 363–372, 2011.

12 Danny Dolev, Christoph Lenzen, and Shir Peled. Tri, tri again: Finding triangles and small subgraphs in a distributed setting. In *26th International Symposium on Distributed Computing*, pages 195–209, 2012.

13 Andrew Drucker, Fabian Kuhn, and Rotem Oshman. On the power of the congested clique model. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 367–376, 2014.

14 Michael Elkin and Ofer Neiman. Efficient algorithms for constructing very sparse spanners and emulators. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 652–669. SIAM, 2017.

15 Paul Erdős, András Hajnal, and J. W. Moon. A problem in graph theory. *The American Mathematical Monthly*, 71(10):1107–1110, 1964.

16 Guy Even, Reut Levi, and Moti Medina. Faster and simpler distributed algorithms for testing and correcting graph properties in the congest-model. *CoRR*, abs/1705.04898, 2017. URL: `http://arxiv.org/abs/1705.04898`.

17 Orr Fischer, Tzlil Gonen, and Rotem Oshman. Distributed property testing for subgraph-freeness revisited. *CoRR*, abs/1705.04033, 2017. URL: `http://arxiv.org/abs/1705.04033`.

18 Pierre Fraigniaud, Pedro Montealegre, Dennis Olivetti, Ivan Rapaport, and Ioan Todinca. Distributed subgraph detection. *CoRR*, abs/1706.03996, 2017. URL: `http://arxiv.org/abs/1706.03996`.

19 Pierre Fraigniaud and Dennis Olivetti. Distributed detection of cycles. In *29th ACM on Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2017.

20 Pierre Fraigniaud, Ivan Rapaport, Ville Salo, and Ioan Todinca. Distributed testing of excluded subgraphs. In *30th Int. Symposium on Distributed Computing (DISC)*, volume 9888 of *LNCS*, pages 342–356. Springer, 2016.

21 Silvio Frischknecht, Stephan Holzer, and Roger Wattenhofer. Networks cannot compute their diameter in sublinear time. In *23rd ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1150–1162, 2012.

22 Oded Goldreich, Shari Goldwasser, and Dana Ron. Property testing and its connection to learning and approximation. *Journal of the ACM (JACM)*, 45(4):653–750, 1998.

23 Oded Goldreich and Dana Ron. Property testing in bounded degree graphs. *Algorithmica*, 32(2):302–343, 2002.

24 Stephan Holzer and Roger Wattenhofer. Optimal distributed all pairs shortest paths and applications. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 355–364, 2012. `doi:10.1145/2332432.2332504`.

25 Kazuo Iwama and Yuichi Yoshida. Parameterized testability. In *Innovations in Theoretical Computer Science, ITCS'14, Princeton, NJ, USA, January 12-14, 2014*, pages 507–516, 2014. `doi:10.1145/2554797.2554843`.

26 Taisuke Izumi and François Le Gall. Triangle finding and listing in CONGEST networks. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2017.

27 Stasys Jukna and Georg Schnitger. Triangle-freeness is hard to detect. *Combinatorics, Probability, & Computing*, 11(6):549–569, 2002.

28 Shay Kutten and David Peleg. Fast distributed construction of small k-dominating sets and applications. *J. Algorithms*, 28(1):40–66, 1998. `doi:10.1006/jagm.1998.0929`.

29 Tom Leighton. *Introduction to Parallel Algorithms and Architectures*. Morgan Kaufmann, 1992.

30 Christoph Lenzen and Boaz Patt-Shamir. Fast partial distance estimation and applications. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 153–162, 2015. `doi:10.1145/2767386.2767398`.

31 Gary L Miller, Richard Peng, and Shen Chen Xu. Parallel graph decompositions using random shifts. In *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*, pages 196–203. ACM, 2013.

32 Burkhard Monien. How to find long paths efficiently. In *Analysis and design of algorithms for combinatorial problems*, volume 109 of *North-Holland Math. Stud.*, pages 239–254. North-Holland, Amsterdam, 1985. doi:10.1016/S0304-0208(08)73110-4.

33 Danupon Nanongkai. Distributed approximation algorithms for weighted shortest paths. In *ACM Symposium on Theory of Computing (STOC)*, pages 565–573, 2014. doi:10.1145/2591796.2591850.

34 Hiroaki Ookawa and Taisuke Izumi. Filling logarithmic gaps in distributed complexity for global problems. In *41st International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, pages 377–388, 2015. doi:10.1007/978-3-662-46078-8_31.

35 Judea Pearl. Fusion, propagation, and structuring in belief networks. *Artif. Intell.*, 29(3):241–288, 1986.

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

Error-Sensitive Proof-Labeling Schemes*†

Laurent Feuilloley[1] and Pierre Fraigniaud[2]

1 **Institut de Recherche en Informatique Fondamentale (IRIF), CNRS and University Paris Diderot, Paris, France**
2 **Institut de Recherche en Informatique Fondamentale (IRIF), CNRS and University Paris Diderot, Paris, France**

Abstract

Proof-labeling schemes are known mechanisms providing nodes of networks with *certificates* that can be *verified* locally by distributed algorithms. Given a boolean predicate on network states, such schemes enable to check whether the predicate is satisfied by the actual state of the network, by having nodes interacting with their neighbors only. Proof-labeling schemes are typically designed for enforcing fault-tolerance, by making sure that if the current state of the network is illegal with respect to some given predicate, then at least one node will detect it. Such a node can raise an alarm, or launch a recovery procedure enabling the system to return to a legal state. In this paper, we introduce *error-sensitive* proof-labeling schemes. These are proof-labeling schemes which guarantee that the number of nodes detecting illegal states is linearly proportional to the edit-distance between the current state and the set of legal states. By using error-sensitive proof-labeling schemes, states which are far from satisfying the predicate will be detected by many nodes, enabling fast return to legality. We provide a structural characterization of the set of boolean predicates on network states for which there exist error-sensitive proof-labeling schemes. This characterization allows us to show that classical predicates such as, e.g., acyclicity, and leader admit error-sensitive proof-labeling schemes, while others like regular subgraphs don't. We also focus on *compact* error-sensitive proof-labeling schemes. In particular, we show that the known proof-labeling schemes for spanning tree and minimum spanning tree, using certificates on $O(\log n)$ bits, and on $O(\log^2 n)$ bits, respectively, are error-sensitive, as long as the trees are locally represented by adjacency lists, and not just by parent pointers.

1998 ACM Subject Classification D.1.3 Concurrent Programming – distributed programming, F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Fault-tolerance, distributed decision, distributed property testing

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.16

1 Introduction

In the context of fault-tolerant distributed computing, it is desirable that the computing entities in the system be able to detect whether the system is in a legal state (w.r.t. some boolean predicate, potentially expressed in various forms of logics) or not. In the framework of distributed network computing, several mechanisms have been proposed to ensure such a detection (see, e.g., [1, 2, 4, 5, 22]). Among them, *proof-labeling schemes* [22] are mechanisms enabling failure detection based on additional information provided to the nodes. More specifically, a proof-labeling scheme is composed of a *prover*, and a *verifier*. A prover is an oracle that assigns a *certificate* to each node of any given network, and a verifier is a

* Both authors received additional support from ANR project DESCARTES, and Inria project GANG.
† Full version available online at `http://arxiv.org/abs/1705.04144`.


© Laurent Feuilloley and Pierre Fraigniaud;
licensed under Creative Commons License CC-BY

distributed algorithm that locally checks whether the collection of certificates is a *distributed proof* that the network is in a legal state with respect to a given predicate – by "locally", we essentially mean: by having each node interacting with its neighbors only.

The prover is actually an abstraction. In practice, the certificates are provided by a distributed algorithm solving some task (see, e.g., [3, 6, 22]). For instance, let us consider spanning tree construction, where every node must compute a pointer to a neighboring node such that the collection of pointers form a tree spanning all nodes in the network. In that case, the algorithm in charge of constructing a spanning tree is also in charge of constructing the certificates providing a distributed proof allowing a verifier to check that proof locally. That is, the verifier must either accept or reject at every node, under the following constraints. If the constructed set of pointers form a spanning tree, then the constructed certificates must lead the verifier to accept at every node. Instead, if the constructed set of pointers does not form a spanning tree, then, for every possible certificate assignment to the nodes, at least one node must reject. The rejecting node may then raise an alarm, or launch a recovery procedure. Abstracting the construction of the certificates thanks to a prover enables to avoid delving into the implementation details relative to the distributed construction of the certificates, for focussing attention on whether such certificates exist, and on what should be their forms. The reader is referred to [7] for more details about the connections between proof-labeling schemes and fault-tolerant computing.

One weakness of proof-labeling schemes is that they may not allow the system running the verifier to distinguish between a global state which is slightly erroneous, and a global state which is completely bogus. In both cases, it is only required that at least one node detects the illegality of the state. In the latter case though, having only one node raising an alarm, or launching a recovery procedure for bringing the whole system back to a legal state, might be quite inefficient. Instead, if many nodes would detect the errors, then bringing back the system into a legal state will be achieved by a collection of local resets running in parallel, instead of a single reset traversing the whole network sequentially.

Moreover, in several contexts like, e.g., *property-testing* [14, 15], monitoring an error-prone system is implemented via an external mechanism involving a monitor that is probing the system by querying a (typically small) subset of nodes chosen at random. *Non-deterministic* property-testing has been recently investigated in the literature [18, 24], where a certificate is given to the property-testing algorithm. Such a certificate is however global. Instead, we are interested in decentralized certificates, which can also be viewed as, say, annotations provided to the nodes of a network, or to the entries of a database. The correction of the network, or of the database, is then checked by a property-testing algorithm querying nodes at random for recovering the individual states of these nodes, including their certificates. To be efficient, such distributed certificates must guarantee that, if the monitored system is far from being correct, then many nodes are capable to detect the error. Indeed, if just one node is capable to detect the error, then the probability that the monitoring system will query that specific node is very low, resulting in a large amount of time before the error is detected.

In this paper, we aim at designing *error-sensitive* proof-labeling schemes, which guarantee that system states that are far from being correct can be detected by many nodes, providing faster recovery if the error detection mechanism is decentralized, or faster discovery if this error detection mechanism is centralized.

More specifically, the distance between two global states of a distributed system is defined as the *edit-distance* between these two states, i.e., the minimum number of individual states required to be modified in order to move from one global state to the other. A proof-labeling scheme is *error-sensitive* if there exists a constant $\alpha > 0$ such that, for any erroneous system

state S, the number of nodes detecting the error is at least $\alpha\, d(S)$, where $d(S)$ is the shortest edit-distance between S and a correct system state. The choice of a linear dependency between the number of nodes detecting the error, and the edit-distance to legal states is not arbitrary, but motivated by the following two observations.

- On the one hand, a linear dependency is somewhat the best that we may hope for. Indeed, let us consider a k-node network G in some illegal state S for which r nodes are detecting the illegality of S — think about, e.g., the spanning tree predicate. Then, let us make n copies of G and its state S, potentially linked by $n-1$ additional edges if one insists on connectivity. In the resulting kn-node network, we get that $O(rn)$ nodes are detecting illegality, which grows linearly with the number of nodes, as n grows.
- On the other hand, while a sub-linear dependency may still be useful in some contexts, this would be insufficient in others. For instance, in the context of property testing, for systems that are ϵ-far from being correct (i.e., essentially, an ϵ fraction of the individual states are incorrect), the linear dependency enables to find a node capable to detect the error after $O(1/\epsilon)$ expected number of queries to random nodes. Instead, a sub-linear dependency would yield an expected number of queries that grows with the size of the system before querying a node capable to detect the error.

Our results. We consider boolean predicates on graphs with labeled nodes, as in, e.g., [25]. Given a graph G, a labeling of G is a function $\ell : V(G) \to \{0,1\}^*$ assigning binary strings to nodes. A *labeled graph* is a pair (G,ℓ) where G is a graph, and ℓ is a labeling of G. Given a boolean predicate $\mathcal{P}$ on labeled graphs, the *distributed language* associated to $\mathcal{P}$ is

$$\mathcal{L} = \{(G,\ell) \text{ satisfying } \mathcal{P}\}.$$

It is known that every (Turing decidable) distributed language admits a proof-labeling scheme [17, 22]. We show that the situation is radically different when one is interested in error-sensitive proof-labeling schemes. In particular, not all distributed languages admit an error-sensitive proof-labeling scheme. Moreover, the existence of error-sensitive proof-labeling schemes for the solution of a distributed task is very much impacted by the way the task is specified. For instance, in the case of spanning tree construction, we show that asking every node to produce a single pointer to its parent in the tree cannot be certified in an error-sensitive manner, while asking every node to produce the list of its neighbors in the tree can be certified in an error-sensitive manner.

Our first main result is a structural characterization of the distributed languages for which there exist error-sensitive proof-labeling schemes. Namely, a distributed language admits an error-sensitive proof-labeling scheme if and only if it is *locally stable*. The notion of local stability is purely structural. Roughly, a distributed language $\mathcal{L}$ is locally stable if a labeling ℓ resulting from copy-pasting parts of correct labelings to different subsets $S_1, \ldots, S_k$ of nodes in a graph G results in a labeled graph (G,ℓ) that is not too far from being legal, in the sense that the edit-distance between (G,ℓ) and $\mathcal{L}$ is proportional to the size of the boundary of the subsets $S_1, \ldots, S_k$ in G, and not to the size of these subsets. This characterization allows us to show that important distributed languages (such as, e.g., acyclicity, leader, etc.) admit error-sensitive proof-labeling schemes, while some very basic distributed languages (such as, e.g., regular subgraph, etc.) do not admit error-sensitive proof-labeling schemes.

Our second main contribution is a proof that the known space-optimal proof-labeling schemes for spanning tree with $O(\log n)$-bit certificates, and for minimum spanning tree (MST) with $O(\log^2 n)$-bit certificates, are both error-sensitive, whenever the trees are encoded at each node by an adjacency list (and not by a single pointer to the parent). Hence, error-sensitivity comes at no cost for spanning tree and MST. Proving this result requires to

establish some kind of matching between the erroneously labeled nodes and the rejecting nodes. Establishing this matching is difficult because, for both spanning tree and MST, the rejecting nodes might be located far away from the erroneous nodes. Indeed, the presence of certificates helps local detection of errors, but decorrelates the nodes at which the alarms take place from the nodes at which the errors take place. (See Section 6 for a discussion about *proximity-sensitive* proof-labeling schemes). Moreover, in the case of MST, the known space-optimal proof-labeling scheme uses $O(\log n)$ "layers" of spanning trees (corresponding roughly to the $O(\log n)$ levels of fragments constructed by Borůvka algorithm). It is not a priori clear that errors occurring at different levels are necessarily detected by different nodes, i.e., that k errors are necessarily detected by $\Omega(k)$ nodes, and not just by $O(k/\log n)$ nodes.

Related work. As mentioned before, one important motivation for our work is fault-tolerant distributed computing, with the help of failure detection mechanisms such as proof-labeling schemes. Proof-labeling schemes were introduced in [22]. A tight bound of $\Theta(\log^2 n)$ bits on the size of the certificates for certifying MST was established in [19, 20]. Several variants of proof-labeling schemes have been investigated in the literature, including verification at distance greater than one [17], and the design of proofs with identity-oblivious certificates [12]. Connections between proof-labeling schemes and the design of distributed (silent) self-stabilizing algorithms were studied in [7]. Extensions of proof-labeling schemes for the design of (non-silent) self-stabilizing algorithms were investigated in [21]. In all these work, the number of nodes susceptible to detect an incorrect configuration is not considered, and the only constraint imposed on the error-detection mechanism is that an erroneous configuration must be detected by at least one node. Our work requires the number of nodes detecting an erroneous configuration to grow linearly with the number of errors.

Another important motivation for our work is property testing. Graph property testing was investigated in numerous papers (see [14, 15] for an introduction to the topic), and was recently extended to a non-deterministic setting [18, 24] in which the centralized algorithm is provided with a centralized certificate. Distributed property testing has been introduced in [8], and formalized in [9] (see also [13]). Our work may find applications to centralized property testing, but where the certificate is decentralized. Our error-sensitive scheme guarantees that if the current configuration of the network is ϵ-far from being correct, then probing a constant expected number of nodes is sufficient to detect that this configuration is erroneous.

From a higher perspective, our approach aims at closing the gap between local distributed computing and centralized computing in networks, by studying distributed error-detection mechanisms that perform locally, but generate individual outputs that are related to the global correctness of the system at hand. As such, it is worth mentioning other efforts in the same direction, including especially work in the context of centralized local computing, like, e.g., [10, 16, 26].

Distributed property testing and proof-labeling schemes are different forms of distributed decision mechanisms, which have been investigated under various models for distributed computing. We refer to [11] for a recent survey on distributed decision.

2 Model and definitions

Throughout the paper, all graphs are assumed to be connected and simple (no self-loops, and no parallel edges. Given a node v of a graph G, we denote by $N(v)$ the open neighborhood of v, i.e., the set of neighbors of v in G. In some contexts (as, e.g., MST), the considered graphs may be edge-weighted.

All results in this paper are stated in the classical LOCAL model [27] for distributed network computing, where networks are modeled by undirected graphs whose nodes model the computing entities, and edges model the communication links. Recall that the LOCAL model assumes that nodes are given distinct identities (a.k.a. IDs), and that computation proceeds in synchronous rounds. All nodes simultaneously start executing the given algorithm. At each round, nodes exchange messages with their neighbors, and perform individual computation. There are no limits placed on the message size, nor on the amount of computation performed at each round. Specifically, we are interested in *proof-labeling schemes* [22], which are well established mechanisms enabling to locally detect inconsistencies in the global states of networks with respect to some given boolean predicate. Such mechanisms involve a verification algorithm which performs in just a single round in the LOCAL model. In order to recall the definition of proof-labeling schemes, we first recall the definition of *distributed languages* [12].

A distributed language is a collection of labeled graphs, that is, a set $\mathcal{L}$ of pairs (G, ℓ) where G is a graph, and $\ell : V(G) \to \{0,1\}^*$ is a labeling function assigning a binary string to each node of G. Such a labelling may encode just a boolean (e.g., whether the node is in a dominating set or not), or an integer (e.g., in graph coloring), or a collection of neighbor IDs (e.g., for locally encoding a subgraph). A distributed language is said *constructible* if, for every graph G, there exists ℓ such that $(G, \ell) \in \mathcal{L}$. It is *Turing decidable* if there exists a (centralized) algorithm which, given (G, ℓ) returns whether $(G, \ell) \in \mathcal{L}$ or not. All distributed languages considered in this paper are always assumed to be constructible and Turing decidable.

Given a distributed language $\mathcal{L}$, a proof-labeling scheme for $\mathcal{L}$ is a pair prover-verifier $(\mathbf{p}, \mathbf{v})$, where $\mathbf{p}$ is an oracle assigning a certificate function $c : V(G) \to \{0,1\}^*$ to every labeled graph $(G, \ell) \in \mathcal{L}$, and $\mathbf{v}$ is a 1-round distributed algorithm[1] taking as input at each node v its identity $\mathsf{ID}(v)$, its label $\ell(v)$, and its certificate $c(v)$, such that, for every labeled graph (G, ℓ) the following two conditions are satisfied:

- If $(G, \ell) \in \mathcal{L}$ then $\mathbf{v}$ outputs *accept* at every node of G whenever all nodes of G are given the certificates provided by $\mathbf{p}$;
- If $(G, \ell) \notin \mathcal{L}$ then, for every certificate function $c : V(G) \to \{0,1\}^*$, $\mathbf{v}$ outputs *reject* in at least one node of G.

The first condition guarantees the existence of certificates allowing the given legally labeled graph (G, ℓ) to be globally accepted. The second condition guarantees that the verifier cannot be "cheated", that is, an illegally labeled graph will systematically be rejected by at least one node, whatever "fake" certificates are given to the nodes. It is known that every distributed language has a proof-labeling scheme [22].

To define the novel notion of *error-sensitive* proof-labeling schemes, we introduce the following notion of distance between labeled graphs. Let ℓ and ℓ' be two labelings of a same graph G. The *edit distance* between (G, ℓ) and (G, ℓ') is the minimum number of elementary operations required to transform (G, ℓ) into (G, ℓ'), where an elementary operation consists of replacing a node label by another label. That is, the edit distance between (G, ℓ) and (G, ℓ') is simply

$$|\{v \in V(G) : \ell(v) \neq \ell'(v)\}|.$$

The edit-distance from a labeled graph (G, ℓ) to a language $\mathcal{L}$ is the minimum, taken over all labelings ℓ' of G satisfying $(G, \ell') \in \mathcal{L}$, of the edit-distance between (G, ℓ) and (G, ℓ').

[1] That is, every node outputs after having communicated with all its neighbors only once.

Roughly, an error-sensitive proof-labeling scheme satisfies that the number of nodes that reject a labeled graph (G, ℓ) should be (at least) proportional to the distance between (G, ℓ) and the considered language.

▶ **Definition 1.** A proof-labeling scheme $(\mathbf{p}, \mathbf{v})$ for a language $\mathcal{L}$ is *error-sensitive* if there exists a constant $\alpha > 0$, such that, for every labeled graph (G, ℓ),

- If $(G, \ell) \in \mathcal{L}$ then $\mathbf{v}$ outputs *accept* at every node of G whenever all nodes of G are given the certificates provided by $\mathbf{p}$;
- If $(G, \ell) \notin \mathcal{L}$ then, for every certificate function $c : V(G) \to \{0,1\}^*$, $\mathbf{v}$ outputs *reject* in at least $\alpha \, \mathsf{d}$ nodes of G, where d is the edit distance between (G, ℓ) and $\mathcal{L}$, i.e., $\mathsf{d} = \mathrm{dist}\big((G, \ell), \mathcal{L}\big)$.

Note that the at least $\alpha \, \mathsf{d}$ nodes rejecting a labeled graph (G, ℓ) at edit-distance d from $\mathcal{L}$ do not need to be the same for all certificate functions.

3 Basic properties of error-sensitive proof-labeling schemes

Let us first illustrate the notion of error-sensitive proof-labeling scheme by exemplifying its design for a classic example of distributed languages. Let ACYCLIC be the following distributed language:

$$\text{ACYCLIC} = \Big\{ (G, \ell) : \forall v \in V(G),\, \ell(v) \in N(v) \cup \{\bot\} \text{ and } \bigcup_{v \in V(G) \,:\, \ell(v) \neq \bot} \{v, \ell(v)\} \text{ is acyclic} \Big\}$$

That is, the label of a node is interpreted as a pointer to some neighboring node, or to null. Then $(G, \ell) \in$ ACYCLIC if the subgraph of G described by the set of non-null pointers is acyclic. We show that ACYCLIC has an error-sensitive proof-labeling scheme. The proof of this result is easy, as fixing of the labels can be done locally, at the rejecting nodes. Nevertheless, the proposition and its proof serve as a basic example illustrating the notion of error-sensitive proof-labeling scheme.

▶ **Proposition 2.** ACYCLIC *has an error-sensitive proof-labeling scheme.*

Proof. Let $(G, \ell) \in$ ACYCLIC. Every node $v \in V(G)$ belongs to an in-tree rooted at a node r such that $\ell(r) = \bot$. The prover $\mathbf{p}$ provides every node v with its distance $d(v)$ to the root of its in-tree (i.e., number of hops to reach the root by following the pointers specified by ℓ). The verifier $\mathbf{v}$ proceeds at every node v as follows: first, it checks that $\ell(v) \in N(v) \cup \{\bot\}$; second, it checks that, if $\ell(v) \neq \bot$ then $d(\ell(v)) = d(v) - 1$, and if $\ell(v) = \bot$ then $d(v) = 0$. If all these tests are passed, then v accepts. Otherwise, it rejects. By construction, if (G, ℓ) is acyclic, then all nodes accept with these certificates. Conversely, if there is a cycle C in (G, ℓ), then let v be a node with maximum value $d(v)$ in C. Its predecessor in C (i.e., the node $u \in C$ with $\ell(u) = v$) rejects. So $(\mathbf{p}, \mathbf{v})$ is a proof-labeling scheme for ACYCLIC. We show that $(\mathbf{p}, \mathbf{v})$ is error-sensitive. Suppose that $\mathbf{v}$ rejects (G, ℓ) at $k \geq 1$ nodes. Let us replace the label $\ell(v)$ of each rejecting node v by the label $\ell'(v) = \bot$, and keep the labels of all other nodes unchanged, i.e., $\ell'(v) = \ell(v)$ for every node where $\mathbf{v}$ accepts. We have $(G, \ell') \in$ ACYCLIC. Indeed, by construction, the label of every node u in (G, ℓ') has a well-formatted label $\ell'(v) \in N(v) \cup \{\bot\}$. Moreover, let us assume, for the purpose of contradiction, that there is a cycle C in (G, ℓ'). By definition, every node v of this cycle is pointing to $\ell'(v) \in N(v)$. Thus $\ell'(v) = \ell(v)$ for every node of C, from which it follows that no nodes of C was rejecting with ℓ, a contradiction with the fact that, as observed before,

$\mathbf{v}$ rejects every cycle. Therefore $(G, \ell') \in$ ACYCLIC. Hence the edit-distance between (G, ℓ) and ACYCLIC is at most k. It follows that $(\mathbf{p}, \mathbf{v})$ is error-sensitive, with sensitivity parameter $\alpha \geq 1$. ◀

The definition of error-sensitiveness is based on the existence of a proof-labeling scheme for the considered language. However, two different proof-labeling schemes for the same language may have different sensitivity parameters α. In fact, we show that every non-trivial language admits a proof-labeling schemes which is *not* error-sensitive. That is, the following result shows that demonstrating the existence of a proof-labeling scheme that is *not* error-sensitive for a language does not prevent that language to have another proof-labeling scheme which *is* error-sensitive. We say that a distributed language is *trivially approximable* if there exists a constant d such that every labeled graph (G, ℓ) is at edit-distance at most d from $\mathcal{L}$. The proof of the following result can be found in the full version.

▶ **Proposition 3.** *Let $\mathcal{L}$ be a distributed language. Unless $\mathcal{L}$ is trivially approximable, there exists a proof-labeling scheme for $\mathcal{L}$ that is not error-sensitive.*

Recall that the fact that every distributed language has a proof-labeling scheme can be established by using a *universal* proof-labeling scheme $(\mathbf{p}_{univ}, \mathbf{v}_{univ})$ (see [17]). Given a distributed language $\mathcal{L}$, and a labeled graph $(G, \ell) \in \mathcal{L}$ on an n-node graph G, a universal certificate $c : V(G) \to \{0,1\}^*$ for that labeled graph is defined for every node $u \in V(G)$ by the triple $c(u) = (T, M, L)$ where nodes are ordered from 1 to n in arbitrary order, T is a vector with n entries indexed from 1 to n where $T[i]$ is the ID of the ith node v, $L[i]$ is the label $\ell(v)$ of the ith node v, and M is the adjacency matrix of G. The prover $\mathbf{p}_{univ}$ assigns $c(u)$ to every node $u \in V(G)$. The verifier $\mathbf{v}_{univ}$ then checks at every node u that its certificate is consistent with the certificates given to its neighbors (i.e., they all have the same T, L, and M, the indexes matches with the IDs, and the actual neighborhood of v is as it is specified in T, L and M). If this test is not passed, then $\mathbf{v}_{univ}$ outputs *reject* at u, otherwise it outputs *accept* or *reject* according to whether the labeled graph described by (M, L) is in $\mathcal{L}$ or not. It is easy to check that $(\mathbf{p}_{univ}, \mathbf{v}_{univ})$ is indeed a proof-labeling scheme for $\mathcal{L}$. The universal proof-labeling scheme has the following nice property, that we state as a lemma for further references in the text (see proof in the full version).

▶ **Lemma 4.** *If a distributed language $\mathcal{L}$ has an error-sensitive proof-labeling scheme, then the universal proof-labeling scheme applied to $\mathcal{L}$ is error-sensitive.*

While every distributed language has a proof-labeling scheme, we show, using Lemma 4, that there exist languages for which there are no error-sensitive proof-labeling schemes (see the full version for the proof).

▶ **Proposition 5.** *There exist languages that do not admit any error-sensitive proof-labeling scheme.*

▶ Remark. The language REGULAR used in the proof of Proposition 5 to establish the existence of languages that do not admit any error-sensitive proof-labeling schemes actually belongs to the class LCL of locally checkable labelings [25]. Therefore, the fact that a language is easy to check locally does not help for the design of error-sensitive proof-labeling schemes.

We complete this warmup section by some observations regarding the encoding of distributed data structures. Let us consider the following two distributed languages, both corresponding to spanning tree. The first language, ST_p, encodes the spanning trees using pointers to parents, while the second language, ST_l, encodes the spanning trees by listing all the incident edges of each node in these tree.

$$\text{ST}_p = \Big\{(G,\ell) : \quad \forall v \in V(G),\, \ell(v) \in N(v) \cup \{\perp\} \\ \text{and } \Big(\bigcup_{v \in V(G) : \ell(v) \neq \perp} \{v, \ell(v)\} \Big) \text{ forms a spanning tree} \Big\}$$

$$\text{ST}_l = \Big\{(G,\ell) : \quad \forall v \in V(G),\, \ell(v) \subseteq N(v) \text{ and } u \in \ell(v) \text{ iff } v \in \ell(u), \\ \text{and } \Big(\bigcup_{v \in V(G)} \bigcup_{u \in \ell(v)} \{u, v\} \Big) \text{ forms a spanning tree} \Big\}.$$

Obviously, ST_p is just a compressed version of ST_l as the latter can be constructed from the former in just one round. However, note that ST_p cannot be recover from ST_l in a constant number of rounds, because ST_p provides a consistent orientation of the edges in the tree. It follows that ST_p is an encoding of spanning trees which is actually strictly richer than ST_l. This difference between ST_p and ST_l is not anecdotal, as we shall prove later that ST_l admits an error-sensitive proof-labeling scheme, while we show hereafter that ST_p is not appropriate for the design of error-sensitive proof-labeling schemes.

▶ **Proposition 6.** ST_p *does not admit any error-sensitive proof-labeling scheme.*

Proof. Let P_n be the n-node path $u_1, u_2, \ldots, u_n$ with n even. Let ℓ_0, ℓ_1, and ℓ_2 be labelings defined by $\ell_1(u_i) = u_{i+1}$ for all $1 \leq i < n$, and $\ell_1(u_n) = \perp$; $\ell_2(u_i) = u_{i-1}$ for all $1 < i \leq n$, and $\ell_2(u_1) = \perp$; and $\ell_3(u_i) = u_{i-1}$ for all $1 < i \leq \frac{n}{2}$, $\ell_3(u_i) = u_{i+1}$ for all $\frac{n}{2} + 1 \leq i < n$, and $\ell_3(u_1) = \ell_3(u_n) = \perp$. We have $(P_n, \ell_1) \in \text{ST}_p$ and $(P_n, \ell_2) \in \text{ST}_p$, while the distance from (P_n, ℓ_3) to ST_p is at least $\frac{n}{2}$. Let $(\mathbf{p}, \mathbf{v})$ be a proof-labeling scheme for ST_p. Consider the case of (P_n, ℓ_3) where every u_i, $i = 1, \ldots, \frac{n}{2}$, is given the certificate assigned by $\mathbf{p}$ to u_i in (P_n, ℓ_2), and every u_i, $i = \frac{n}{2} + 1, \ldots, n$, is given the certificate assigned by $\mathbf{p}$ to u_i in (P_n, ℓ_1). With such certificates, (P_n, ℓ_3) is rejected by $\mathbf{v}$ at $u_{\frac{n}{2}}$ and $u_{\frac{n}{2}+1}$ only. ◀

4 Characterization

We now define the notion of *local stability*, which allows us to characterize the distributed languages admitting an error-sensitive proof-labeling scheme. This notion naturally pops up in the context of proof-labeling schemes [22] and locally checkable proofs in general [17]. Indeed, in these latter frameworks, languages that are "hard" to prove, in the sense that they require certificates of large size (typically of $\Omega(n^2)$ bits), are not locally stable, in the sense that glueing together two legal labeled graphs, say by connecting them by an edge, results in a labeled graph which can be very far from being legal. Local stability also naturally pops up in the context of the classical construction tasks which admit local algorithms, such as $(\Delta + 1)$-coloring and MIS [23]. Indeed, those tasks share the property that any partial solution can be extended to a larger solution without modifying the already assigned labels. Extending the partial solution actually only depends on the "border" of the current partial solution.

More specifically, let G be a graph, and let H be a subgraph of G, that is, a graph H such that $V(H) \subseteq V(G)$, and $E(H) \subseteq E(G)$. We denote by $\partial_G H$ the set of nodes at the *boundary* of H in G, that is, which belongs to $V(H)$, and are incident to an edge in $E(G) \setminus E(H)$.

Given a labeling ℓ of a graph G, and a subgraph H of G, the labeling ℓ_H denotes the labeling of H induced by ℓ restricted to the nodes of H:

$$\ell_H(v) = \begin{cases} \ell(v) & \text{if } v \in V(H) \\ \emptyset & \text{otherwise (where } \emptyset \text{ denotes the empty string).} \end{cases}$$

Roughly, a distributed language $\mathcal{L}$ is locally stable if, by copy-pasting parts of legal labelings with small cuts between these parts, the resulting labeled graph is not too far from being legal. More precisely, let G be a graph, and let $H_1, \ldots, H_k$ be a family of connected subgraphs of G such that $(V(H_i))_{i=1,\ldots,k}$ is a partition of $V(G)$. For every $i \in \{1, \ldots, k\}$, let us consider a labeled graph $(G_i, \ell_i) \in \mathcal{L}$ such that H_i is a subgraph of G_i. Let ℓ be the labeling of G defined as $\ell = \sum_{i=1}^{k} \ell_i$, i.e. for every $v \in V(G)$, $\ell(v) = \ell_i(v)$ where i is such that $v \in V(H_i)$. We say that such a labeled graph (G, ℓ) is *induced* by the labeled graphs (G_i, ℓ_i), $i = 1, \ldots, k$, *via* the subgraphs $H_1, \ldots, H_k$.

▶ **Definition 7.** A language $\mathcal{L}$ is *locally stable* if there exists a constant $\beta > 0$, such that, for every labeled graph (G, ℓ) induced by labeled graphs $(G_i, \ell_i) \in \mathcal{L}$, $i = 1, \ldots, k$, via subgraphs $H_1, \ldots, H_k$, the edit-distance between (G, ℓ) and $\mathcal{L}$ is at most $\beta \, | \cup_{i=1}^{k} \partial_G H_i \cup \partial_{G_i} H_i |$.

That is, the labeled graph resulting from cut-and-pasting parts of legally labeled graphs (G_i, ℓ_i), $i = 1, \ldots, k$, is at edit-distance from $\mathcal{L}$ upper bounded by the number of nodes at the boundary of the subgraphs H_i in G and G_i, and is independent of the number of nodes in each of these subgraphs H_i, $i = 1, \ldots, k$.

We have now all ingredients to state our characterization result:

▶ **Theorem 8.** *Let $\mathcal{L}$ be a distributed language. $\mathcal{L}$ admits an error-sensitive proof-labeling scheme if and only if $\mathcal{L}$ is locally stable.*

Proof. We first show that if a distributed language $\mathcal{L}$ admits an error-sensitive proof-labeling scheme then $\mathcal{L}$ is locally stable. So, let $\mathcal{L}$ be a distributed language, and let $(\mathbf{p}, \mathbf{v})$ be an error-sensitive proof-labeling scheme for $\mathcal{L}$ with sensitivity parameter α. Let (G, ℓ) be a labeled graph induced by labeled graphs $(G_i, \ell_i) \in \mathcal{L}$, $i = 1, \ldots, h$, via the subgraphs $H_1, \ldots, H_h$ for some $h \geq 1$. Since, for every $i \in \{1, \ldots, h\}$, $(G_i, \ell_i) \in \mathcal{L}$, there exists a certificate function c_i such that $\mathbf{v}$ accepts at every node of (G_i, ℓ_i) provided with the certificate function c_i. Now, let us consider the labeled graph (G, ℓ), with certificate $c_i(u)$ on every node $u \in V(H_i)$ for all $i = 1, \ldots, h$. With such certificates, the nodes in $V(H_i)$ that are not in $\partial_G H_i \cup \partial_{G_i} H_i$ have the same close neighborhood in (G, ℓ) and in (G_i, ℓ_i). Therefore, they accept in (G, ℓ) the same way they accept in (G_i, ℓ_i). It follows that the number of rejecting nodes is bounded by $| \cup_{i=1}^{h} \partial_G H_i \cup \partial_{G_i} H_i |$, and therefore (G, ℓ) is at edit-distance at most $\frac{1}{\alpha} | \cup_{i=1}^{h} \partial_G H_i \cup \partial_{G_i} H_i |$ from $\mathcal{L}$. Hence, $\mathcal{L}$ is locally stable, with parameter $\beta = \frac{1}{\alpha}$.

It remains to show that if a distributed language is locally stable then it admits an error-sensitive proof-labeling scheme. Let $\mathcal{L}$ be a locally stable distributed language with parameter β. We prove that the universal proof-labeling scheme $(\mathbf{p}_{univ}, \mathbf{v}_{univ})$ for $\mathcal{L}$ (cf. Section 3) is error-sensitive for some parameter α depending only on β. Let $(G, \ell) \notin \mathcal{L}$, and let us fix some certificate function c. The verifier $\mathbf{v}_{univ}$ rejects in at least one node. We show that if $\mathbf{v}_{univ}$ rejects at k nodes, then the edit-distance between (G, ℓ) and $\mathcal{L}$ is at most k/α for some constant $\alpha > 0$ depending only on β. For this purpose, let us consider the outputs of $\mathbf{v}_{univ}$ applied to (G, ℓ) with certificate c, and let us define the graph G' as the graph obtained from G by removing all edges for which $\mathbf{v}_{univ}$ rejects at both extremities. Note that the graph G' may not be connected.

Let C be a connected component of G', with at least one node u at which $\mathbf{v}_{univ}$ accepts. Let $c(u) = (T, M, L)$ be the certificate of node u, as it should be in the universal proof-labeling scheme as described in section 3. Since $\mathbf{v}_{univ}$ accepts at u, node u shares the same triple

(T, M, L) with all its neighbors in G', as $\mathbf{v}_{univ}$ would reject at u otherwise. Similarly, for every neighbor v of u, it must be the case that v agrees on (T, M, L) with each of its neighbors w in G', as otherwise $\mathbf{v}_{univ}$ would have rejected at both v and w, and the edge $\{v, w\}$ would have been removed from G. It follows that all nodes in C share the same triple (T, M, L) as the one given to the accepting node u. Also (M, L) coincides with the local structure of C and its labeling ℓ at all accepting nodes in C. Moreover, since $\mathbf{v}_{univ}$ accepts at u, we have $(M, L) \in \mathcal{L}$. We denote by (G_C, ℓ_C) this labeled graph in $\mathcal{L}$.

Let C be a connected component of G' where all nodes reject. In fact, by construction, such a component is composed of just one isolated node. For every such isolated rejecting node u, let us denote by (G_C, ℓ_C) a labeled graph composed of a unique node, with ID equal to the ID of u, and with labeling $\ell_C(u)$ such that $(G_C, \ell_C) \in \mathcal{L}$.

Let $\mathcal{C}$ be the set of all connected components of G'. Let (G, ℓ') be the graph induced by labeled graphs (G_C, ℓ_C) via the subgraphs $C \in \mathcal{C}$. Note that (G, ℓ) and (G, ℓ') coincide, but for the isolated rejecting nodes. By local stability, (G, ℓ') is at edit-distance at most $\beta\,|\cup_{C\in\mathcal{C}} \partial_G C \cup \partial_{G_C} C|$ from $\mathcal{L}$. Now, the nodes in $\cup_{C\in\mathcal{C}} \partial_G C \cup \partial_{G_C} C$ are exactly the rejecting nodes. Thus the number k of rejecting nodes satisfies $k = |\cup_{C\in\mathcal{C}} \partial_G C \cup \partial_{G_C} C|$, and the edit-distance from (G, ℓ') to $\mathcal{L}$ is at most $\beta\, k$. On the other hand, by construction, the edit-distance between (G, ℓ') and (G, ℓ) is at most the number of isolated rejecting nodes, that is, at most k. Therefore, the edit-distance between (G, ℓ) and $\mathcal{L}$ is at most $(\beta + 1)\, k$. Thus, the universal proof-labeling scheme is error-sensitive, with parameter $\alpha = \frac{1}{\beta+1}$. ◀

Proposition 5 can be viewed as a corollary of Theorem 8 as it is easy to show that REGULAR is not locally stable. Nevertheless, local stability may not always be as easy to establish, because it is based on merging an arbitrary large number of labeled graphs. We thus consider another property, called *strong local stability*, which is easier to check, and which provides a sufficient condition for the existence of an error-sensitive proof-labeling scheme. Given two labeled graphs (G, ℓ) and (G', ℓ'), and a subgraph H of both G and G', the labeling $\ell - \ell_H + \ell'_H$ for G is the labeling such that, for every node $v \in V(G)$, $(\ell - \ell_H + \ell'_H)(v) = \ell'_H(v)$ if $v \in V(H)$, and $(\ell - \ell_H + \ell'_H)(v) = \ell(v)$ otherwise.

▶ **Definition 9.** A language $\mathcal{L}$ is *strongly locally stable* if there exists a constant $\beta > 0$, such that, for every graph H, and every two labeled graphs $(G, \ell) \in \mathcal{L}$ and $(G', \ell') \in \mathcal{L}$ admitting H as a subgraph, the labeled graph $(G, \ell - \ell_H + \ell'_H)$ is at edit-distance at most $\beta\,|\partial_{G'} H + \partial_G H|$ from $\mathcal{L}$.

The following lemma states that strong local stability is indeed a notion that is at least as strong as local stability (see proof in the full version).

▶ **Lemma 10.** *If a language $\mathcal{L}$ is strongly locally stable, then it is locally stable.*

In fact, strong local stability is a notion strictly stronger than local stability, although they coincide on bounded-degree graphs (cf. the full version). Thanks to the characterization in Theorem 8, and to the sufficient condition of Lemma 10, we immediately get error-sensitiveness for the language

$$\text{LEADER} = \big\{(G, \ell) : \forall v \in V(G),\ \ell(v) \in \{0, 1\} \text{ and there exists a unique } v \in V(G) \text{ for which } \ell(v) = 1\big\}.$$

▶ **Corollary 11.** LEADER *admits an error-sensitive proof-labeling scheme.*

Also, one can show that the language ST_l of spanning trees, whenever encoded by adjacency lists, admits an error-sensitive proof-labeling scheme.

This is in contrast to Proposition 6 (see proof in the full version).

▶ **Corollary 12.** ST_l *admits an error-sensitive proof-labeling scheme.*

Also, Theorem 8 allows us to prove (see proof in the full version) that minimum-weight spanning tree (MST) is error-sensitive (whenever the tree is encoded locally by adjacency lists). More specifically, let

$$\mathrm{MST}_l = \Big\{(G,\ell) : \forall v \in V(G),\ \ell(v) \subseteq N(v) \text{ and } \Big(\bigcup_{v\in V(G)} \bigcup_{u\in\ell(v)} \{u,v\}\Big) \text{ forms a MST}\Big\}. \quad (1)$$

▶ **Corollary 13.** MST_l *admits an error-sensitive proof-labeling scheme.*

5 Compact error-sensitive proof-labeling schemes

The characterization of Theorem 8 together with Lemma 4 implies an upper bound of $O(n^2)$ bits on the certificate size for the design of error-sensitive proof-labeling schemes for locally stable distributed languages. In this section, we show that the certificate size can be drastically reduced in certain cases. It is known that spanning tree and minimum spanning tree have proof-labeling schemes using certificates of polylogarithmic size $\Theta(\log n)$ bits [4, 22], and $\Theta(\log^2 n)$ bits [20], respectively. We show the proof-labeling schemes for spanning tree and MST in [4, 20, 22] are actually error-sensitive.

Recall that Proposition 6 proved that spanning tree does not admit any error-sensitive proof-labeling schemes whenever the tree is encoded at each node by a pointer to its parent: ST_p does not have any error-sensitive proof-labeling scheme. However, we show that ST_l, i.e., the language of spanning trees encoded by adjacency lists, does have a very compact error-sensitive proof-labeling scheme.

▶ **Theorem 14.** ST_l *has an error-sensitive proof-labeling scheme with certificates of size* $O(\log n)$ *bits.*

For figures that illustrate the construction of the following proof see the full version.

Proof. We show that the classical proof-labeling scheme $(\mathbf{p}, \mathbf{v})$ for ST_l is error-sensitive. On instances of the language, i.e., on labeled graphs (G,ℓ) where ℓ encodes a spanning tree T of G, the prover $\mathbf{p}$ chooses an arbitrary root r of T, and then assigns to every node u a certificate $(I(u), P(u), d(u))$ where $I(u) = \mathsf{ID}(r)$, $P(u)$ is the ID of the parent of u in the tree (or $ID(u)$ if u is the root), and $d(u)$ the hop-distance in the tree from u to r. The verifier $\mathbf{v}$ at every node u first checks that:

- the adjacency lists are consistent, that is, if u is in the list of v, then v is in the list of u;
- there exists a neighbor of u with ID $P(u)$, we denote it $p(u)$;
- the node u has the same root-ID $I(u)$ as all its neighbors in G;
- $d(u) \geq 0$.

Then, the verifier checks that:

- if $\mathsf{ID}(u) \neq I(u)$ then $d(p(u)) = d(u)-1$, and for every other neighbor w in ℓ, $d(w) = d(u)+1$ and $p(w) = u$;
- if $\mathsf{ID}(u) = I(u)$ then $P(u) = ID(u)$, $d(u) = 0$, and every neighbor w of u in ℓ satisfies $d(w) = d(u) + 1$ and $p(w) = u$.

By construction, if $(G,\ell) \in \mathrm{ST}_l$, then $\mathbf{v}$ accepts at every node. Also, it is easy to check that if $(G,\ell) \notin \mathrm{ST}_l$, then, for every certificate function c, at least one node rejects.

To establish error-sensitivity for the above proof-labeling scheme, let us assume that $\mathbf{v}$ rejects at $k \geq 1$ nodes with some certificate function c. Then, let (G',ℓ') be the labeled graph coinciding with (G,ℓ) except that all edges for which $\mathbf{v}$ rejects at both endpoints are

removed both from G, and from the adjacency lists in ℓ of the endpoints of these edges. Note that modifying ℓ into ℓ' only requires to edit labels of nodes that are rejecting. The graph G' may be disconnected. Let (C, ℓ'_C) be a connected component of (G', ℓ').

We claim that the edges of ℓ'_C form a forest in C. First note that if there is a cycle in the edges of ℓ'_C, then this cycle already existed in ℓ because we have added no edges when transforming ℓ into ℓ'. Consider such a cycle in ℓ, and the certificates given by $\mathbf{p}$. Either an edge is not oriented, that is no node uses this edge to point to its parent, or the cycle is consistently oriented and then distances are not consistent. In both cases two adjacent nodes of the cycle would reject when running $\mathbf{v}$. Then this cycle cannot be present in ℓ'_C, as at least one edge has been removed. As a consequence ℓ'_C form a forest of C. In C, if a node is connected to no other node by an edge of ℓ'_C, we will consider it as a tree of one node. With this convention, ℓ' is a spanning forest of G'.

We will now bound the number of trees in ℓ' by a function of k. The number of trees in ℓ' is equal to the sum of the number of trees in each component (C, ℓ'_C).

Let us run $\mathbf{v}$ on graph (C, ℓ'_C), and let k_C be the number of rejecting nodes. Observe that for every two nodes u and v in a component C, it holds that $I(u) = I(v)$. Indeed, otherwise, there would exist two adjacent nodes u and v in C with $I(u) \neq I(v)$, resulting in $\mathbf{v}$ rejecting at both nodes, which would yield the removal of $\{u, v\}$ from G. Consequently, at most one tree of ℓ'_C has a root whose ID corresponds to the ID given in the certificate. Then the number of trees in ℓ'_C is bounded by $k_C + 1$, and the total number of trees is bounded $\sum_C k_C + 1 = (\sum_C k_C) + |C|$.

Note that because of the design of the proof-labeling scheme, the nodes that accept when running $\mathbf{v}$ on (G, ℓ) also accept in (G', ℓ'). Then $\sum_C k_C \leq k$.

Let V_C be the set of nodes of C. It is easy to see that for all C, there exists a node of V_C that rejects when we run $\mathbf{v}$ on (G, ℓ). Indeed if there is no rejecting node, then no edge between C and the rest of the graph is removed, and then there is only one component in the graph. But then all node accept, which contradict the fact that $k \geq 1$. Then $|C| \leq k$.

So overall all ℓ' encodes a spanning forest with at most $2k$ trees. Such a labeling can thus be modified to get a spanning tree by modifying the labels of at most $4k$ nodes. That is, $(\mathbf{p}, \mathbf{v})$ is error-sensitive with parameter $\alpha \geq \frac{1}{4}$. ◀

Finally, we show that the compact proof-labelling scheme in [20, 22] for minimum-weight spanning tree, as specified in Eq. (1) of Section 4 is error-sensitive when the edge weights are distinct.

▶ **Theorem 15.** MST_l *admits an error-sensitive proof-labeling scheme with certificates of size* $O(\log^2 n)$ *bits.*

Hereafter, we provide a sketch of proof for Theorem 15 (the complete proof is deferred to the full version).

Sketch of proof. A classic proof-labeling scheme for MST (see, e.g., [19, 20, 22]) consists in encoding a run of Borůvka algorithm. Recall that Borůvka algorithm maintains a spanning forest whose trees are called fragments, starting with the forest in which every node forms a fragment. The algorithm proceeds in a sequence of steps. At each step, it selects the lightest outgoing edge from every fragment of the current forest, and adds all these edges to the MST, while merging the fragments linked by the selected edges. This algorithm eventually produces a single fragment, which is a MSTof the whole graph, after at most a logarithmic number of steps.

At each node u, the certificate of the scheme consists of a table with a logarithmic number of fields, one for each round of Borůvka algorithm. For each step of Borůvka algorithm, the corresponding entry of the table provides a proof of correctness for the fragment including u, plus the certificate of a tree pointing to the lightest outgoing edge of the fragment. The verifier verifies the structures of the fragments, and the fact that no outgoing edges from each fragment have smaller weights than the one given in the certificate. It also checks that the different fields of the certificate are consistent (for instance, it checks that, if two adjacent nodes are in the same fragment at step i, then they are also in the same fragment at step $i+1$).

To prove that this classic scheme is error-sensitive, we perform the same decomposition as in the proof of Theorem 14, removing the edges that have both endpoints rejecting. We then consider each connected component C of the remaining graph, together with the subgraph S of that component described by the edges of the given labeling. In general, S is not a MST of the component C (S can even be disconnected). Nevertheless, we can still make use of the key property that the subgraph S is not arbitrarily far from a MST of the component C. Indeed, the edges of S form a forest, and these edges belong to a MST of the component. As a consequence, it is sufficient to add a few edges to S for obtaining a MST. To show that S is indeed not far from being a MST of C, we define a relaxed version of Borůvka algorithm, and show that the labeling of the nodes corresponds to a proper run of this modified version of Borůvka algorithm. We then show how to slightly modify both the run of the modified Borůvka algorithm, and the labeling of the nodes, to get a MST of the component. Finally, we prove that the collection of MSTs of the components can be transformed into a MST of the whole graph, by editing a few node labels only. ◀

6 Conclusion

In this paper, we consider on a stronger notion of proof-labeling scheme, named *error-sensitive* proof-labeling scheme, and provides a structural characterization of the distributed languages that can be verified using such a scheme in distributed networks. This characterization highlights the fact that some basic network properties do *not* have error-sensitive proof-labeling schemes, which is in contrast to the fact that every property has a proof-labeling scheme. However, important network properties, like acyclicity, leader, spanning tree, MST, etc., do admit error-sensitive proof-labeling schemes. Moreover, these schemes can be designed with the same certificate size as the one for the classic proof-labeling schemes for these properties.

Our study of error-sensitive proof-labeling schemes raises intriguing questions. In particular, we observed that every distributed languages seems to fit in one of the following two scenarios: either it does not admit error-sensitive proof-labeling schemes, or it admits error-sensitive proof-labeling schemes with the same certificate size as the most compact proof-labeling schemes known for this language. We do not know whether there exists a distributed language admitting error-sensitive proof-labeling schemes, but such that all error-sensitive proof-labeling schemes for that language use certificates larger than the ones used for the most compact proof-labeling schemes for that language.

Proximity-sensitivity. Another desirable property for a proof-labeling scheme is *proximity-sensitivity*, requiring that every error is detected by a node close to that error. Proximity-sensitivity appears to be a very demanding notion, even stronger than error-sensitivity, for the former implies the later whenever the errors are spread out in the network. It would be

informative to provide a structural characterization of the distributed languages that can be verified using proximity-sensitive proof-labeling schemes, and at which cost in term of label size.

References

1 Yehuda Afek and Shlomi Dolev. Local stabilizer. *J. Parallel Distrib. Comput.*, 62(5):745–765, 2002. doi:10.1006/jpdc.2001.1823.

2 Yehuda Afek, Shay Kutten, and Moti Yung. The local detection paradigm and its application to self-stabilization. *Theor. Comput. Sci.*, 186(1-2):199–229, 1997. doi:10.1016/S0304-3975(96)00286-1.

3 Baruch Awerbuch, Shay Kutten, Yishay Mansour, Boaz Patt-Shamir, and George Varghese. Time optimal self-stabilizing synchronization. In *25th ACM Symposium on Theory of Computing (STOC)*, pages 652–661, 1993. doi:10.1145/167088.167256.

4 Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-stabilization by local checking and correction (extended abstract). In *32nd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 268–277, 1991. doi:10.1109/SFCS.1991.185378.

5 Joffroy Beauquier, Sylvie Delaët, Shlomi Dolev, and Sébastien Tixeuil. Transient fault detectors. *Distributed Computing*, 20(1):39–51, 2007. doi:10.1007/s00446-007-0029-x.

6 Lélia Blin and Pierre Fraigniaud. Space-optimal time-efficient silent self-stabilizing constructions of constrained spanning trees. In *35th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 589–598, 2015. doi:10.1109/ICDCS.2015.66.

7 Lélia Blin, Pierre Fraigniaud, and Boaz Patt-Shamir. On proof-labeling schemes versus silent self-stabilizing algorithms. In *16th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 18–32, 2014. doi:10.1007/978-3-319-11764-5_2.

8 Zvika Brakerski and Boaz Patt-Shamir. Distributed discovery of large near-cliques. *Distributed Computing*, 24(2):79–89, 2011.

9 Keren Censor-Hillel, Eldar Fischer, Gregory Schwartzman, and Yadu Vasudev. Fast distributed algorithms for testing graph properties. In *30th International Symposium on Distributed Computing (DISC)*, pages 43–56, 2016. doi:10.1007/978-3-662-53426-7_4.

10 Guy Even, Moti Medina, and Dana Ron. Best of two local models: Local centralized and local distributed algorithms. Technical report, arXiv CoRR abs/1402.3796, 2014.

11 Laurent Feuilloley and Pierre Fraigniaud. Survey of distributed decision. *Bulletin of the EATCS*, 119, 2016.

12 Pierre Fraigniaud, Amos Korman, and David Peleg. Towards a complexity theory for local distributed computing. *J. ACM*, 60(5):35:1–35:26, 2013. doi:10.1145/2499228.

13 Pierre Fraigniaud, Ivan Rapaport, Ville Salo, and Ioan Todinca. Distributed testing of excluded subgraphs. In *30th Int. Symposium on Distributed Computing (DISC)*, volume 9888 of *LNCS*, pages 342–356. Springer, 2016.

14 Oded Goldreich. A brief introduction to property testing. In *Studies in Complexity and Cryptography – Miscellanea on the Interplay between Randomness and Computation*, number 6650 in LNCS, pages 465–469. Springer, 2011. doi:10.1007/978-3-642-22670-0_31.

15 Oded Goldreich. Introduction to testing graph properties. In *Studies in Complexity and Cryptography – Miscellanea on the Interplay between Randomness and Computation*, number 6650 in LNCS, pages 470–506. Springer, 2011. doi:10.1007/978-3-642-22670-0_32.

16 Mika Göös, Juho Hirvonen, Reut Levi, Moti Medina, and Jukka Suomela. Non-local probes do not help with many graph problems. In *30th International Symposium on Distributed Computing (DISC)*, pages 201–214, 2016. doi:10.1007/978-3-662-53426-7_15.

17 Mika Göös and Jukka Suomela. Locally checkable proofs in distributed computing. *Theory of Computing*, 12(19):1–33, 2016. doi:10.4086/toc.2016.v012a019.

18 Tom Gur and Ron D. Rothblum. Non-interactive proofs of proximity. In *6th Conference on Innovations in Theoretical Computer Science (ITCS)*, pages 133–142, 2015. doi:10.1145/2688073.2688079.

19 Liah Kor, Amos Korman, and David Peleg. Tight bounds for distributed MST verification. In *28th International Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 69–80, 2011. doi:10.4230/LIPIcs.STACS.2011.69.

20 Amos Korman and Shay Kutten. Distributed verification of minimum spanning trees. *Distributed Computing*, 20(4):253–266, 2007. doi:10.1007/s00446-007-0025-1.

21 Amos Korman, Shay Kutten, and Toshimitsu Masuzawa. Fast and compact self-stabilizing verification, computation, and fault detection of an MST. *Distributed Computing*, 28(4):253–295, 2015. doi:10.1007/s00446-015-0242-y.

22 Amos Korman, Shay Kutten, and David Peleg. Proof labeling schemes. *Distributed Computing*, 22(4):215–233, 2010. doi:10.1007/s00446-010-0095-3.

23 Nathan Linial. Locality in distributed graph algorithms. *SIAM J. Comput.*, 21(1):193–201, 1992. doi:10.1137/0221015.

24 László Lovász and Katalin Vesztergombi. Non-deterministic graph property testing. *Combinatorics, Probability & Computing*, 22(5):749–762, 2013. doi:10.1017/S0963548313000205.

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

26 Michal Parnas and Dana Ron. Approximating the minimum vertex cover in sublinear time and a connection to distributed algorithms. *Theoretical Compututer Science*, 381(1-3):183–196, 2007. doi:10.1016/j.tcs.2007.04.040.

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

Improved Deterministic Distributed Matching via Rounding*

Manuela Fischer

ETH Zurich, Switzerland
manuela.fischer@inf.ethz.ch

Abstract

We present improved deterministic distributed algorithms for a number of well-studied matching problems, which are simpler, faster, more accurate, and/or more general than their known counterparts. The common denominator of these results is a *deterministic distributed rounding* method for certain *linear programs*, which is the first such rounding method, to our knowledge. A sampling of our end results is as follows.

- An $O\big(\log^2 \Delta \cdot \log n\big)$-round deterministic distributed algorithm for computing a maximal matching, in n-node graphs with maximum degree Δ. This is the first improvement in about 20 years over the celebrated $O(\log^4 n)$-round algorithm of Hańćkowiak, Karoński, and Panconesi [SODA'98, PODC'99].
- A deterministic distributed algorithm for computing a $(2+\varepsilon)$-approximation of maximum matching in $O\big(\log^2 \Delta \cdot \log \frac{1}{\varepsilon} + \log^* n\big)$ rounds. This is exponentially faster than the classic $O(\Delta + \log^* n)$-round 2-approximation of Panconesi and Rizzi [DIST'01]. With some modifications, the algorithm can also find an ε-maximal matching which leaves only an ε-fraction of the edges on unmatched nodes.
- An $O\big(\log^2 \Delta \cdot \log \frac{1}{\varepsilon} + \log^* n\big)$-round deterministic distributed algorithm for computing a $(2+\varepsilon)$-approximation of a maximum weighted matching, and also for the more general problem of maximum weighted b-matching. These improve over the $O\big(\log^4 n \cdot \log_{1+\varepsilon} W\big)$-round $(6+\varepsilon)$-approximation algorithm of Panconesi and Sozio [DIST'10], where W denotes the maximum normalized weight.
- A deterministic local computation algorithm for a $(2+\varepsilon)$-approximation of maximum matching with $2^{O(\log^2 \Delta)} \cdot \log^* n$ queries. This improves almost exponentially over the previous deterministic constant approximations which have query-complexity of $2^{\Omega(\Delta \cdot \log \Delta)} \cdot \log^* n$.

A full version of this paper with all proofs is available on arXiv.org [9].

1998 ACM Subject Classification G.2.2 Graph Algorithms

Keywords and phrases distributed graph algorithms, deterministic distributed algorithms, rounding linear programs, maximal matching, maximum matching approximation

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.17

1 Introduction and Related Work

We work with the standard LOCAL model of distributed computing [24]: the network is abstracted as a graph $G = (V, E)$, with $n = |V|$, $m = |E|$, and maximum degree Δ. Each node has a unique identifier. In each round, each node can send a message to each of its neighbors. We do not limit the message sizes, but for all the algorithms that we present,

* A full version of the paper is available at http://arxiv.org/abs/1703.00900.


© Manuela Fischer;
licensed under Creative Commons License CC-BY

$O(\log n)$-bit messages suffice. We assume that all nodes have knowledge of $\log \Delta$ up to a constant factor. If this is not the case, it is enough to try exponentially increasing estimates for $\log \Delta$.

1.1 Broader Context, and Deterministic Distributed Rounding

Efficient deterministic distributed graph algorithms remain somewhat of a rarity, despite the intensive study of the area since the 1980's. In fact, among the four classic problems of the area – maximal independent set, $(\Delta + 1)$-vertex-coloring, maximal matching, and $(2\Delta - 1)$-edge-coloring – only for maximal matching a poly $\log n$-round deterministic algorithm is known, due to a breakthrough of Hańćkowiak, Karoński, and Panconesi [14, 16]. Finding poly $\log n$-round deterministic algorithms for the other three problems remains a long-standing open question, since [24]. In a stark contrast, in the world of randomized algorithms, all these problems have $O(\log n)$-round [28, 1] or even more efficient algorithms [3, 10, 17].

Despite this rather bleak state of the art for deterministic algorithms, there is immense motivation for them. Here are three sample reasons: (1) One traditional motivation is rooted in the classic complexity-theoretic quest which seeks to understand the difference between the power of randomized and distributed algorithms. (2) Another traditional motivation comes from practical settings where even small error probabilities cannot be tolerated. (3) Nowadays, there is also a more modern motive: we now understand that in order to have faster randomized algorithms, we *must* come up with faster deterministic algorithms.[1] This connection goes in two directions: (A) Almost all the recent developments in randomized algorithms use the *shattering technique* [3, 10, 17, 12] which randomly breaks down the graph into small components, typically of size poly $\log n$, and then solves them via a deterministic algorithm. Speeding up (the n-dependency in) these randomized algorithms needs faster deterministic algorithms. (B) The more surprising direction is the reverse. Chang et al. [4] recently showed that for a large class of problems the randomized complexity on n-node graphs is at least the deterministic complexity on $\Theta\big(\sqrt{\log n}\big)$-node graphs. Hence, if one improves over (the n-dependency in) the current randomized algorithms, one has inevitably improved the corresponding deterministic algorithm.

Ghaffari, Kuhn, and Maus [11] recently proved a *completeness*-type result which shows that *"the only obstacle"* for efficient deterministic distributed graph algorithms is deterministically *rounding* fractional values to integral values while approximately preserving some linear constraints.[2] To put it more positively, if we find an efficient deterministic method for rounding, we would get efficient algorithms for essentially all the classic local graph problems, including the four mentioned above.

Our results become more instructive when viewed in this context. The common denominator of our results is a deterministic distributed method which allows us to round fractional matchings to integral matchings. This can be more generally seen as rounding the fractional solutions of a special class of *linear programs* (LPs) to integral solutions. To the best of our knowledge, this is the first known *deterministic distributed rounding* method. We can now say that

> *matching admits an efficient deterministic algorithm because*
> *matching admits an efficient deterministic distributed rounding.*

[1] For instance, our improvement in the deterministic complexity of maximal matching directly improves the randomized complexity of maximal matching, as we formally state in Corollary 3.

[2] Stating this result formally and in full generality requires some definitions. See [11] for the precise statement.

1.2 Our Results

We provide improved distributed algorithms for a number of matching problems, as we overview next.

1.2.1 Approximate Maximum Matching

▶ **Theorem 1.** *There is an $O\big(\log^2 \Delta \cdot \log \frac{1}{\varepsilon} + \log^* n\big)$-round deterministic distributed algorithm for a $(2+\varepsilon)$-approximate maximum matching, for any $\varepsilon > 0$.*

There are three remarks in order, regarding this result:

- For constant $\varepsilon > 0$, this $O(\log^2 \Delta + \log^* n)$-round algorithm is significantly faster than the previously best known deterministic constant approximations, especially in low-degree graphs: the $O(\Delta + \log^* n)$-round 2-approximation of Panconesi and Rizzi [30], the $O(\log^4 n)$-round 2-approximation of Hańćkowiak et al. [16], the $O(\log^4 n)$-round $(3/2)$-approximation of Czygrinow et al. [6, 7], and its extension [5] which finds a $(1+\varepsilon)$-approximation in $\log^{O\left(\frac{1}{\varepsilon}\right)} n$ rounds.
- This $O(\log^2 \Delta + \log^* n)$-round complexity gets close to the lower bound – due to the celebrated results of Kuhn et al. [22, 23] and Linial [24] – of $\Omega(\log \Delta / \log\log \Delta + \log^* n)$ that holds for any constant approximation of matching, even for randomized algorithms.
- This distributed LOCAL algorithm can be transformed to a deterministic *Local Computation Algorithm* (LCA) [2, 33] for a $(2+\varepsilon)$-approximation of maximum matching, with a query complexity of $2^{O(\log^3 \Delta)} \cdot \log^* n$. This is essentially by using the standard method of Parnas and Ron [32], with an additional idea of [8]. Using slightly more care, the query complexity can be improved to $2^{O(\log^2 \Delta)} \cdot \log^* n$. Since formally stating this result requires explaining the computational model of LCAs, we defer that to the journal version. We remark that this query complexity improves almost exponentially over the previous deterministic constant approximations with $2^{\Omega(\Delta \cdot \log \Delta)} \cdot \log^* n$ [8].

1.2.2 (Almost) Maximal Matching, and Edge Dominating Set

Maximal Matching. Employing our approximation algorithm for maximum matching, we get an $O(\log^2 \Delta \cdot \log n)$-round deterministic distributed algorithm for maximal matching.

▶ **Theorem 2.** *There is an $O(\log^2 \Delta \cdot \log n)$-round deterministic maximal matching algorithm.*

This is the first improvement in about 20 years over the breakthroughs of Hańćkowiak et al., which presented first an $O(\log^7 n)$- [14] and then an $O(\log^4 n)$-round [16] algorithm for maximal matching.

As alluded to before, this improvement in the deterministic complexity directly leads to an improvement in the n-dependency of the randomized algorithms. In particular, plugging in our improved deterministic algorithm in the maximal matching algorithm of Barenboim et al. [3] improves their round complexity from $O(\log^4 \log n + \log \Delta)$ to $O(\log^3 \log n + \log \Delta)$.

▶ **Corollary 3.** *There is an $O(\log^3 \log n + \log \Delta)$-round randomized distributed algorithm that with high probability[3] computes a maximal matching.*

[3] As standard, *with high probability* means with probability at least $1 - 1/n^c$, for a desirably large constant $c \geq 2$.

Almost Maximal Matching. Recently, there has been quite some interest in characterizing the Δ-dependency in the complexity of maximal matching, either with no dependency on n at all or with at most an $O(\log^* n)$ additive term [18, 13]. Göös et al. [13] conjectured that

there should be no $o(\Delta) + O(\log^* n)$ algorithm for computing a maximal matching.

Theorem 2 does not provide any news in this regard, because of its multiplicative $\log n$-factor. Indeed, our findings also seem to be consistent with this conjecture and do not suggest any way for breaking it. However, using some extra work, we can get a faster algorithm for ε-maximal matching, a matching that leaves only ε-fraction of edges among unmatched nodes, for a desirably small $\varepsilon > 0$.

▶ **Theorem 4.** *There is an $O\big(\log^2 \Delta \cdot \log \frac{1}{\varepsilon} + \log^* n\big)$-round deterministic distributed algorithm for an ε-maximal matching, for any $\varepsilon > 0$.*

This theorem statement is interesting because of two aspects: (1) This faster almost maximal matching algorithm sheds some light on the difficulties of proving the aforementioned conjecture. In a sense, any conceivable proof of this conjectured lower bound must distinguish between maximal and almost maximal matchings and rely on the fact that precisely a maximal matching is desired, and not just something close to it. Notice that since the complexity of Theorem 4 grows slowly as a function of ε, we can choose ε quite small. By setting $\varepsilon = \Delta^{-\operatorname{poly}\log \Delta}$, we get an algorithm that, in $O(\operatorname{poly}\log \Delta + \log^* n)$ rounds, produces a matching that seems to be maximal for almost all nodes, even if they look up to their $\operatorname{poly}\log \Delta$-hop neighborhood. (2) Perhaps, in some practical settings, this almost maximal matching, which practically looks maximal for essentially all nodes, may be as useful as maximal matching, especially since it can be computed much faster.

Approximate Minimum Edge Dominating Set. As a corollary of the almost maximal matching algorithm of Theorem 4, we get a fast algorithm for approximating *minimum edge dominating set*, which is the smallest set of edges such that any edge shares at least one endpoint with them. The proof can be found in the full version [9].

▶ **Corollary 5.** *2+eps-approx-EDS There is an $O(\log^2 \Delta \cdot \log \frac{\Delta}{\varepsilon} + \log^* n)$-round deterministic distributed algorithm for a $(2+\varepsilon)$-approximate minimum edge dominating set, for any $\varepsilon > 0$.*

Previously, the fastest algorithms ran in $O(\Delta + \log^* n)$ rounds [30] or $O(\log^4 n)$ rounds [16], providing 2-approximations. Moreover, Suomela [34] provided roughly 4-approximations in $O(\Delta^2)$ rounds, in a restricted variant of the LOCAL model with only port numberings.

1.2.3 Approximate Maximum Weighted Matching and B-Matching

An interesting aspect of the method we use is its flexibility and generality. In particular, the algorithm of Theorem 1 can be easily extended to computing a $(2+\varepsilon)$-approximation of maximum weighted matching, and more interestingly, to a $(2+\varepsilon)$-approximation of maximum weighted *b-matching*. These extensions can be found in the full version [9].

▶ **Theorem 6.** *There is an $O(\log^2 \Delta \cdot \log \frac{1}{\varepsilon} + \log^* n)$-round deterministic distributed algorithm for a $(2+\varepsilon)$-approximate maximum weighted matching, or b-matching, for any $\varepsilon > 0$.*

To the best of our knowledge, this is the first distributed deterministic algorithm for approximating maximum (weighted) b-matching. Moreover, even in the case of standard matching, it improves over the previously best-known algorithm: A deterministic algorithm

for $(6+\varepsilon)$-approximation of maximum weighted matching was provided by Panconesi and Sozio [31], with a round complexity of $O\big(\log^4 n \cdot \log_{1+\varepsilon} W\big)$, where W denotes the maximum normalized weight. However, that deterministic algorithm does not extend to b-matching.

1.3 Related Work, Randomized Distributed Matching Approximation

Aside from the deterministic algorithms discussed above, there is a long line of research on randomized distributed approximation algorithms for matching: for the unweighted case, [19] provide a 2-approximation in $O(\log n)$ rounds, and [26] a $(1+\varepsilon)$-approximation in $O(\log n)$ for any constant $\varepsilon > 0$. For the weighted case, [35, 27, 26] provide successively improved algorithms, culminating in the $O(\log \frac{1}{\varepsilon} \cdot \log n)$-round $(2+\varepsilon)$-approximation of [26]. Moreover, [21] present an $O(\log n)$-round randomized algorithm for 2-approximate weighted b-matching.

2 Our Deterministic Rounding Method, in a Nutshell

The main ingredient in our results is a simple deterministic method for *rounding* fractional solutions to integral solutions. We believe that this *deterministic distributed rounding* will be of interest well beyond this paper. To present the flavor of our deterministic rounding method, here we overview it in a simple special case: we describe an $O(\log^2 \Delta)$-round algorithm for a constant approximation of the maximum unweighted matching in 2-colored bipartite graphs. The precise algorithm and proof appear in Section 4.1.1.

Fractional Solution. First, notice that finding a fractional approximate maximum matching is straightforward. In $O(\log \Delta)$ rounds, we can compute a fractional matching $\mathbf{x} \in [0,1]^m$ whose total value $\sum_e x_e$ is a constant approximation of maximum matching. One standard method is as follows: start with all edge values at $x_e = 2^{-\lceil \log \Delta \rceil}$. Then, for $O(\log \Delta)$ rounds, in each round raise all edge values x_e by a 2-factor, except for those edges that are incident to a node v such that $\sum_{e \in E(v)} x_e \geq 1/2$. Throughout, $E(v) := \{e \in E \colon v \in e\}$ denotes the set of edges incident to node v. One can easily see that this fractional matching has total value $\sum_e x_e$ within a 4-factor of the maximum matching.

Gradual Rounding. We gradually round this fractional matching $\mathbf{x} \in [0,1]^m$ to an integral matching $\mathbf{x}' \in \{0,1\}^m$ while ensuring that we do not lose much of the value, i.e., $\sum_e x'_e \geq (\sum_e x_e)/C$, for some constant C. We have $O(\log \Delta)$ rounding phases, each of which takes $O(\log \Delta)$ rounds. In each phase, we get rid of the smallest (non-zero) values and thereby move closer to integrality. The initial fractional matching has[4] only values $x_e = 2^{-i}$ for $i \in \{0, \ldots, \lceil \log \Delta \rceil\}$ or $x_e = 0$. In the k^{th} phase, we partially round the edge values $x_e = 2^{-i}$ for $i = \lceil \log \Delta \rceil - k + 1$. Some of these edges will be raised to $x_e = 2 \cdot 2^{-i}$, while others are dropped to $x_e = 0$. The choices are made in a way that keeps $\sum_e x_e$ essentially unchanged, as we explain next.

Consider the graph H edge-induced by edges e with value $x_e = 2^{-i}$. For the sake of simplicity, suppose all nodes of H have even degree. Dealing with odd degrees requires some delicate care, but it will not incur a loss worse than an $O\big(2^{-i}\big)$-fraction of the total value. In this even-degree graph H, we effectively want that for each node v of H, half of its edges

[4] Any fractional maximum matching can be transformed to this format, with at most a 2-factor loss in the total value: simply round down each value to the next power of 2, and then drop edges with values below $2^{-(\lceil \log \Delta \rceil + 1)}$.

raise $x_e = 2^{-i}$ to $x_e = 2 \cdot 2^{-i}$ while the others drop it to $x_e = 0$. For that, we generate a degree-2 graph H' by replacing each node v of H with $d_H(v)/2$ nodes, each of which gets two of v's edges[5]. Notice that the edge sets of H' and H are the same. Graph H' is simply a set of cycles of even length, as H was bipartite.

In each cycle of H', we would want that the raise and drop of edge weights is alternating. That is, odd-numbered, say, edges are raised to $x_e = 2 \cdot 2^{-i}$ while even-numbered edges are dropped to $x_e = 0$. This would keep $\mathbf{x}$ a valid fractional matching – meaning that each node v still has $\sum_{e \in E(v)} x_e \leq 1$ – because the summation $\sum_{e \in E(v)} x_e$ does not increase, for each node v. Furthermore, it would keep the total weight $\sum_e x_e$ unchanged. If the cycle is shorter than length $O(\log \Delta)$, this raise/drop sequence can be identified in $O(\log \Delta)$ rounds. For longer cycles, we cannot compute such a perfect alternation in $O(\log \Delta)$ rounds. However, one can do something that does not lose much[6]: imagine that we chop the longer cycles into edge-disjoint paths of length $\Theta(\log \Delta)$. In each path, we drop the endpoints to $x_e = 0$ while using a perfect alternation inside the path. These border settings mean we lose $\Theta(1/\log \Delta)$-fraction of the weight. Thus, even over all the $O(\log \Delta)$ iterations, the total loss is only a small constant fraction of the total weight.

3 Preliminaries

Matching and Fractional Matching. An integral matching M is a subset of E such that $e \cap e' = \emptyset$ for all $e \neq e' \in M$. It can be seen as an assignment of values $x_e \in \{0, 1\}$ to edges, where $x_e = 1$ iff $e \in M$, such that $c_v := \sum_{e \in E(v)} x_e \leq 1$ for all $v \in V$. When the condition $x_e \in \{0, 1\}$ is relaxed to $0 \leq x_e \leq 1$, such an assignment is called a fractional matching.

B-Matching. A b-matching for b-values $\{1 \leq b_v \leq d_G(v) \colon v \in V\}$ is an assignment of values $x_e \in \{0, 1\}$ to edges $e \in E$ such that $\sum_{e \in E(v)} x_e \leq b_v$ for all $v \in V$. Throughout, $d_G(v)$ denotes the degree of v in G. Again, one can relax this to fractional b-matchings by replacing $x_e \in \{0, 1\}$ with $0 \leq x_e \leq 1$.

Maximal and ε-Maximal Matching. An integral matching is called maximal if we cannot add any edge to it without violating the constraints. For $\varepsilon > 0$, we say that $M \subseteq E$ is an ε- maximal matching if $|\Gamma^+(M)| \geq (1 - \varepsilon)|E|$ for $\Gamma^+(M) := \{e \in E \mid \exists e' \in M \colon e \cap e' \neq \emptyset\}$, that is, if after removing the edges in and incident to M from G, at most $\varepsilon|E|$ edges remain.

Maximum and Approximate Maximum Matching. A matching M^* is called maximum if it is the/a largest matching in terms of cardinality. For any $c > 1$, we say that a matching is c-approximate if $c \sum_{e \in E} x_e \geq |M^*|$ for a maximum matching M^*. In a weighted graph where each edge e is assigned a weight $w_e \geq 0$, we say that M^* is a maximum weighted matching if it is the/a matching with maximum weight $w(M^*) := \sum_{e \in M^*} w_e$. An integral matching M is a c-approximate weighted matching if $c \sum_{e \in M} w_e \geq w(M^*)$.

We now state some simple and well-known facts about matchings.

▶ **Lemma 7.** *For a maximal matching M and a maximum matching M^* in $G = (V, E)$, we have the following two properties: (i) $|M| \geq \frac{|E|}{2\Delta - 1}$, and (ii) $\frac{|M^*|}{2} \leq |M| \leq |M^*|$.*

[5] This simple idea has been used frequently before. For instance, it gives an almost trivial proof of Petersen's 2-factorization theorem from 1891 [29]. It has also been used by [20, 14, 16].

[6] Our algorithm actually does something slightly different, but describing this ideal procedure is easier.

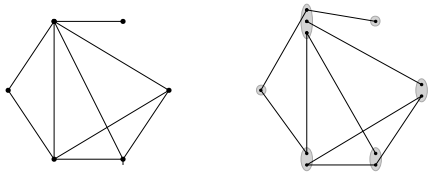


Figure 1 A graph and its 2-decomposition.

▶ **Lemma 8** (Panconesi and Rizzi [30]). *There is an $O(\Delta + \log^* n)$-round deterministic distributed algorithm for maximal matching. Furthermore, if a q-coloring of the graph is provided, then the algorithm runs in $O(\Delta + \log^* q)$ rounds.*

Many problems are easier in small-degree graphs. To exploit this fact, we sometimes use the following simple transformation which decomposes a graph into graphs with maximum degree 2 – that is, node-disjoint paths and cycles – with the same edge set, in zero rounds. As mentioned before, this has been used frequently in prior work [29, 20, 14, 16].

2-decomposition. We 2-*decompose* graph G as follows. For every node $v \in V$, introduce $\lceil \frac{d_G(v)}{2} \rceil$ copies and arbitrarily split its incident edges among these copies in such a way that every copy has degree 2, with the possible exception of one copy which has degree 1 (when v has odd degree). The graph on these copy nodes is what we call a 2-decomposition of G. See Figure 1 for an example.

4 Approximate Maximum Matching

We present a $(2+\varepsilon)$-approximation algorithm for maximum matching, proving Theorem 1. The first step towards this goal is finding a constant approximation, explained in Section 4.1. We show in Section 4.2 how to further improve this approximation ratio to $2+\varepsilon$.

4.1 Constant Approximate Maximum Matching

In this subsection, we show how to compute a constant approximation.

▶ **Lemma 9.** *There is an $O(\log^2 \Delta + \log^* n)$-round deterministic distributed algorithm for a c-approximate maximum matching, for some constant c.*

The key ingredient for our c-approximation algorithm of Lemma 9 is a distributed algorithm that computes a constant approximate maximum matching in the special case of a 2-colored bipartite graph. We first present the algorithm for this special case in Section 4.1.1, and then explain in Section 4.1.2 how to reduce the general graph case to the bipartite case, hence proving Lemma 9.

4.1.1 Constant Approximate Maximum Matching in Bipartite Graphs

Next, we show how to find a c-approximate matching in a 2-colored bipartite graph.

▶ **Lemma 10.** *There is an $O(\log^2 \Delta)$-round deterministic distributed algorithm for a c-approximate maximum matching in a 2-colored bipartite graph, for some constant c.*

Roadmap. The proof of Lemma 10 is split into three parts. In the first step, explained in Lemma 13, we compute a $2^{-\lceil \log \Delta \rceil}$-fractional 4-approximate maximum matching in $O(\log \Delta)$ rounds. The second step, which is also the main step of our method and is formalized in Lemma 14, is a method to round these fractional values to almost integrality in $O(\log^2 \Delta)$ rounds. In the third step, presented in Lemma 15, we resort to a simple constant-round algorithm to transform the almost integral matching that we have found up to this step into an integral matching. As a side remark, we note that we explicitly state some of the constants in this part of the paper, for the sake of readability. We remark that these constants are not the focus of this work, and we have not tried to optimize them.

We start with some helpful definitions.

▶ **Definition 11** (Loose and tight nodes and edges)**.** Given a fractional matching, we call a node v *loose* if $c_v = \sum_{e \in E(v)} x_e \leq \frac{1}{2}$, and *tight* otherwise, where $E(v) := \{e \in E \colon v \in e\}$. We call an edge *loose* if both of its endpoints are loose; otherwise, the edge is called *tight*.

▶ **Definition 12** (The fractionality of a fractional matching)**.** We call a fractional matching 2^{-i}-fractional for an $i \in \mathbb{N}$ if $x_e \in \{0\} \bigcup \left\{2^{-j} \colon 0 \leq j \leq i\right\}$. Notice that a 2^{-0}-fractional matching is simply an integral matching.

Step 1, Fractional Matching. We show that a simple greedy algorithm already leads to a fractional 4-approximate maximum matching.

▶ **Lemma 13.** *There is an $O(\log \Delta)$-round deterministic distributed algorithm for a $2^{-\lceil \log \Delta \rceil}$-fractional 4-approximate maximum matching.*

Proof. Initially, set $x_e = 2^{-\lceil \log \Delta \rceil}$ for all $e \in E$. This trivially satisfies the constraints $c_v = \sum_{e \in E(v)} x_e \leq 1$. Then, we iteratively raise the value of all loose edges in parallel by a 2-factor. This can be done in $O(\log \Delta)$ rounds, since at the latest when the value of an edge is $1/2$, both endpoints would be tight. Once all edges are tight, for a maximum matching M^* we have $\sum_{e \in E} x_e = \frac{1}{2} \sum_{v \in V} c_v \geq \frac{1}{2} \sum_{e=\{u,v\} \in M^*} (c_u + c_v) > \frac{|M^*|}{4}$. ◀

Step 2, Main Rounding. The heart of our approach, the Rounding Lemma, is a method that successively turns a 2^{-i}-fractional matching into a 2^{-i+1}-fractional one, for decreasing values of i, while only sacrificing the approximation ratio by a little.

▶ **Lemma 14** (Rounding Lemma)**.** *There is an $O\big(\log^2 \Delta\big)$-round deterministic distributed algorithm that transforms a $2^{-\lceil \log \Delta \rceil}$-fractional 4-approximate maximum matching in a 2-colored bipartite graph into a 2^{-4}-fractional 14-approximate maximum matching.*

Proof. Iteratively, for $k = 1, \ldots, \lceil \log \Delta \rceil - 4$, in phase k, we get rid of edges e with value $x_e = 2^{-i}$ for $i = \lceil \log \Delta \rceil - k + 1$ by either increasing their values by a 2-factor to $x_e = 2^{-i+1}$ or setting them to $x_e = 0$. In the following, we describe the process for one phase k, thus a fixed i.

Let H be the graph induced by the set $E_i := \{e \in E \colon x_e = 2^{-i}\}$ of edges with value 2^{-i} and use H' to denote its 2-decomposition. Notice that H' is a node-disjoint union of paths and even-length cycles. Set $\ell = 12 \log \Delta$. We call a path/cycle *short* if it has length at most ℓ, and *long* otherwise. We now process short and long cycles and paths, by distinguishing three cases, as we discuss next. Each of these cases will be done in $O(\log \Delta)$ rounds, which implies that the complexity of one phase is $O(\log \Delta)$. Thus, over all the $O(\log \Delta)$ phases, this rounding algorithm takes $O(\log^2 \Delta)$ rounds.

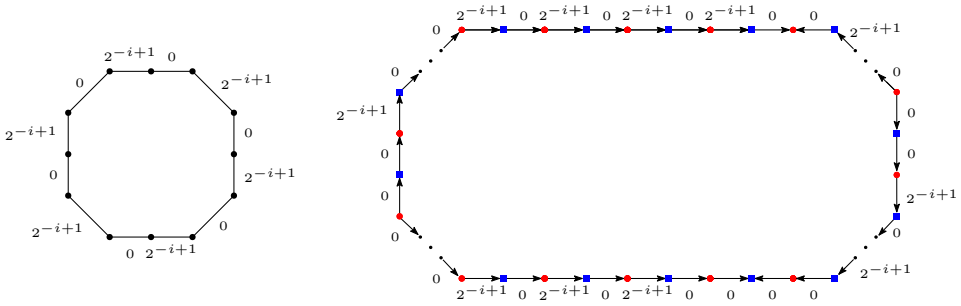


Figure 2 The edge values of a short and a long cycle induced by edges in E_i after rounding. In the long cycle, nodes of color 1 are depicted as blue squares and nodes of color 2 as red circles.

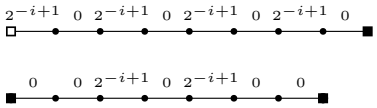


Figure 3 The edge values of two short paths induced by edges in E_i after rounding. Tight endpoints are depicted as (unfilled) boxes and loose endpoints as (filled) squares.

Case A, Short Cycles. Alternately set the values of the edges to 0 and to 2^{-i+1}. Since the cycle has even length, the values $c_v = \sum_{e \in E(v)} x_e$ for all nodes v in the cycle remain unaffected by this update. Moreover, the total value of the edges in the cycle stays the same.

Case B, Long Cycles and Long Paths. We first orient the edges in a manner that ensures that each maximal directed path has length at least ℓ. This is done in $O(\ell)$ rounds. For that purpose, we start with an arbitrary orientation of the edges. Then, for each $j = 1, \ldots, \lceil \log \ell \rceil$, we iteratively merge two (maximal) directed paths of length $< 2^j$ that are directed towards each other by reversing the shorter one, breaking ties arbitrarily. For more details of this orientation step, we refer to [15, Fact 5.2].

Given this orientation, we determine the new values of x_e as follows. Recall that we are given a 2-coloring of nodes. Set the value of all border edges (that is, edges that have an incident edge such that they are either oriented towards each other or away from each other) to 0, increase the value of a non-border edge to 2^{-i+1} if it is oriented towards a node of color 1, say, and set it to 0 otherwise.

Now, we show that this process generates a valid fractional matching while incurring only a small loss in the value. Observe that no constraint is violated, as for each node the value of at most one incident edge can be raised to 2^{-i+1} while the other is dropped to 0. Moreover, in each maximal directed path, we can lose at most $3 \cdot 2^{-i}$ in the total sum of edge values. This happens in the case of an odd-length path starting with a node of color 2. Hence, we lose at most a $\frac{3}{\ell}$-fraction of the total sum of the edge values in long cycles and long paths.

Case C, Short Paths. Give the path an arbitrary direction, that is, identify the first and the last node. Set the value of the first edge to 2^{-i+1} if the first node is loose, and to 0 otherwise. Alternately, starting with value 0 for the second edge, set the value of every even edge to 0 and of every odd edge to 2^{-i+1}. If the last edge should be set to 2^{-i+1} (that is, the path has odd length) but the last node is tight, set the value of that last edge to 0 instead.

We now discuss the validity of the new fractional matching. If a node v is in the interior of the path, that is, not one of the endpoints, then v can have at most one of its incident edges

increased to 2^{-i+1} while the other one decreases to 0. Hence the summation $c_v = \sum_{e \in E(v)} x_e$ does not increase. If v is the first or last node in the path, the value of the edge incident to v is increased only if v was loose, i.e., if $c_v = \sum_{e \in E(v)} x_e \leq \frac{1}{2}$. In this case, we still have $c_v = \sum_{e \in E(v)} x_e \leq 1$ after the increase, as the value of the edge raises by at most a 2-factor.

We now argue that the value of the matching has not decreased by too much during this update. For that, we group the edges into blocks of two consecutive edges, starting from the first edge. If the path has odd length, the last block consists of a single edge. It is easy to see that the block value, that is, the sum of the values of its two edges, of every interior (neither first nor last) block is unaffected.

If an endpoint v of a path is loose, the value of the block containing v remains unchanged or increases (in the case of an odd-length path ending in v). If v is tight, then the value of its block stays the same or decreases by 2^{-i+1}, which is at most a 2^{-i+2}-fraction of the value c_v.

This allows us to bound the loss in terms of these tight endpoints. The crucial observation is that every node can be endpoint of a short path at most once. This is because, in the 2-decomposition, a node can be the endpoint of a path only if it has a degree-1 copy. This happens only if it has odd degree, and in that case, it has exactly one degree-1 copy, hence, also exactly one endpoint of a short path. Therefore, we lose at most a 2^{-i+2}-fraction in $\sum_{v \in V} c_v$ when updating the values in short paths.

Analyzing the Overall Effect of Rounding. First, we show that over all the rounding phases, the overall loss is only a constant fraction of the total value $\sum_{e \in E} x_e$.

Let $x_e^{(i)}$ and $c_v^{(i)}$ denote the value of edge e and node v, respectively, before eliminating all the edges with value 2^{-i}. Putting together the loss analyses discussed above, we get

$$\sum_{e \in E} x_e^{(i-1)} \geq \sum_{e \in E} x_e^{(i)} - \frac{3}{\ell} \sum_{e \in E} x_e^{(i)} - 2^{-i+2} \sum_{v \in V} c_v^{(i)} \geq \left(1 - \frac{3}{\ell} - 2^{-i+3}\right) \sum_{e \in E} x_e^{(i)}.$$

It follows that

$$\begin{aligned}\sum_{e \in E} x_e^{(4)} &\geq \left(\prod_{i=5}^{\lceil \log \Delta \rceil} \left(1 - \frac{3}{\ell} - 2^{-i+3}\right)\right) \sum_{e \in E} x_e^{(\lceil \log \Delta \rceil)} \geq \left(\prod_{i=5}^{\lceil \log \Delta \rceil} e^{-2\left(\frac{3}{\ell} + 2^{-i+3}\right)}\right) \sum_{e \in E} x_e^{(\lceil \log \Delta \rceil)} \\ &\geq e^{-\frac{1}{4} - 16 \sum_{i=5}^{\lceil \log \Delta \rceil} 2^{-i}} \sum_{e \in E} x_e^{(\lceil \log \Delta \rceil)} \geq \frac{1}{e^{\frac{5}{4}}} \sum_{e \in E} x_e^{(\lceil \log \Delta \rceil)} \geq \frac{1}{4e^{\frac{5}{4}}} |M^*| \geq \frac{1}{14} |M^*|\end{aligned}$$

for a maximum matching M^*, recalling that we started with a 4-approximate maximum matching. Here, the second inequality holds because $\frac{3}{\ell} + 2^{-i+3} \leq \frac{1}{2}$, as $i \geq 5$.

Finally, observe that in all the rounding phases the constraints $c_v = \sum_{e \in E(v)} x_e \leq 1$ are preserved, since the value c_v can increase by at most a 2-factor and only when v is loose. ◀

Step 3, Final Rounding. So far, we have an almost integral matching. Next, we round all edges to either 0 or 1, by finding a maximal matching in the subgraph induced by edges with positive value.

▶ **Lemma 15.** *There is an $O(1)$-round deterministic distributed algorithm that, given a 2^{-4}-fractional 14-approximate maximum matching in a 2-colored bipartite graph, computes an integral 434-approximate maximum matching.*

Proof. In the given 2^{-4}-fractional matching, $x_e \neq 0$ means $x_e \geq \frac{1}{16}$. Thus, a node cannot have more than 16 incident edges with non-zero value in this fractional matching. In this constant-degree subgraph, a maximal matching M can be found in $O(1)$ rounds using the algorithm in Lemma 8, recalling that we are given a 2-coloring. We have $|M| \geq \frac{|\{e \in E \colon x_e > 0\}|}{31} \geq \frac{1}{31} \sum_{e \in E} x_e$ by Lemma 7 (i), and, since we started with a 14-approximation, M is 434-approximate. ◀

4.1.2 Constant Approximate Maximum Matching in General Graphs

We now explain how the approximation algorithm for maximum matchings in 2-colored bipartite graphs can be employed to find approximate maximum matchings in general graphs. The main idea is to transform the given general graph into a bipartite graph with the same edge set in such a way that a matching in this bipartite graph can be easily turned into a matching in the general graph.

Proof of Lemma 9. Let $\overrightarrow{E}$ be an arbitrary orientation of the edges E. Split every node $v \in V$ into two siblings v_{in} and v_{out}, and add an edge $\{u_{\text{out}}, v_{\text{in}}\}$ to E_B for every oriented edge $(u,v) \in \overrightarrow{E}$. Let $V_{\text{in}} := \{v_{\text{in}} \colon v \in V\}$ and $V_{\text{out}} := \{v_{\text{out}} \colon v \in V\}$ be the nodes with color 1 and 2, respectively. By Lemma 10, a c-approximate maximum matching M_B in the bipartite graph $B = (V_{\text{in}} \bigcup V_{\text{out}}, E_B)$ can be computed in $O\big(\log^2 \Delta\big)$ rounds. We now go back to V, that is, merge v_{in} and v_{out} back into v. This makes the edges of M_B incident to v_{in} or v_{out} now be incident to v, leaving us with a graph $G' = (V, M_B) \subseteq G$ with maximum degree 2.

We compute a maximal matching M' in G'. Using the algorithm of Lemma 8, this can be done in $O(\log^* n)$ rounds. If an poly Δ-coloring of G is provided, which implies a coloring of G' with poly Δ colors, the round complexity of this step is merely $O(\log^* \Delta)$.

It follows from Lemma 7 (i) that $|M'| \geq \frac{|M_B|}{3} \geq \frac{|M_B^*|}{3c} \geq \frac{|M^*|}{3c}$ for maximum matchings M_B^* in B and M^* in G, respectively. Thus, M' is a $3c$-approximate maximum matching in G. The last inequality is true since by introducing additional nodes but leaving the edge set unchanged (when going from G to B), the maximum matching size cannot decrease. ◀

4.2 Wrap-Up: $(2+\varepsilon)$-Approximate Matching and Maximal Matching

In this section, we iteratively invoke the constant approximation algorithm of 9 to obtain algorithms for a $(2+\varepsilon)$-approximate maximum matching (Theorem 1) and a maximal matching (Theorem 2).

The approximation ratio of a matching algorithm can be improved from c to $2+\varepsilon$ easily, by $O\big(\log \frac{1}{\varepsilon}\big)$ repetitions: each time, we apply the algorithm of Lemma 9 to the remaining graph, and remove the found matching together with its neighboring edges from the graph.

Before explaining the details, we present the following frequently used trick.

▶ Remark. If a poly Δ-coloring of a graph is provided, we can go around the $\Omega(\log^* n)$ lower bound of Linial [24], omitting the additive $O(\log^* n)$ term from the round complexity of the algorithms presented in this paper. More generally, if such an algorithm is invoked iteratively, one can first precompute an $O(\Delta^2)$-coloring in $O(\log^* n)$ rounds using Linial's algorithm [25], which allows us to replace the $O(\log^* n)$ term by $O(\log^* \Delta)$ by Lemma 8 in each iteration.

Proof of Theorem 1. Starting with $G_0 = G$, for $i = 0, \dots, k-1$, where $k = O\big(\log \frac{1}{\varepsilon}\big)$, iteratively compute a c-approximate maximum matching M_i in G_i, using the algorithm of Lemma 9. We delete M_i together with its incident edges from the graph, that is, set $G_{i+1} = (V, E(G_i) \setminus \Gamma^+(M_i))$.

Now, we argue that the obtained matching $\bigcup_{i=0}^{k-1} M_i$ is $(2+\varepsilon)$-approximate. To this end, we bound the size of a maximum matching in the remainder graph G_k.

Let M_i^* be a maximum matching in G_i. An inductive argument shows that $|M_i^*| \leq \big(1 - \frac{1}{c}\big)^i |M^*|$. Indeed, observe $|M_{i+1}^*| \leq |M_i^*| - |M_i| \leq \big(1 - \frac{1}{c}\big)|M_i^*| \leq \big(1 - \frac{1}{c}\big)^{i+1}|M^*|$, where the first inequality holds since otherwise $M_{i+1}^* \cup M_i$ would be a better matching than M_i^* in G_i, contradicting the latter's optimality. For $k = \log_{1-\frac{1}{c}} \frac{\varepsilon}{2(2+\varepsilon)}$, we thus have $|M_k^*| \leq \frac{\varepsilon}{2(2+\varepsilon)}|M^*|$. As $\bigcup_{i=0}^{k-1} M_i$ is a maximal matching in $G \setminus G_k$ by construction, $\Big(\bigcup_{i=0}^{k-1} M_i\Big) \cup M_k^*$

is a maximal matching in G. By Lemma 7 (ii), this means that $\left|\bigcup_{i=0}^{k-1} M_i\right| + |M_k^*| \geq \frac{|M^*|}{2}$, hence $\left|\bigcup_{i=0}^{k-1} M_i\right| \geq \left(\frac{1}{2} - \frac{\varepsilon}{2(2+\varepsilon)}\right)|M^*| \geq \frac{|M^*|}{2+\varepsilon}$.

We have $O\big(\log \frac{1}{\varepsilon}\big)$ iterations, each taking $O(\log^2 \Delta + \log^* n)$ rounds. As mentioned in Remark 4.2, by precomputing an $O(\Delta^2)$-coloring in $O(\log^* n)$ rounds, the round complexity of each iteration can be decreased to $O(\log^2 \Delta + \log^* \Delta) = O(\log^2 \Delta)$, leading to an overall running time of $O(\log^2 \Delta \cdot \log \frac{1}{\varepsilon} + \log^* n)$ rounds. ◀

▶ Remark. The analysis above shows that the matching M computed by the algorithm of Theorem 1 is not only $(2+\varepsilon)$-approximate, but also has the property that any matching in the remainder graph (induced by $E \setminus \Gamma^+(M)$) can have size at most $\varepsilon|M^*|$ for a maximum matching M^* in G.

If one increases the number of repetitions to $O(\log n)$, the found matching is maximal.

Proof of Theorem 2. Apply the c-approximation algorithm of Lemma 9 for $k = \log_{1-\frac{1}{c}} \frac{1}{n}$ iterations on the respective remainder graph, as described in the proof of Theorem 1. The same analysis (also adopting the notation from there) shows that a maximum matching M_k^* in the remainder graph G_k must have size $|M_k^*| \leq \frac{|M^*|}{n} < 1$, which means that G_k is an empty graph. But then $\bigcup_{i=1}^{k-1} M_i$ must be maximal. ◀

5 Almost Maximal Matching

In the previous section, we have seen how one can obtain a matching that reduces the size of the matching in the remainder graph, that is, the graph after removing the matching and all incident edges, by a constant factor. Intuitively, one would expect that this also reduces the number of remaining edges by a constant factor, which would directly lead to an (almost) maximal matching just by repetitions. However, this is not the case, since not every matched edge removes the same number of edges from the graph, particularly in non-regular graphs. This calls for an approach that weights edges incident to nodes of different degrees differently, which naturally brings into play weighted matchings.

In Lemma 16, we present a fast algorithm that finds a constant approximation of maximum weighted matching based on the algorithm of Theorem 1. Then, we use this algorithm, by assigning certain weights to the edges, to find a matching that removes a constant fraction of the edges in Lemma 17. Via $O\big(\log \frac{1}{\varepsilon}\big)$ repetitions of this, each time removing the found matching and its incident edges, we get an ε-maximal matching. More details are provided in the proof of Theorem 4 in the end of this section. Observe that when setting $\varepsilon = \frac{1}{n^2}$, thus increasing the number of repetitions to $O(\log n)$, we obtain a maximal matching.

▶ **Lemma 16.** *There is an $O\big(\log^2 \Delta + \log^* n\big)$-round deterministic distributed algorithm for a 256-approximate maximum weighted matching.*

Proof. We assume without loss of generality that the edge weights are normalized, that is, from a set $\{1, \ldots, W\}$ for some maximum weight W. Round the weights w_e for $e \in E$ down to the next power of 8, resulting in weights w'_e. This rounding procedure lets us lose at most a 8-factor in the total weight and provides us with a decomposition of G into graphs $C_i = (V, E_i)$ with $E_i := \{e \in E \colon w'_e = 8^i\}$ for $i \in \{0, \ldots, \lfloor \log_8 W \rfloor\}$.

In parallel, run the algorithm of Theorem 1 with $\varepsilon = 1$ on every C_i to find a 3-approximate maximum matching M_i in C_i in $O(\log^2 \Delta + \log^* n)$ rounds. Observe that while the edges in $\bigcup_i M_i$ do not form a matching, since edges from M_i and M_j for $i \neq j$ can be neighboring, a

matching $M \subseteq \bigcup_i M_i$ can be obtained by deleting all but the highest-index edge in every such conflict, that is, by removing all edges $e \in M_i$ with an incident edge $e' \in M_j$ for a $j > i$.

In the following, we argue that the weight of M cannot be too small compared to the weight of $\bigcup_i M_i$ by an argument based on counting in two ways.

Every edge $e \in (\bigcup_i M_i) \setminus M$ puts blame w'_e on an edge in M as follows. Since $e \notin M$, there is an edge e' incident to e such that $e \in M_i$ and $e' \in M_j$ for some $j > i$. If $e' \in M$, then e blames weight w_e on e'. If $e' \notin M$, then e puts blame w_e on the same edge as e' does.

For an edge $e \in M \cap E_i$ and $j \in [i]$, let n_j be the maximum number of edges from M_{i-j} that blame e. An inductive argument shows that $n_j \leq 2^j$. Indeed, there can be at most two edges from M_{i-1} blaming e, at most one per endpoint of e, and, for $j > 1$, we have $n_j \leq 2 + \sum_{j'=1}^{j-1} n_{j'} \leq 2 + \sum_{j'=1}^{j-1} 2^{j'} = 2^j$, since at most two edges in M_{i-j} can be incident to e and at most one further edge can be incident to each edge in $M_{i-j'}$ for $j' < j$.

Therefore, overall, at most $\sum_{j=1}^{i} 2^j 8^{i-j} \leq \frac{1}{3} 8^i \leq \frac{1}{3} w'_e$ weight is blamed on e. This means that $\sum_{e \in (\cup_i M_i) \setminus M} w'_e \leq \frac{1}{3} \sum_{e \in M} w'_e$, hence $\sum_{e \in \cup_i M_i} w'_e \leq \frac{4}{3} \sum_{e \in M} w'_e$, and lets us conclude that $\sum_{e \in M^*} w_e \leq 8 \sum_{e \in M^*} w'_e \leq 24 \sum_{e \in \cup_i M_i} w'_e \leq 32 \sum_{e \in M} w'_e \leq 256 \sum_{e \in M} w_e$ for a maximum weighted matching M^*. ◀

Next, we explain how to use this algorithm to remove a constant fraction of edges, by introducing appropriately chosen weights. We define the weight of each edge to be the number of its incident edges. This way, an (approximate) maximum weighted matching corresponds to a matching that removes a large number of edges.

▶ **Lemma 17.** *There is an $O\big(\log^2 \Delta + \log^* n\big)$-round deterministic distributed algorithm for a $\frac{511}{512}$-maximal matching.*

Proof. For each edge $e = \{u, v\} \in E$, introduce a weight $w_e = d_G(u) + d_G(v) - 1$, and apply the algorithm of Lemma 16 to find a 256-approximate maximum weighted matching M in G.

For the weight $w(M^*)$ of a maximum weighted matching M^*, it holds that $w(M^*) \geq |E|$, as the following simple argument based on counting in two ways shows. Let every edge in E put a blame on an edge in M^* that is responsible for its removal from the graph as follows. An edge $e \in M^*$ blames itself. An edge $e \notin M^*$ blames an arbitrary incident edge $e' \in M^*$. Notice that at least one such edge must exist, as otherwise M^* would not even be maximal. In this way, $|E|$ many blames have been put onto edges in M^* such that no edge $e = \{u, v\} \in M^*$ is blamed more than w_e times, as e can be blamed by itself and any incident edge. Therefore, indeed $w(M^*) = \sum_{e \in M^*} w_e \geq |E|$, and, as M is a 256-approximate, it follows that $\sum_{e \in M} w_e \geq \frac{|E|}{256}$.

Now, observe that w_e is the number of edges that are deleted when removing e together with its incident edges from G. Since every edge can be incident to at most two matched edges (and thus can be deleted by at most two edges in the matching), in total $|\Gamma^+(M)| \geq \frac{1}{2} \sum_{e \in M} w_e \geq \frac{|E|}{512}$ many edges are removed from G when deleting the edges in and incident to M, which proves that M is a $\frac{511}{512}$-maximal matching. ◀

We iteratively invoke this algorithm to successively reduce the number of remaining edges.

Proof of Theorem 4. For $i = 0, \ldots, k = O\big(\log \frac{1}{\varepsilon}\big)$ and $G_0 = G$, iteratively apply the algorithm of Lemma 17 to G_i to get a c-maximal matching M_i in G_i. Set $G_{i+1} = (V, E(G_i) \setminus \Gamma^+(M_i))$, that is, remove the matching and its neighboring edges from the graph. Then $M := \bigcup_{i=0}^{k-1} M_i$ for $k = \log_c \varepsilon$ is ε-approximate, since $|E \setminus \Gamma^+(M)| = |E(G_k)| \leq c^k |E| \leq \varepsilon |E|$, using $|E(G_{i+1})| \leq c\,|E(G_i)|$.

Overall, recalling Remark 4.2, this takes $O(\log^2 \Delta \cdot \log \frac{1}{\varepsilon} + \log^* n)$. ◀

Acknowledgment. I want to thank Mohsen Ghaffari for suggesting this topic, for his guidance and his support, as well as for the many valuable and enlightening discussions. I am also thankful to Seth Pettie for several helpful comments.

References

1 Noga Alon, László Babai, and Alon Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *Journal of algorithms*, 7(4):567–583, 1986.

2 Noga Alon, Ronitt Rubinfeld, Shai Vardi, and Ning Xie. Space-efficient local computation algorithms. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1132–1139, 2012.

3 Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The locality of distributed symmetry breaking. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 321–330, 2012.

4 Yi-Jun Chang, Tsvi Kopelowitz, and Seth Pettie. An exponential separation between randomized and deterministic complexity in the LOCAL model. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 615–624, 2016.

5 Andrzej Czygrinow and Michał Hańćkowiak. Distributed algorithm for better approximation of the maximum matching. In *International Computing and Combinatorics Conference*, pages 242–251, 2003.

6 Andrzej Czygrinow, Michał Hańćkowiak, and Edyta Szymańska. Distributed algorithm for approximating the maximum matching. *Discrete Applied Mathematics*, 143(1):62–71, 2004.

7 Andrzej Czygrinow, Michał Hańćkowiak, and Edyta Szymańska. A fast distributed algorithm for approximating the maximum matching. In *Proceedings of the Annual European Symposium on Algorithms (ESA)*, volume 3221, pages 252–263, 2004.

8 Guy Even, Moti Medina, and Dana Ron. Deterministic stateless centralized local algorithms for bounded degree graphs. In *Proceedings of the Annual European Symposium on Algorithms (ESA)*, pages 394–405, 2014.

9 Manuela Fischer. Improved deterministic distributed matching via rounding. *CoRR*, abs/1703.00900, 2017. URL: `http://arxiv.org/abs/1703.00900`.

10 Mohsen Ghaffari. An improved distributed algorithm for maximal independent set. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 270–277, 2016.

11 Mohsen Ghaffari, Fabian Kuhn, and Yannic Maus. On the complexity of local distributed graph problems. In *Proceedings of the Symposium on Theory of Computing (STOC)*, pages 784–797, 2017.

12 Mohsen Ghaffari and Hsin-Hao Su. Distributed degree splitting, edge coloring, and orientations. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2505–2523, 2017.

13 Mika Göös, Juho Hirvonen, and Jukka Suomela. Linear-in-delta lower bounds in the local model. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 86–95, 2014.

14 Michał Hańćkowiak, Michał Karonski, and Alessandro Panconesi. On the distributed complexity of computing maximal matchings. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 219–225, 1998.

15 Michał Hańćkowiak, Michał Karonski, and Alessandro Panconesi. On the distributed complexity of computing maximal matchings. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 219–225, 1998.

16 Michał Hańćkowiak, Michał Karoński, and Alessandro Panconesi. A faster distributed algorithm for computing maximal matchings deterministically. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 219–228, 1999.

17 David G. Harris, Johannes Schneider, and Hsin-Hao Su. Distributed ($\Delta+1$)-coloring in sublogarithmic rounds. In *Proceedings of the Symposium on Theory of Computing (STOC)*, pages 465–478, 2016.

18 Juho Hirvonen and Jukka Suomela. Distributed maximal matching: Greedy is optimal. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 165–174, 2012.

19 Amos Israeli and Alon Itai. A fast and simple randomized parallel algorithm for maximal matching. *Information Processing Letters*, 22(2):77–80, 1986.

20 Amos Israeli and Yossi Shiloach. An improved parallel algorithm for maximal matching. *Information Processing Letters*, 22(2):57–60, 1986.

21 Christos Koufogiannakis and Neal E. Young. Distributed fractional packing and maximum weighted b-matching via tail-recursive duality. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, pages 221–238, 2009.

22 Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. The price of being near-sighted. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 980–989, 2006.

23 Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. Local computation: Lower and upper bounds. *J. ACM*, 63(2):17:1–17:44, 2016.

24 Nathan Linial. Distributive graph algorithms - global solutions from local data. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 331–335, 1987.

25 Nathan Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992.

26 Zvi Lotker, Boaz Patt-Shamir, and Seth Pettie. Improved distributed approximate matching. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 129–136, 2008.

27 Zvi Lotker, Boaz Patt-Shamir, and Adi Rosen. Distributed approximate matching. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 167–174, 2007.

28 Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM journal on computing*, 15(4):1036–1053, 1986.

29 Henry Martyn Mulder. Julius petersen's theory of regular graphs. *Discrete mathematics*, 100(1-3):157–175, 1992.

30 Alessandro Panconesi and Romeo Rizzi. Some simple distributed algorithms for sparse networks. *Distributed computing*, 14(2):97–100, 2001.

31 Alessandro Panconesi and Mauro Sozio. Fast primal-dual distributed algorithms for scheduling and matching problems. *Distributed Computing*, 22(4):269–283, 2010.

32 Michal Parnas and Dana Ron. Approximating the minimum vertex cover in sublinear time and a connection to distributed algorithms. *Theoretical Computer Science*, 381(1):183–196, 2007.

33 Ronitt Rubinfeld, Gil Tamir, Shai Vardi, and Ning Xie. Fast local computation algorithms. In *Proceedings of the Symposium on Innovations in Computer Science (ICS)*, pages 223–238, 2011.

34 Jukka Suomela. Distributed algorithms for edge dominating sets. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 365–374, 2010.

35 Mirjam Wattenhofer and Roger Wattenhofer. Distributed weighted matching. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, pages 335–348, 2004.

Sublogarithmic Distributed Algorithms for Lovász Local Lemma, and the Complexity Hierarchy*

Manuela Fischer[1] **and Mohsen Ghaffari**[2]

1 **ETH Zürich, Switzerland**
`manuela.fischer@inf.ethz.ch`
2 **ETH Zürich, Switzerland**
`ghaffari@inf.ethz.ch`

Abstract

Locally Checkable Labeling (LCL) problems include essentially all the classic problems of LOCAL distributed algorithms. In a recent enlightening revelation, Chang and Pettie [FOCS'17] showed that *any* LCL (on bounded degree graphs) that has an $o(\log n)$-round randomized algorithm can be solved in $T_{LLL}(n)$ rounds, which is the randomized complexity of solving (a relaxed variant of) the Lovász Local Lemma (LLL) on bounded degree n-node graphs. Currently, the best known upper bound on $T_{LLL}(n)$ is $O(\log n)$, by Chung, Pettie, and Su [PODC'14], while the best known lower bound is $\Omega(\log \log n)$, by Brandt et al. [STOC'16]. Chang and Pettie conjectured that there should be an $O(\log \log n)$-round algorithm (on bounded degree graphs).

Making the first step of progress towards this conjecture, and providing a significant improvement on the algorithm of Chung et al. [PODC'14], we prove that $T_{LLL}(n) = 2^{O(\sqrt{\log \log n})}$. Thus, any $o(\log n)$-round randomized distributed algorithm for any LCL problem on bounded degree graphs can be automatically sped up to run in $2^{O(\sqrt{\log \log n})}$ rounds.

Using this improvement and a number of other ideas, we also improve the complexity of a number of graph coloring problems (in arbitrary degree graphs) from the $O(\log n)$-round results of Chung, Pettie and Su [PODC'14] to $2^{O(\sqrt{\log \log n})}$. These problems include defective coloring, frugal coloring, and list vertex-coloring.

1998 ACM Subject Classification G.2.2 Graph Algorithms

Keywords and phrases Distributed Graph Algorithms, the Lovász Local Lemma (LLL), Locally Checkable Labeling problems (LCL), Defective Coloring, Frugal Coloring, List Vertex-Coloring

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.18

1 Introduction and Related Work

The Lovász Local Lemma (LLL), introduced by Erdős and Lovász in 1975 [14], is a beautiful result which shows that, for a set of "bad events" in a probability space that have certain sparse dependencies, there is a non-zero probability that none of them happens. This result has become a central tool in the *probabilistic method* [2], when proving that certain combinatorial objects exist. Although the LLL itself does not provide an efficient way for finding these objects, and that remained open for about 15 years, a number of efficient centralized algorithms have been developed for it, starting with Beck's breakthrough in 1991 [7], through [1, 13, 26, 28, 33], and leading to the elegant algorithm of Moser and Tardos in 2010 [29]. See also [9, 19, 21–24] for some of the related work on that track.

* A full version of the paper is available at `https://arxiv.org/abs/1705.04840`.


© Manuela Fischer and Mohsen Ghaffari;
licensed under Creative Commons License CC-BY

In contrast, distributed algorithms for LLL and the related complexity are less well-understood. This question has gained an extraordinary significance recently, due to revelations that show that LLL is a "complete" problem for sublogarithmic-time problems. Next, we first overview the concrete statement of the LLL, and then discuss what is known about its distributed complexity, and what its special significance is for distributed algorithms. Then we proceed to presenting our contributions.

1.1 The LLL and its Special Role in Distributed Algorithms

The Lovász Local Lemma. Consider a finite set $\mathcal{V}$ of independent random variables, and a finite family $\mathcal{X}$ of n (bad) events on these variables. Each event $A \in \mathcal{X}$ depends on some subset $\text{vbl}(A) \subseteq \mathcal{V}$ of variables. Define the dependency graph $G_{\mathcal{X}} = (\mathcal{X}, \{(A, B) \mid \text{vbl}(A) \cap \text{vbl}(B) \neq \emptyset\})$ that connects any two events which share at least one variable. Let d be the maximum degree in this graph, i.e., each event $A \in \mathcal{X}$ shares variables with at most d other events $B \in \mathcal{X}$. Finally, define $p = \max_{A \in \mathcal{X}} \Pr[A]$. The Lovász Local Lemma [14] shows that $\Pr[\cap_{A \in \mathcal{X}} \bar{A}] > 0$, under the *LLL criterion* that $epd \leq 1$. Intuitively, if a local union bound is satisfied around each node in $G_{\mathcal{X}}$, with some slack, then there is a positive probability to avoid all bad events.

What's Known about Distributed LLL? In the standard distributed formulation of LLL, we consider LOCAL-model [25, 32] algorithms that work on the n-node dependency graph $G_{\mathcal{X}}$, where per round each node can send a message to each of its neighbors[1].

Moser and Tardos [29] provided an $O(\log^2 n)$-round randomized distributed algorithm. Chung, Pettie, and Su [12] presented an $O(\log n \cdot \log^2 d)$-round algorithm, which was later improved slightly to $O(\log n \cdot \log d)$ [17]. Perhaps more importantly, under a modestly stronger criterion that $epd^2 < 1$, which is satisfied in most of the standard applications, they gave an $O(\log n)$-round algorithm [12]. This remains the best known distributed algorithm. On the other hand, Brandt et al. [8] showed a lower bound of $\Omega(\log \log n)$ rounds, which holds even if a much stronger LLL criterion of $p2^d < 1$ is satisfied. Even under this exponentially stronger criterion, the best known upper bound changed only slightly to $O(\log n / \log \log n)$ [12].

Completeness of LLL for Sublogarithmic Distributed Algorithms. Chang and Pettie [11] showed that any $o(\log n)$-round randomized algorithm $\mathcal{A}$ for any Locally Checkable Labeling (LCL) problem $\mathcal{P}$ – a problem whose solution can be checked in $O(1)$ rounds [30], which includes all the classic local problems – on bounded degree graphs can be transformed to an algorithm with complexity $O(T_{LLL}(n))$. Here, $T_{LLL}(n)$ denotes the randomized complexity for solving LLL on n-node bounded-degree graphs, with high probability.

In a nutshell, their idea is to "lie" to the algorithm $\mathcal{A}$ and say that the network size is some much smaller value $n^* \ll n$. This deceived algorithm $\mathcal{A}$ may have a substantial probability to fail, creating an output that violates the requirements of the LCL problem $\mathcal{P}$ somewhere. However, the probability of failure in each local neighborhood is at most $1/n^*$. Choosing n^* a large enough constant, depending on the complexity of $\mathcal{A}$, the algorithm $\mathcal{A}$ provides an LLL system – where we have one bad event for violation of each local requirement of $\mathcal{P}$ – that satisfies the criterion $pd^c < 1$ for a desirably large constant c. Hence, we can solve this LLL system and thus obtain a solution for the original LCL problem $\mathcal{P}$ in $O(T_{LLL}(n))$ time.

[1] One can imagine a few alternative graph formulations, all of which turn out to be essentially equivalent in the LOCAL model, up to an $O(1)$ overhead in complexity.

This result implies that LLL is important not only for a few special problems, but in fact for essentially *all* sublogarithmic-time distributed problems. Due to this remarkable role, Chang and Pettie state that "*understanding the distributed complexity of the LLL is a significant open problem.*" Furthermore, although a wide gap between the best upper bound $O(\log n)$ [12] and lower bound $\Omega(\log \log n)$ [8] persists, they conjecture the latter to be tight:

▶ **Conjecture** (Chang, Pettie [11]). *There exists a sufficiently large constant c such that the distributed LLL problem can be solved in $O(\log \log n)$ time on bounded degree graphs, under the symmetric LLL criterion $pd^c < 1$.*[2]

1.2 Our Contributions

Faster Distributed LLL. We make a significant step of progress towards this conjecture:

▶ **Theorem 1.** *There is a $2^{O(\sqrt{\log \log n})}$-round randomized distributed algorithm that, with high probability*[3]*, solves the LLL problem with degree at most $d = O(\log^{1/5} \log n)$, under a symmetric polynomial LLL criterion $p(ed)^{32} < 1$.*[4]

This improves over the $O(\log n)$-round algorithm of Chung et al. [12]. We note that even under a significantly stronger exponential LLL criterion – formally requiring $4ep2^d d^4 < 1$ – the best known round complexity was $O(\log n / \log \log n)$ [12]. Furthermore, we note that a key ingredient in developing Theorem 1 is a deterministic distributed algorithm for LLL, which we present in Theorem 11. To the best of our knowledge, this is the first (non-trivial) deterministic distributed LLL algorithm. In fact, we believe that any conceivable future improvements on Theorem 1 may have to improve on this deterministic part.

Moreover, our method provides some further supporting evidence for the conjecture of Chang and Pettie. In particular, if one finds a $\operatorname{poly} \log n$-round deterministic algorithm for $\big(O(\log n), O(\log n)\big)$ network decomposition [31] – a central problem that has remained open for a quarter century, but is often perceived as likely to be true – then, combining that with our method would prove $T_{LLL}(n) = \operatorname{poly}(\log \log n)$.

A Gap in the Randomized Distributed Complexity Hierarchy. Putting Theorem 1 with [11, Theorem 6], we get the following automatic speedup result:

▶ **Corollary 2.** *Let $\mathcal{A}$ be a randomized* LOCAL *algorithm that solves some LCL problem $\mathcal{P}$ on bounded degree graphs, w.h.p., in $o(\log n)$ rounds. Then, it is possible to transform $\mathcal{A}$ into a new randomized* LOCAL *algorithm $\mathcal{A}'$ that solves $\mathcal{P}$, w.h.p., in $2^{O(\sqrt{\log \log n})}$ rounds.*

Using a similar method, and our deterministic LLL algorithm (Theorem 11), we obtain the following corollary, the proof of which is deferred to the full version [15]. This corollary shows that any $o(\log \log n)$-round randomized algorithm for an LCL problem on bounded degree graphs can be improved to a deterministic $O(\log^* n)$-round algorithm. This result seems to be implicit in the recent work of Chang, Kopelowitz, and Pettie [10], though with a quite different proof, and it can be derived from [10, Corollary 3] and [10, Theorem 3].

---

[2] This statement, as is, has a small imprecision: one should assume either that $d \geq 2$, in which case $pd^c < 1$ can be replaced with $p(ed)^{c'} < 1$ for some other constant c', or that $pd^c < 1/2$. Otherwise, two events of head or tail for a fair coin have $p = 1/2$ and $d = 1$, thus $pd^c < 1$, but one cannot avoid both.

[3] As standard, the phrase *with high probability* (w.h.p.) indicates that an event has probability at least $1 - n^{-c}$, for a sufficiently large constant c.

[4] We remark that we did not try to optimize the constants.

▶ **Corollary 3.** *Let $\mathcal{A}$ be a randomized* LOCAL *algorithm that solves some LCL problem $\mathcal{P}$ on bounded degree graphs, w.h.p., in $o(\log \log n)$ rounds. Then, it is possible to transform $\mathcal{A}$ into a new deterministic* LOCAL *algorithm $\mathcal{A}'$ that solves $\mathcal{P}$ in $O(\log^* n)$ rounds.*

Faster Distributed Algorithms for Graph Colorings via LLL. For some distributed graph problems on bounded degree graphs, we can immediately get faster algorithms by applying Theorem 1. However, there are two quantifiers which appear to limit the applicability of Theorem 1: (L1) it requires a stronger form of the LLL criterion, concretely needing $p(ed)^{32} < 1$ instead of $epd \leq 1$; (L2) it applies mainly to bounded degree graphs.

We explain how to overcome these two limitations in most of the LLL-based problems studied by Chung, Pettie, and Su [12]. Regarding limitation (L1), we show that even though in many coloring problems the direct LLL formulation would not satisfy the polynomial criterion $p(ed)^{32} < 1$, we can still solve the problem, through a number of iterations of partial colorings, each satisfying this stronger LLL criterion. Regarding limitation (L2), we explain how in many problems, the first step of our LLL algorithm, which is its only part that relies on bounded degrees, can be replaced by a faster randomized step for that coloring.

The end results of our method include algorithms with round complexity $2^{O(\sqrt{\log \log n})}$ for a number of coloring problems, improving on the corresponding $O(\log n)$-round algorithms of Chung, Pettie, and Su [12]: defective coloring, frugal coloring, and list vertex-coloring. The first two are presented respectively, in Section 4, Section 5. The third coloring result, as well as some of the proofs, are deferred to the full version of this article [15].

2 Preliminaries

2.1 Network Decompositions

Roughly speaking, a *network decomposition* [4, 31] partitions the nodes into a few blocks, each of which is made of a number of low-diameter connected components. More formally, the definition is as follows:

▶ **Definition 4** (Network Decomposition)**.** Given a graph $G = (V, E)$, a partition of the nodes V into C vertex-disjoint blocks $V_1, V_2, \ldots, V_C$ is a (C, D) network decomposition if in each block's induced subgraph $G[V_i]$ each connected component has diameter at most D.

▶ **Lemma 5** (The Network Decomposition Algorithm)**.** *Given an n-node network $G = (V, E)$, there is a deterministic distributed algorithm that computes a $(\lambda, n^{1/\lambda} \cdot \log n)$ network decomposition of G in $\lambda \cdot n^{1/\lambda} \cdot 2^{O(\sqrt{\log n})}$ rounds.*

The proof of Lemma 5 is deferred to the full version; it works mainly by putting together some ideas of Awerbuch and Peleg [5], Panconesi and Srinivasan [31], and Awerbuch et al. [3]. However, we are not aware of this result appearing in prior work.

2.2 Shattering

In a number of our algorithms, we make use of the following lemma which, roughly speaking, shows that if each node of the graph remains with some small probability and we have certain independence between these events, the remaining connected components are "small". We remark that this lemma or its variants are key ingredients in Beck's LLL method [7], sometimes referred to as the *shattering lemma*, and analogues of it appear in the literature [1, 6, 17, 18, 20, 26, 27]. The proof of this lemma is deferred to the full version.

▶ **Lemma 6** (The Shattering Lemma). *Let $G = (V, E)$ be a graph with maximum degree Δ. Consider a process which generates a random subset $B \subseteq V$ where $P(v \in B) \leq \Delta^{-c_1}$, for some constant $c_1 \geq 1$, and that the random variables $1(v \in B)$ depend only on the randomness of nodes within at most c_2 hops from v, for all $v \in V$, for some constant $c_2 \geq 1$. Moreover, let $H = G^{[2c_2+1, 4c_2+2]}$ be the graph which contains an edge between u and v iff their distance in G is between $2c_2 + 1$ and $4c_2 + 2$. Then with probability at least $1 - n^{-c_3}$, for any constant $c_3 < c_1 - 4c_2 - 2$, we have the following three properties:*

(P1) *$H[B]$ has no connected component U with $|U| \geq \log_\Delta n$.*

(P2) *$G[B]$ has size at most $O\big(\log_\Delta n \cdot \Delta^{2c_2}\big)$.*

(P3) *Each connected component of $G[B]$ admits a $(\lambda, O(\log^{1/\lambda} n \cdot \log^2 \log n))$ network decomposition, for any integer $\lambda \geq 1$, which can be computed in $\lambda \cdot \log^{1/\lambda} n \cdot 2^{O(\sqrt{\log \log n})}$ rounds, deterministically.*

3 Our General Algorithm for Lovász Local Lemma

In this section, we explain our sublogarithmic-time LLL algorithm of Theorem 1, which solves LLL in $2^{O(\sqrt{\log \log n})}$ rounds on bounded degree graphs, given the condition that $p(ed)^{32} < 1$.

The Lovász Local Lemma Setting. We consider a finite set $\mathcal{V}$ of independent random variables, and a finite family $\mathcal{X}$ of n bad events on these variables. Each event $A \in \mathcal{X}$ depends on some subset $\text{vbl}(A) \subseteq \mathcal{V}$ of variables. In fact, essentially without loss of generality, we assume that each random variable in $\mathcal{V}$ is a fair random bit. We note that in practically all settings of interest, we can rewrite the basic random variables as a function of at most $\text{poly}(n)$ independent random bits, hence transitioning from arbitrary set of random variables to another space of random variables with just fair random bits. The number of random bits will impact only the local computations and as such, since we are working in the local model which does not assume a limited computational power, we can allow the number of random bits to be arbitrarily large. The distributed algorithms that we describe work on the dependency graph of the events, defined as $G_{\mathcal{X}} = (\mathcal{X}, \{(A, B) \mid \text{vbl}(A) \cap \text{vbl}(B) \neq \emptyset\})$. That is, this graph has one vertex for each event and that connects any two events which share at least one variable. Then, d denotes the maximum degree in this graph, and $p = \max_{A \in \mathcal{X}} \Pr[A]$.

Our General LLL Algorithm. The algorithm is developed in two stages, as we overview next. In the first stage, presented in Section 3.1, we explain a randomized algorithm with complexity $\lambda \cdot \log^{1/\lambda} n \cdot 2^{O(\sqrt{\log \log n})}$, given that an LLL criterion $p(ed)^{4\lambda} < 1$ is satisfied. In the main regime of interest, the best LLL criterion exponent that we will assume is $\lambda = O(1)$, and thus this $(\lambda \cdot \log^{1/\lambda} n \cdot 2^{O(\sqrt{\log \log n})})$-round algorithm, on its own, would not get us to our target complexity of $2^{O(\sqrt{\log \log n})}$, although still being an improvement on the $O(\log n)$-round algorithm of [12].

In the second stage, presented in Section 3.2, we improve this complexity to $2^{O(\sqrt{\log \log n})}$. That improvement works mainly by viewing the sublogarithmic-time local algorithm of Section 3.1 as setting up a new LLL, with a much larger exponent λ in its LLL criterion, hence allowing us get to a much smaller complexity by (recursively) applying the same scheme. This speed up is inspired by the ideas of Chung and Pettie [11] which showed that LLL can be used to speed up sublogarithmic-time local algorithms.[5]

[5] Though, we find this recursive application of the idea to speed up the complexity of LLL itself, through

3.1 The Base LLL Algorithm

▶ **Theorem 7.** *For any integer $\lambda \geq 8$, there is a randomized distributed algorithm solving the LLL problem under the symmetric criterion $p(ed)^{4\lambda} < 1$, in $O(d^2) + \lambda \cdot \log^{1/\lambda} n \cdot 2^{O(\sqrt{\log \log n})}$ rounds, with high probability.*

This algorithm consists of two parts: (1) a randomized algorithm, explained in Section 3.1.1, which performs some partial sampling in the LLL space, thus setting some of the variables, in a manner that *shatters* the graph, hence leaving small connected components among the unset variables; (2) a deterministic LLL algorithm, explained in Section 3.1.2, which we use to solve the remaining small connected components. To the best of our knowledge, this is the first non-trival deterministic distributed LLL algorithm. In Section 3.1.3, we combine these two parts, concluding the proof of Theorem 7.

This general *shattering* style for randomized algorithms – which first performs some randomized steps to break the graph into small remaining connected components, and then uses deterministically solves these remaining components – is rooted in the breakthrough LLL algorithm of Beck [7], and has been used extensively before [1, 6, 17, 18, 20, 26, 27].

3.1.1 The Randomized Part

We now explain the randomized component of our LLL algorithm for bounded degree graphs, which performs a partial sampling in the LLL space, thus setting some of the variables, in a manner that guarantees the following two properties: (1) the conditional probabilities of the bad events, conditioned on the already set variables, satisfy a polynomial LLL criterion, (2) the connected components of the events on variables that remain unset are "small" (e.g., for bounded degree graphs, they have size at most $O(\log n)$), with high probability.

These two properties together will allow us to invoke the deterministic LLL algorithm that we present later in Section 3.1.2 on the remaining components of variables that remain unset. In particular, (1) means that the bad events $\mathcal{X}$ form another LLL problem on the variables that remain unset, where each new bad event has probability at most $\sqrt{p}$. Furthermore, (2) ensures that the components are small enough to make the deterministic algorithm efficient.

Our partial sampling is inspired by a sequential LLL algorithm of Molloy and Reed [26].

▶ **Lemma 8** (Random Partial Setting for the LLL Variables)**.** *There is a randomized distributed algorithm that computes, w.h.p., in $O(d^2 + \log^* n)$ rounds, a partial assignment of values to variables – setting the values of the variables in a set $\mathcal{V}^* \subseteq \mathcal{V}$, hence leaving the variables in $\mathcal{V}' := \mathcal{V} \setminus \mathcal{V}^*$ unset – of an LLL satisfying $p(ed)^{4\lambda} < 1$, for any integer $\lambda \geq 8$, such that*

- **(i)** *$Pr[A \mid \mathcal{V}^*] \leq \sqrt{p}$ for all $A \in \mathcal{X}$, and*
- **(ii)** *w.h.p. each connected component of $G^2_{\mathcal{X}}[\mathcal{V}']$ admits a $(\lambda, O(\log^{1/\lambda} n \cdot \log^2 \log n))$ network decomposition, which can be computed in $\lambda \cdot \log^{1/\lambda} n \cdot 2^{O(\sqrt{\log \log n})}$ rounds, deterministically.*

Proof. We first compute a (d^2+1)-coloring of the square graph $G^2_{\mathcal{X}}$ on the events, which can be done even deterministically in $\tilde{O}(d) + O(\log^* n)$ rounds [16]. Suppose $\mathcal{X}_i$ is the set of events colored with color i, for $i \in \{1, \ldots, d^2+1\}$. We process the color classes one by one.

During the process, some variables may get frozen, as we discuss soon. The process assigns values for all non-frozen variables, as follows: For each color $i \in \{1, \ldots, d^2+1\}$, and for each node $A \in \mathcal{X}_i$ in parallel, we make node A sample values for its (so far non-frozen)

strengthening the corresponding LLL criterion, somewhat amusing.

variables locally, one by one. Notice that since we are using a coloring of $G_{\mathcal{X}}^2$, for each color i, each event $B \in \mathcal{X}$ shares variables with at most one event $A \in \mathcal{X}_i$. Hence, during this iteration, at most one node A is sampling variables of event B. Consider a node A that is choosing values for some random variables. Each time, when A is choosing a value for a variable $v \in \text{vbl}(A)$, it checks whether this setting makes one of the events $B \in \mathcal{X}$ involving variable v *dangerous*. We call an event B *dangerous* if $\Pr[B|\mathcal{V}_B^*] \geq \sqrt{p}/2$, where $\mathcal{V}_B^*$ denotes the already set variables of B up to this point in the sampling process. If the recently set variable v leads to a dangerous event B, then node A *freezes* variable v as well as all the remaining variables of event B. We do not assign values to frozen variables in the remainder of the randomized sampling process. We have two key observations regarding this process:

▶ **Observation 9.** *At the end of each iteration, for each event $A \in \mathcal{X}$, the conditional probability of event A, conditioned on the already made assignments $\mathcal{V}_A^*$, is at most $\sqrt{p}$.*

Proof Sketch. The first time that an event A becomes dangerous, all of its remaining variables get frozen and no other assignment gets made for its variables. By definition, before A becoming dangerous, the conditional probability of event A, conditioned on the already made assignments $\mathcal{V}_A^*$, is at most $\sqrt{p}/2$. If A becomes dangerous, that's because of setting of one last random variable. Since we have assumed that the random variables are all fair random bits, at the time of setting one last bit, the conditional probability of event A can increase by at most a 2 factor. Hence, once A becomes dangerous and all of its remaining variables get frozen, its conditional probability is at most $\sqrt{p}$. ◀

▶ **Observation 10.** *For each event $A \in \mathcal{X}$, the probability of A having at least one unset variable is at most $2(d+1)\sqrt{p}$. Furthermore, this is independent of events that are further than 2 hops from A.*

Proof Sketch. We first claim that for each $B \in \mathcal{X}$, the probability that B ever becomes dangerous is at most $2\sqrt{p}$. This is because otherwise the total probability of B happening would exceed p. Notice that during the process, some variables get a value assigned to them, and some variables get frozen, because of B or some other adjacent event becoming dangerous. More concretely, to focus on just one event B, let us consider two processes for revealing the samples values for variables of B. In the first process, we sample all the variables in one shot. Clearly, in this process, $Pr[B] \leq p$. The second process has two phases: in the first phase, we examine the variables of B sequentially, one by one, and each time sample a value for each variable, with one exception: at each time, an adversary might call some variables that have not been revealed so-far "frozen"and moves them to the second phase; any such frozen variable will not be sampled in the first phase. This adversary can take into account all the possibilities of neighboring LLL-events making a random variable of B become frozen. Now in this process, if the conditional probability of event B given the already assigned variables exceeds $\sqrt{p}/2$, or if we run out of non-frozen variables, the first phase ends. In the second phase, we simultaneously sample all the variables that have been moved to the second phase. Now, notice that the probability of event B happening is exactly the same in the two processes; the second process is just a different order of revealing the sampled values of the first process. Hence, also in the second process, we have $Pr[B] \leq p$. Now, in the second process, let B' be the event that the conditional probability of event B at the end of the first phase (given its assignments variables) exceeds $\sqrt{p}/2$. By definition, $Pr[B'|B] \geq \sqrt{p}/2$. Since $Pr[B] \leq p$, we have $Pr[B'] \leq \frac{p}{\sqrt{p}/2} = 2\sqrt{p}$. Thus, in our LLL sampling process, the probability that each event $B \in \mathcal{X}$ ever becomes dangerous is at most $2\sqrt{p}$.

Now, an event $A \in \mathcal{X}$ can have frozen variables only if at least one of its neighboring events B, or event A itself, becomes dangerous. Since A has at most d neighboring events, by a union bound, the latter has probability at most $2(d+1)\sqrt{p}$. ◀

Observation 9 implies property (i) of Lemma 8. We use Observation 10 to conclude that the events with at least one unset variable comprise "small" connected components. In particular, we apply Lemma 6 to $G^2_{\mathcal{X}}$ with the random partial setting process generating a set $B \subseteq \mathcal{X}$ of the events that have at least one variable unset. By Observation 10, each event remains with probability at most $2(d+1)\sqrt{p} \leq 2(d+1) \cdot e^{-2\lambda} \cdot d^{-2\lambda} \leq d^{-15}$. These events depend only on events within at most 1 hop in $G^2_{\mathcal{X}}$ and hence 2 hops in $G_{\mathcal{X}}$. Thus, Lemma 6 (P3) shows that with probability at least $1 - n^{-3}$ property (ii) holds. ◀

3.1.2 The Deterministic Part

▶ **Theorem 11.** *For any integer $\lambda \geq 1$, the distributed LLL problem can be solved deterministically in $\lambda \cdot n^{1/\lambda} \cdot 2^{O(\sqrt{\log n})}$ rounds, under the symmetric LLL criterion $p(ed)^\lambda < 1$. If the algorithm is provided a (λ, γ) network decomposition of the square graph $G^2_{\mathcal{X}}$, then the LLL algorithm runs in just $O(\lambda \cdot (\gamma + 1))$ rounds.*

We make a black-box invocation to the distributed algorithm stated in Lemma 5 for computing a $(\lambda, n^{1/\lambda} \cdot \log n)$ network decomposition, and then solve the LLL problem on top of this decomposition, by going through its blocks one by one.

The running time of our deterministic LLL algorithm hence directly depends on the network decomposition it works with. In particular, if there is a $\operatorname{poly}\log n$-round deterministic distributed algorithm that computes a $(\operatorname{poly}\log n, \operatorname{poly}\log n)$ network decomposition, then this algorithm solves any LLL problem satisfying the criterion $p(ed)^\lambda < 1$ (with $\lambda = \operatorname{poly}\log n$) deterministically in $\operatorname{poly}\log n$ rounds. This would then directly improve the running time of the randomized LLL algorithm of Theorem 7 to $\operatorname{poly}(\log\log n)$, proving that $T_{LLL}(n) = \operatorname{poly}(\log\log n)$, thus almost confirming the conjecture of Chang and Pettie [11].

In fact, we believe that a conceivable future improvement of our LLL algorithm may need to improve this deterministic component, ideally to complexity $O(\log n)$ for proving the $T_{LLL}(n) = O(\log\log n)$ conjecture of Chang and Pettie [11].

Proof of Theorem 11. We first compute a $(\lambda, n^{1/\lambda} \cdot \log n)$ network decomposition of $G^2_{\mathcal{X}}$, which decomposes its nodes into λ disjoint blocks $\mathcal{X}_1, \ldots, \mathcal{X}_\lambda$, such that each connected component of $G^2_{\mathcal{X}}[\mathcal{X}_i]$ has diameter at most $n^{1/\lambda} \cdot \log n$. This decomposition can be computed in $\lambda \cdot n^{1/\lambda} \cdot 2^{O(\sqrt{\log n})}$ rounds, using Lemma 5.

Then, iteratively for $i = 1, \ldots, \lambda$, we assign values to all variables of events in $\mathcal{X}_i$ that have remained unset. The values are chosen is such a way that, after i steps, the conditional probability of *any* event in $\mathcal{X}$, conditioned on all the assignments in variables of events in $\bigcup_{j=1}^{i} \mathcal{X}_j$, is at most $p(ed)^i < 1$. Once $i = \lambda$, since the conditional failure probability is $p(ed)^\lambda < 1$ but all the variables are already assigned, we know that none of the events occurs.

The base case $i = 0$ is trivial. In the following, we explain how to set the values for variables involved in events of $\mathcal{X}_i$ in $n^{1/\lambda} \cdot \log n$ rounds. Let $\mathcal{V}_i$ be the set of variables in events of $\mathcal{X}_i$ that remain with no assigned value. We form a new LLL problem, as follows: For each bad event $A \in \mathcal{X}$, we introduce an event $B_{A,i}$ on the space of values of $\mathcal{V}_i$. This is the event that the values of $\mathcal{V}_i$ get chosen such that the conditional probability of the event A, conditioned on the variables in $\bigcup_{j=1}^{i} \mathcal{V}_j$, is larger than $p(ed)^i$. Notice that $Pr[B_{A,i} \mid \bigcup_{j=1}^{i-1} \mathcal{V}_j] \leq \frac{p(ed)^{i-1}}{p(ed)^i} = \frac{1}{ed}$. This is because, the variables $\bigcup_{j=1}^{i-2} \mathcal{V}_j$ are set such that

the conditional probability of $B_{A,i}$ given these set values is at most $p(ed)^{i-1}$, and thus, the probability that the values of $\mathcal{V}_{i-1}$ get chosen that the conditional probability given the set values in $\bigcup_{j=1}^{i-1}\mathcal{V}_j$ exceeds $p(ed)^i$ is at most $\frac{p(ed)^{i-1}}{p(ed)^i} = \frac{1}{ed}$. Moreover, each event $B_{A,i}$ depends on at most d other events $B_{A',i}$. Hence, the family of events $B_{A,i}$ on the variable set $\mathcal{V}_i$ satisfies the conditions of the tight (symmetric) LLL. Therefore, by the Lovász local lemma, we know that there exists an assignment to variables of $\mathcal{V}_i$ which makes no event $B_{A,i}$ happen. That is, an assignment such that the conditional probability of each event A, conditioned on the assignments in $\bigcup_{j=1}^{i}\mathcal{V}_j$, is bounded by at most $p(ed)^i$.

Given the existence, we find such an assignment in $n^{1/\lambda}\cdot\log n$ rounds, as follows: each component of $G^2_{\mathcal{X}}[\mathcal{X}_i]$ first gathers the whole topology of this component (as well as its incident events and the current assignments to any of their variables), in $O(n^{1/\lambda}\log n)$ rounds. Then, it decides about an assignment for its own variables in $\mathcal{V}_i$, by locally brute-forcing all possibilities. Different components can decide independently as there is no event that shares variables with two of them, since they are non-adjacent in $G^2_{\mathcal{X}}$. ◀

3.1.3 Wrap-Up: Base LLL Algorithm

Proof of Theorem 7. We run the randomized algorithm of Lemma 8 for computing a partial setting of the variables, in $O(d^2+\log^* n)$ rounds. Then, by Lemma 8 (i), the remaining events $\mathcal{X}'$ (those which have at least one unset variable) form a new LLL system on the unset variables, where each bad event has probability at most $\sqrt{p}$.

Moreover, by Lemma 8 (ii), each connected component of the square graph $G^2_{\mathcal{X}}[\mathcal{X}']$ of these remaining events $\mathcal{X}'$ has a $(\lambda, O(\log^{1/\lambda} n\cdot\log^2\log n))$ network decomposition, which we can compute in $\lambda\cdot\log^{1/\lambda} n\cdot 2^{O(\sqrt{\log\log n})}$ rounds, deterministically. From now on, we handle the remaining events in different connected components of $G^2_{\mathcal{X}}[\mathcal{X}']$ independently.

Since $\sqrt{p}(ed)^\lambda < 1$, we can now invoke the deterministic LLL algorithm of Theorem 11 on top of the network decomposition of each component. Our deterministic LLL then runs in $\lambda\cdot\log^{1/\lambda} n\cdot\log^2\log n$ additional rounds, and finds assignments for these remaining variables, without any of the events occurring, hence solving the overall LLL problem. The overall round complexity is $O(d^2)+\lambda\cdot\log^{1/\lambda} n\cdot 2^{O(\sqrt{\log\log n})}$. ◀

3.2 Improving the Base LLL Algorithm via Bootstrapping

Proof of Theorem 1. In Theorem 7, we saw an algorithm $\mathcal{A}$ that solves any n-event LLL under the criterion $p(ed)^{32}<1$ in $T_{n,d}=O(d^2+\log^{1/4} n)$ rounds. We now explain how to bootstrap this algorithm to run in $2^{O(\sqrt{\log\log n})}$ rounds, on bounded degree graphs.

Inspired by the idea of Chang and Pettie [11], we will lie to $\mathcal{A}$ and say that the LLL graph has $n^* \ll n$ nodes, for a value of n^* to be fixed later. Then, $\mathcal{A}_{n^*}$ runs in $T_{n^*,d} = O(d^2+\log^{1/4} n^*)$ rounds. In this algorithm, the probability of any local failure (i.e., a bad event of LLL happening) is at most $1/n^*$. We can view this as a new system of bad events which satisfies a much stronger LLL criterion. In particular, we consider each of the previous bad LLL events as a bad event of the new LLL system, on the space of the random values used by $\mathcal{A}_{n^*}$, but now we connect two bad events if their distance is at most $2T_{n^*,d}+1$. Notice that if two events are not connected in this new LLL, then in algorithm $A_{n^*,d}$, they depend on disjoint sets of random variables and thus they are independent.

The degree of the new LLL system is $d' = d^{2T_{n^*,d}+1} = d^{O(d^2+\log^{1/4} n^*)}$. On the other hand, the probability of the bad events of the new system is at most $p' = 1/n^*$. Hence, the polynomial LLL criterion is satisfied with exponent $\lambda' = \frac{\log_d n^*}{O(d^2+\log^{1/4} n^*)}$. We choose

$n^* = \log n$, which, for $d = O((\log\log n)^{1/5})$, means $\lambda' = \Omega(\sqrt{\log\log n})$. Hence, this new LLL system can be solved using the LLL algorithm of Theorem 7 in time

$$(d')^2 + \lambda' \cdot \log^{1/\lambda'} n \cdot 2^{O(\sqrt{\log\log n})} =$$
$$d^{O(d^2+(\log\log n)^{1/4})} + \sqrt{\log\log n} \cdot (\log n)^{1/\Omega(\sqrt{\log\log n})} \cdot 2^{O(\sqrt{\log\log n})} = 2^{O(\sqrt{\log\log n})}.$$

We should note that these are rounds on the new LLL system, but each of them can be performed in $2T_{n^*,d} + 1 = O(d^2 + \log^{1/4} n^*) = O(\sqrt{\log\log n})$ rounds on the original graph. Hence, the overall complexity is still $2^{O(\sqrt{\log\log n})}$. ◀

We next state another result obtained via this speedup method, targeting higher degree graphs, which we will use in our coloring algorithms. The proof is deferred to the full version.

▶ **Lemma 12.** *Let $\mathcal{A}$ be a randomized* LOCAL *algorithm that solves some LCL problem $\mathcal{P}$ on n-node graphs with maximum degree $d \leq 2^{O(\log^{1/4}\log n)}$ in $O(\log^{1/4} n)$ rounds. Then, it is possible to transform $\mathcal{A}$ into a new randomized* LOCAL *algorithm $\mathcal{A}'$ that solves $\mathcal{P}$, w.h.p., in $2^{O(\sqrt{\log\log n})}$ rounds.*

4 Defective Coloring

An f-defective coloring is a (not necessarily proper) coloring of nodes, where each node has at most f neighbors with the same color. In other words, in an f-defective coloring, each color class induces a subgraph with maximum degree f. Chung, Pettie, and Su [12] gave an $O(\log n)$-round distributed algorithm for computing an f-defective coloring with $O(\Delta/f)$ colors. We here improve this complexity to $2^{O\left(\sqrt{\log\log n}\right)}$ rounds.

▶ **Theorem 13.** *There is a $2^{O\left(\sqrt{\log\log n}\right)}$-round randomized distributed algorithm that computes an f-defective $O(\Delta/f)$-coloring in an n-node graph with maximum degree Δ, w.h.p., for any integer $f \geq 0$.*

Direct LLL Formulation of Defective Coloring. Chung, Pettie, and Su [12] give a formulation of f-defective $\lceil 2\Delta/f \rceil$-coloring as LLL as follows. Each node picks a color uniformly at random. For each node v, there is a bad event D_v that v has more than f neighbors assigned the same color as v. The probability of a neighbor u having the same color as v is $f/(2\Delta)$. Hence, the expected number of neighbors of v with the same color as v is at most $f/2$. By a Chernoff bound, the probability of v having more than f neighbors with the same color is at most $e^{-f/6}$. Moreover, the dependency degree between the bad events D_v is $d \leq \Delta^2$. Therefore, $p(ed)^{32} \leq e^{-f/6+32+64\log\Delta} < 1$ for $f = \Omega(\log\Delta)$.

We are unable to directly apply our LLL algorithm of Theorem 1 to this formulation, because: (A) For $f = o(\log\Delta)$, this LLL formulation does not satisfy the polynomial criterion $p(ed)^{32} < 1$, (B) even if this criterion is satisfied, the dependency degree d may be larger than what Theorem 1 can handle.

Iterative LLL Formulation of Defective Coloring via Bucketing. Instead of directly finding an f-defective $O(\Delta/f)$-coloring with one LLL problem – i.e., a partition of G into $O(\Delta/f)$ buckets with maximum degree f each – we gradually approach this goal by iteratively partitioning the graph into buckets, until they have maximum degree f. In other words, we slow down the process of partitioning. We gradually decrease the degree, moving from maximum degree x to $\log^5 x$ in one iteration. We can see each of these bucketing steps – that

is, the partitioning into subgraphs – as a partial coloring, which fixes some bits of the final color. Each of these slower partitioning steps can be formulated as an LLL. The function $x \mapsto \log^5 x$ is chosen large enough for the corresponding LLL to satisfy the polynomial criterion, and small enough so that decreasing the degree from Δ to f does not take too many iterations, namely $O(\log^* \Delta)$ iterations only.

We now explain how a defective coloring problem can be solved using iterated bucketing. We first formulate the bucketing as an LLL problem satisfying the polynomial LLL criterion, and present ways for solving this LLL for different ranges of Δ. Then, we explain how iterated application of solving these bucketing LLLs leads to a partition of the graph into $O(\Delta/f)$ many degree-f buckets.

One Iteration of Bucketing. In one bucketing step, we would like to partition our graph with degree Δ into roughly Δ/Δ' buckets, each with maximum degree Δ', for a $\Delta' = \Omega(\log^5 \Delta)$. Notice that we can achieve the defective coloring of Theorem 13, by repeating this bucketing procedure, iteratively. See the proof of Theorem 13, which appears in the full version, for details of iterative bucketing. Each iteration of bucketing can be formulated as an LLL as follows.

LLL Formulation of Bucketing. Let $k = (1+\varepsilon)\Delta/\Delta'$ for $\varepsilon = \log^2 \Delta/\sqrt{\Delta'}$. We consider the random variables assigning each node a bucket number in $[k]$. Then, we introduce a bad event D_v for node v if more than Δ' neighbors of v are assigned the same number as v. In expectation, the number of neighbors of a node in the same bucket is at most $\Delta'/(1+\varepsilon)$. By a Chernoff bound, the probability of having more than Δ' neighbors in the same bucket is at most $p = e^{-\Omega(\varepsilon^2\Delta')} = e^{-\Omega(\log^4 \Delta)}$. Moreover, the dependency degree between these bad events is $d \leq \Delta^2$. Hence, this LLL satisfies the polynomial criterion.

If $\Delta \leq O(\log^{1/10} \log n)$, then $d = O(\log^{1/5} \log n)$, and thus we can directly apply the LLL algorithm of Theorem 1 to compute such a bucketing in $2^{O(\sqrt{\log\log n})}$ rounds. For larger values of Δ, however, we cannot apply Theorem 1. The following lemma discusses how we handle this range by sacrificing a 2-factor in the number of buckets. In a nutshell, the idea is to just perform one sampling step of bucketing, and then to deal with nodes with too large degree separately, by setting up another bucketing LLL. While the first LLL on the whole graph could not be solved directly, the second LLL is formulated only for a "small" subset of nodes, which allows an efficient solution. Because of the two trials of solving an LLL, we lose a 2-factor in the total number of buckets.

▶ **Lemma 14.** *For $\Delta \geq \Omega(\log^{1/10} \log n)$, there is a $2^{O(\sqrt{\log\log n})}$-round randomized distributed algorithm that computes a bucketing into $2k$ buckets with maximum degree Δ' each, for $\Delta' = \Omega(\log^5 \Delta)$, $\varepsilon = \log\Delta/\sqrt{\Delta'}$, and $k = (1+\varepsilon)\Delta/\Delta'$, with high probability.*

5 Frugal coloring

A *β-frugal coloring* is a proper coloring in which no color appears more than β times in the neighborhood of any node. We improve the complexity of β-frugal $O(\Delta^{1+1/\beta})$-coloring from $O(\log n)$ by Chung, Pettie, and Su [12] to $2^{O(\sqrt{\log\log n})}$.

▶ **Theorem 15.** *There is a $2^{O(\sqrt{\log\log n})}$-round randomized distributed algorithm that computes a β-frugal $(120 \cdot \Delta^{1+1/\beta})$-coloring[6] in a n-node graph with maximum degree Δ, w.h.p., for any integer $\beta \geq 1$.*

[6] We remark that we have not tried to optimize this constant 120.

Direct LLL Formulation of Frugal Coloring. Molloy and Reed [27, Theorem 19.3] formulated frugal coloring as an LLL problem in the following straight-forward way: Each node picks a color uniformly at random. There are two types of bad events: On the one hand, we have the *properness* condition, i.e., a bad event $M_{u,v}$, for each $\{u, v\} \in E$, which happens if u and v have the same color. On the other hand, the *frugality* condition – requiring that no node has more than β neighbors of the same color. That is, we have one bad event $F_{u_1,\dots,u_{\beta+1}}$ for each set $u_1, \dots, u_{\beta+1} \in N(v)$ of nodes in the neighborhood of some node v, which happens if all these nodes $u_1, \dots, u_{\beta+1}$ are assigned the same color. For palettes of size C, the probability of a bad event is at most $1/C$ for type 1 and at most $1/C^\beta$ for type 2. Each event depends on at most $(\beta+1)\Delta$ type 1 and at most $(\beta+1)\Delta\binom{\Delta}{\beta}$ type 2 events.

Iterated LLL Formulation of Frugal Coloring via Partial Frugal Coloring. While the above formulation is enough to satisfy the asymmetric tight LLL criterion for $C = O(\Delta^{1+1/\beta})$, it does not satisfy the (symmetric) polynomial LLL. Therefore, the algorithm of Theorem 1 is not directly applicable. We show how to break down the frugal coloring problem into a sequence of few partial coloring problems, coloring only some of the nodes that have remained uncolored, each of them satisfying the polynomial LLL criterion.

Roadmap. In Section 5.1, we formalize our notion of partial frugal colorings and present a method for sampling them. Then, in Section 5.2, we show how to use this sampling to formulate the problem of finding a partial frugal coloring guaranteeing progress (to be made precise) as a polynomial LLL and how to solve it. In Section 5.3, we explain how – after several iterations of setting up and solving these "progress-guaranteeing" LLLs, gradually extending the partial frugal coloring – we can set up and solve one final polynomial LLL for completing the partial coloring, also based on the sampling method presented in Section 5.1.

5.1 Sampling a Partial Frugal Coloring

▶ **Definition 16** (Partial Frugal Coloring)**.** A partial β-frugal coloring of $G = (V, E)$ is an assignment of colors to a subset $V^* \subseteq V$ such that it is proper in $G[V^*]$ and no node in V has more than β neighbors with the same color. In other words, it is a β-frugal coloring of $G[V^*]$ with the additional condition that no uncolored node in $V' := V \setminus V^*$ has more than β neighbors in V^* with the same color.

A partial coloring naturally splits the base graph G into two parts: $G[V^*]$ induced by colored nodes and $G[V']$ induced by uncolored nodes. However, the problem of extending or completing a partial frugal coloring does not only depend on $G[V']$, but also on the base graph G. That is why we introduce the notion of base-graph degree, a property of the uncolored set V' with respect to the base graph G.

▶ **Definition 17** (Base-Graph Degree of a Partial (Frugal) Coloring)**.** Given a partial coloring, we call the number $d(v, V')$ of neighboring uncolored nodes of a node $v \in V$ its *base-graph degree* into the uncolored set V'. Moreover, we call the maximum base-graph degree Δ' of a node $v \in V$ into V' the base-graph degree of V'.

In the following, we show how one can sample a partial frugal coloring, thus randomly assign some of the nodes in a set V' of uncolored nodes a color. The main idea of our sampling process is to pick a color uniformly at random, and then discard it if this choice would lead to a violation (in terms of properness and frugality). In order to increase the chances of a node being colored, instead of just sampling one color, each node v samples x different

colors from x different palettes at the same time, for some parameter $x \geq 1$, and then picks the first color that does not lead to a violation. If v has no such violation-free among its x choices, then v remains uncolored.

The next lemma, the proof of which is deferred to the full version, analyzes the probability of two kinds of events: Event (E1) that a node is uncolored. This event is important if we aim to color all the nodes in V'. Event (E2) that the base-graph degree of a node into the set of uncolored nodes in V' is too large. This event is important if we do not aim at a full coloring of all the nodes in V', but we want to ensure that we make enough progress in decreasing the base-graph degree of the uncolored set.

▶ **Lemma 18.** *Let $G = (V, E)$ be a graph with maximum degree Δ, $V' \subseteq V$ an uncolored set with base-graph degree Δ', $\beta \in [\Delta]$, and $x \geq 1$. Then there is an $O(1)$-round randomized distributed algorithm that computes a partial β-frugal $(20 \cdot x \cdot \Delta' \cdot \Delta^{1/\beta})$-coloring of some of the nodes in V' such that*

- **(i)** *the probability that a node in V' is uncolored is at most 10^{-x},*
- **(ii)** *the probability that the base-graph degree of a node $v \in V$ into the uncolored subset of V' is larger than $5^{-x} \cdot \Delta'$ is at most $e^{-\Omega(5^{-x} \cdot \Delta')}$.*

5.2 Iterated Partial Frugal Coloring

In the following, we first show how a "progress-guaranteeing" partial coloring – that is, a coloring that decreases the base-graph degree of every node quickly enough – can be found based on the sampling process presented in Section 5.1. Then, we prove that by iterating this algorithm for $O(\log^* \Delta)$ repetitions, using different palettes in each iteration, the base-graph degree reduces to $O(\sqrt{\Delta})$.

In one iteration, given a set V' of uncolored nodes, we want to color a subset $V^* \subseteq V'$ such that the uncolored nodes $V'' := V' \setminus V^*$ have a base-graph degree Δ'' that is sufficiently smaller than the base-graph degree Δ' of V'. Note that the sampling of Section 5.1 only provides us with a partial coloring where every node is likely to have a decrease in the base-graph degree. Here, however, we want to enforce that for every node in V there is such a decrease. To this end, we set up an LLL as follows.

LLL Formulation for "Progress-Guaranteeing" Coloring. Performing the sampling of Lemma 18, we have a bad event D_v for every node $v \in V$ that its base-graph degree into V'' is larger than $\Delta'' = 5^{-x} \cdot \Delta'$. By Lemma 18 (ii), we know that the probability of D_v is at most $e^{-\Omega(5^{-x} \cdot \Delta')}$. Moreover, the dependency degree is at most $d \leq \Delta^2$. This LLL thus satisfies the polynomial criterion.

However, as d might be large, we cannot directly apply the LLL algorithm of Theorem 1. In the following, we present an alternative way of finding a partial coloring ensuring a drop in the base-graph degree of every node. In a nutshell, the idea is to just perform one sampling step of a partial frugal coloring, as described in Section 5.1, and then deal with nodes associated with bad events (to be made precise) separately, by setting up another "progress-guaranteeing" LLL. While the first LLL on the whole graph could not be solved directly, the second LLL is formulated only for a "small" subset of nodes, which allows an efficient solution. Because of the two trials of solving an LLL, we lose a 2-factor in the total number of colors. The proof of the next lemma appears in the full version.

▶ **Lemma 19.** *Given a partial β-frugal coloring with uncolored set V' with base-graph degree Δ' and a parameter $x \geq 1$ such that $5^{-x} \cdot \Delta' = \Omega(\sqrt{\Delta})$, there is a $2^{O(\sqrt{\log \log n})}$-round randomized distributed algorithm that computes a partial β-frugal $(40 \cdot x \cdot \Delta' \cdot \Delta^{1/\beta})$-coloring such that the uncolored set has base-graph degree at most $\Delta'' = 5^{-x} \cdot \Delta'$.*

The next lemma describes how through iterated application of finding partial colorings, as supplied by Lemma 19, the base-graph degree of the uncolored set decreases to $O(\sqrt{\Delta})$ after $O(\log^* \Delta)$ rounds and using $O(\Delta^{1+1/\beta})$ colors. The proof appears in the full version.

▶ **Lemma 20.** *There is a $2^{O(\sqrt{\log\log n})}$-round randomized algorithm that computes a partial β-frugal $(80 \cdot \Delta^{1+1/\beta})$-coloring such that the uncolored set V' has base-graph degree $O(\sqrt{\Delta})$.*

5.3 Completing a Partial Frugal Coloring

In this section, we describe how, once the base-graph degree is $O(\sqrt{\Delta})$, all the remaining uncolored nodes can be colored, hence completing the partial frugal coloring. We first give a general formulation for the completion of partial frugal colorings.

LLL Formulation for Completion of Partial Frugal Coloring. Performing the sampling of Lemma 18, we have a bad event U_v for every node $v \in V$ that it is uncolored. By Lemma 18 (i), the probability of U_v is at most 10^{-x}. Moreover, the dependency degree d is at most Δ^2. This LLL satisfies the polynomial criterion if $x = \Omega(\log \Delta)$.

In the following lemma, the proof of which appears in the full version, we show to solve this LLL. The idea is to first perform one sampling step (of Lemma 18), which shatters the graph into "small" components of uncolored nodes, then to set up an LLL for completing the partial coloring, and finally to solve it by employing our deterministic LLL algorithm, on each of the components.

▶ **Lemma 21.** *Given a partial β-frugal coloring and a set V' of uncolored nodes with base degree $\Delta' = O(\sqrt{\Delta})$, there is a $2^{O(\sqrt{\log\log n})}$-round randomized algorithm that completes this β-frugal coloring, by assigning colors to all nodes in V', using $40 \cdot \Delta^{1+1/\beta}$ additional colors.*

A wrap-up of these results about iterated partial colorings and completing a partial coloring immediately leads to a proof of Theorem 15.

Proof of Theorem 15. We first apply the iterated coloring algorithm of Lemma 20 with $80 \cdot \Delta^{1+1/\beta}$ colors, in $2^{O(\sqrt{\log\log n})}$ rounds. Then, we run the algorithms of Lemma 21 to complete this partial coloring with $40 \cdot \Delta^{1+1/\beta}$ additional colors, in $2^{O(\sqrt{\log\log n})}$ rounds. This yields a β-frugal $(120 \cdot \Delta^{1+1/\beta})$-coloring, in $2^{O(\sqrt{\log\log n})}$ rounds. ◀

References

1 Noga Alon. A parallel algorithmic version of the local lemma. In *Proc. of the Symp. on Found. of Comp. Sci. (FOCS)*, pages 586–593. IEEE, 1991.

2 Noga Alon and Joel H. Spencer. *The probabilistic method.* John Wiley & Sons, 2004.

3 Baruch Awerbuch, Bonnie Berger, Lenore Cowen, and David Peleg. Fast distributed network decompositions and covers. *J. of Parallel and Distributed Comp.*, 39(2):105–114, 1996.

4 Baruch Awerbuch, Michael Luby, Andrew V. Goldberg, and Serge A. Plotkin. Network decomposition and locality in distributed computation. In *Proc. of the Symp. on Found. of Comp. Sci. (FOCS)*, pages 364–369, 1989.

5 Baruch Awerbuch and David Peleg. Sparse partitions. In *Proc. of the Symp. on Found. of Comp. Sci. (FOCS)*, pages 503–513, 1990.

6 Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The locality of distributed symmetry breaking. *Journal of the ACM (JACM)*, 63(3):20, 2016.

7 József Beck. An algorithmic approach to the Lovász Local Lemma. I. *Random Structures & Algorithms*, 2(4):343–365, 1991.

8 Sebastian Brandt, Orr Fischer, Juho Hirvonen, Barbara Keller, Tuomo Lempiäinen, Joel Rybicki, Jukka Suomela, and Jara Uitto. A lower bound for the distributed Lovász Local Lemma. In *Proc. of the Symp. on Theory of Comp. (STOC)*, pages 479–488. ACM, 2016.

9 Karthekeyan Chandrasekaran, Navin Goyal, and Bernhard Haeupler. Deterministic algorithms for the Lovász local lemma. In *Pro. of ACM-SIAM Symp. on Disc. Alg. (SODA)*, pages 992–1004, 2010.

10 Yi-Jun Chang, Tsvi Kopelowitz, and Seth Pettie. An exponential separation between randomized and deterministic complexity in the local model. In *Proc. of the Symp. on Found. of Comp. Sci. (FOCS)*, 2016.

11 Yi-Jun Chang and Seth Pettie. A time hierarchy theorem for the local model. In *Proc. of the Symp. on Found. of Comp. Sci. (FOCS)*, 2017, preprint arXiv:1704.06297.

12 Kai-Min Chung, Seth Pettie, and Hsin-Hao Su. Distributed algorithms for the Lovász Local Lemma and graph coloring. In *the Proc. of the Int'l Symp. on Princ. of Dist. Comp. (PODC)*, pages 134–143, 2014.

13 Artur Czumaj and Christian Scheideler. A new algorithm approach to the general Lovász local lemma with applications to scheduling and satisfiability problems. In *Proc. of the Symp. on Theory of Comp. (STOC)*, pages 38–47, 2000.

14 Paul Erdős and László Lovász. Problems and results on 3-chromatic hypergraphs and some related questions. *Infinite and finite sets*, 10(2):609–627, 1975.

15 Manuela Fischer and Mohsen Ghaffari. Sublogarithmic distributed algorithms for Lovász local lemma, and the complexity hierarchy. *preprint arXiv:1705.04840*, 2017. URL: `https://arxiv.org/abs/1705.04840`.

16 Pierre Fraigniaud, Marc Heinrich, and Adrian Kosowski. Local conflict coloring. In *Proc. of the Symp. on Found. of Comp. Sci. (FOCS)*, pages 625–634. IEEE, 2016.

17 Mohsen Ghaffari. An improved distributed algorithm for maximal independent set. In *Pro. of ACM-SIAM Symp. on Disc. Alg. (SODA)*, 2016.

18 Mohsen Ghaffari and Hsin-Hao Su. Distributed degree splitting, edge coloring, and orientations. In *Pro. of ACM-SIAM Symp. on Disc. Alg. (SODA)*, 2017.

19 David G. Harris. Lopsidependency in the Moser-Tardos framework: Beyond the lopsided Lovász local lemma. *ACM Trans. Algorithms*, 13(1):17:1–17:26, December 2016.

20 David G. Harris, Johannes Schneider, and Hsin-Hao Su. Distributed $(\Delta+1)$-coloring in sublogarithmic rounds. In *Proc. of the Symp. on Theory of Comp. (STOC)*, pages 465–478, 2016.

21 David G. Harris and Aravind Srinivasan. The Moser-Tardos framework with partial resampling. In *Proc. of the Symp. on Found. of Comp. Sci. (FOCS)*, pages 469–478. IEEE, 2013.

22 David G. Harris and Aravind Srinivasan. A constructive algorithm for the Lovász local lemma on permutations. In *Pro. of ACM-SIAM Symp. on Disc. Alg. (SODA)*, SODA'14, pages 907–925, 2014.

23 David G. Harris and Aravind Srinivasan. Algorithmic and enumerative aspects of the Moser-Tardos distribution. In *Pro. of ACM-SIAM Symp. on Disc. Alg. (SODA)*, pages 2004–2023, 2016.

24 Kashyap Babu Rao Kolipaka and Mario Szegedy. Moser and Tardos meet Lovász. In *Proc. of the Symp. on Theory of Comp. (STOC)*, pages 235–244, 2011.

25 Nathan Linial. Distributive graph algorithms – global solutions from local data. In *Proc. of the Symp. on Found. of Comp. Sci. (FOCS)*, pages 331–335. IEEE, 1987.

26 Michael Molloy and Bruce Reed. Further algorithmic aspects of the local lemma. In *Proc. of the Symp. on Theory of Comp. (STOC)*, pages 524–529. ACM, 1998.

27 Michael Molloy and Bruce Reed. Graph coloring and the probabilistic method, 2002.

28 Robin A Moser. A constructive proof of the Lovász local lemma. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 343–350. ACM, 2009.

29 Robin A Moser and Gábor Tardos. A constructive proof of the general Lovász Local Lemma. *Journal of the ACM (JACM)*, 57(2):11, 2010.

30 Moni Naor and Larry Stockmeyer. What can be computed locally? *SIAM Journal on Computing*, 24(6):1259–1277, 1995.

31 Alessandro Panconesi and Aravind Srinivasan. Improved distributed algorithms for coloring and network decomposition problems. In *Proc. of the Symp. on Theory of Comp. (STOC)*, pages 581–592. ACM, 1992.

32 David Peleg. *Distributed Computing: A Locality-sensitive Approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

33 Aravind Srinivasan. Improved algorithmic versions of the Lovász local lemma. In *Pro. of ACM-SIAM Symp. on Disc. Alg. (SODA)*, pages 611–620. Society for Industrial and Applied Mathematics, 2008.

Improved Distributed Degree Splitting and Edge Coloring*†

Mohsen Ghaffari[1], Juho Hirvonen[2], Fabian Kuhn[3], Yannic Maus[4], Jukka Suomela[5], and Jara Uitto[6]

1 **ETH Zürich, Switzerland**
ghaffari@inf.ethz.ch
2 **IRIF, CNRS, and University Paris Diderot, France**
juho.hirvonen@irif.fr
3 **University of Freiburg, Germany**
kuhn@cs.uni-freiburg.de
4 **University of Freiburg, Germany**
yannic.maus@cs.uni-freiburg.de
5 **Aalto University, Finland**
jukka.suomela@aalto.fi
6 **ETH Zürich, Switzerland, and University of Freiburg, Germany**
jara.uitto@inf.ethz.ch

Abstract

The degree splitting problem requires coloring the edges of a graph red or blue such that each node has almost the same number of edges in each color, up to a small additive discrepancy. The directed variant of the problem requires orienting the edges such that each node has almost the same number of incoming and outgoing edges, again up to a small additive discrepancy.

We present deterministic distributed algorithms for both variants, which improve on their counterparts presented by Ghaffari and Su [SODA'17]: our algorithms are significantly simpler and faster, and have a much smaller discrepancy. This also leads to a faster and simpler deterministic algorithm for $(2 + o(1))\Delta$-edge-coloring, improving on that of Ghaffari and Su.

1998 ACM Subject Classification C.2.2 Network Protocols

Keywords and phrases Distributed Graph Algorithms, Degree Splitting, Edge Coloring, Discrepancy

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.19

1 Introduction and Related Work

In this work, we present improved distributed (LOCAL model) algorithms for the *degree splitting problem*, and also use them to provide simpler and faster deterministic distributed algorithms for the classic and well-studied problem of *edge coloring*.

LOCAL Model. In the standard LOCAL model of distributed computing[15, 17], the network is abstracted as an n-node undirected graph $G = (V, E)$, and each node is labeled with a unique $O(\log n)$-bit identifier. Communication happens in synchronous rounds of *message*

* Third, fourth and sixth author supported by ERC Grant No. 336495 (ACDC). Second author supported by Ulla Tuominen Foundation.
† A full version of the paper is available at https://arxiv.org/abs/1706.04746.


© Mohsen Ghaffari, Juho Hirvonen, Fabian Kuhn, Yannic Maus, Jukka Suomela, and Jara Uitto;
licensed under Creative Commons License CC-BY

passing, where in each round each node can send a message to each of its neighbors. At the end of the algorithm each node should output its own part of the solution, e.g., the colors of its incident edges in the edge coloring problem. The time complexity of an algorithm is the number of synchronous rounds.

Degree Splitting Problems. The *undirected degree splitting* problem seeks a partitioning of the graph edges E into two parts so that the partition looks almost balanced around each node. Concretely, we should color each edge red or blue such that for each node, the difference between its number of red and blue edges is at most some small *discrepancy* value κ. In other words, we want an assignment $q\colon E \to \{+1, -1\}$ such that for each node $v \in V$, we have $\left|\sum_{e \in E(v)} q(e)\right| \leq \kappa$, where $E(v)$ denotes the edges incident on v. We want κ to be as small as possible.

In the *directed* variant of the degree splitting problem, we should orient all the edges such that for each node, the difference between its number of incoming and outgoing edges is at most a small discrepancy value κ.

Why Should One Care About Distributed Degree Splittings? On the one hand, degree splittings are natural tools for solving other problems with a *divide-and-conquer* approach. For instance, consider the well-studied problem of edge coloring, and suppose that we are able to solve degree splitting efficiently with discrepancy $\kappa = O(1)$. We can then compute an edge coloring with $(2+\varepsilon)\Delta$ colors, for any constant $\varepsilon > 0$; as usual, Δ is the maximum degree of the input graph $G = (V, E)$. For that, we recursively apply the degree splittings on G, each time reapplying it on each of the new colors, for a recursion of height $h = O(\log \varepsilon\Delta)$. This way we partition G in 2^h edge-disjoint graphs, each with maximum degree at most

$$\Delta' = \frac{\Delta}{2^h} + \sum_{i=1}^{h} \frac{\kappa}{2^i} \leq \frac{\Delta}{2^h} + \kappa = O(1/\varepsilon).$$

We can then edge color each of these graphs with $2\Delta' - 1$ colors, using standard algorithms (simultaneously in parallel for all graphs and with a separate color palette for each graph), hence obtaining an overall coloring for G with $2^h \cdot (2\Delta' - 1) \leq 2\Delta + 2^h\kappa = (2+\varepsilon)\Delta$ colors. We explain the details of this relation, and the particular edge coloring algorithm that we obtain using our degree splitting algorithm, later in Section 2.

On the other hand, degree splitting problems are interesting also on their own: they seem to be an elementary locally checkable labeling (LCL) problem[16], and yet, even on bounded degree graphs, their distributed complexity is highly non-trivial. In fact, they exhibit characteristics that are intrinsically different from those of the classic problems of the area, including maximal independent set, maximal matching, $\Delta + 1$ vertex coloring, and $2\Delta - 1$ edge coloring. All of these classic problems admit trivial sequential greedy algorithms, and they can also be solved very fast distributedly on bounded degree graphs, in $\Theta(\log^* n)$ rounds[15]. In contrast, degree splittings constitute a middle ground in the complexity: even on bounded degree graphs, deterministic degree splitting requires $\Omega(\log n)$ rounds, as shown by Chang et al. [6], and randomized degree splitting requires $\Omega(\log \log n)$ rounds, as shown by Brandt et al. [4]. These two lower bounds were presented for the *sinkless orientation* problem, introduced by Brandt et al. [4], which can be viewed as a very special case of directed degree splitting: In sinkless orientation, we should orient the edges so that each node of degree at least d, for some large enough constant d, has at least one outgoing edge. For this special case, both lower bounds are tight[11].

What is Known? First, we discuss the existence of low-discrepancy degree splittings. Any graph admits an undirected degree splitting with discrepancy at most 2. This is the best possible, as can be seen on a triangle. This low-discrepancy degree splitting can be viewed as a special case of a beautiful area called *discrepancy theory* (see e.g. [7] for a textbook coverage), which studies coloring the elements of a ground set red/blue so that each of a collection of given subsets has almost the same number of red and blue elements, up to a small additive discrepancy. For instance, by a seminal result of Beck and Fiala from 1981[2], any hypergraph of rank t (each hyperedge has at most t vertices) admits a red/blue edge coloring with per-node discrepancy at most $2t-2$. See [5, 3] for some slightly stronger bounds, for large t. In the case of standard graphs, where $t=2$, the existence proof is straightforward: Add a dummy vertex and connect it to all odd-degree vertices. Then, take an Eulerian tour, and color its edges red and blue in an alternating manner. In directed splitting, a discrepancy of $\kappa = 1$ suffices, using the same Eulerian tour approach and orienting the edges along a traversal of this tour.

In the algorithmic world, Israeli and Shiloach [13] were the first to consider degree splittings. They used it to provide an efficient parallel (PRAM model) algorithm for maximal matching. This, and many other works in the PRAM model which later used degree splittings (e.g., [14]) relied on computing Eulerian tours, following the above scheme. Unfortunately, this idea cannot be used efficiently in the distributed setting, as an Eulerian tour is a non-local structure: finding and alternately coloring it needs $\Omega(n)$ rounds on a simple cycle.

Inspired by Israeli and Shiloach's method [13], Hanckowiak et al. [12] were the first to study degree splittings in distributed algorithms. They used it to present the breakthrough result of a polylog(n)-round deterministic distributed maximal matching, which was the first efficient deterministic algorithm for one of the classic problems. However, for that, they ended up having to relax the degree splitting problem in one crucial manner: they allowed a $\delta = 1/\operatorname{polylog} n$ fraction of nodes to have arbitrary splits, with no guarantee on their balance. As explained by Czygrinow et al. [8], this relaxation ends up being quite harmful for edge coloring; without fixing that issue, it seems that one can get at best an $O(\Delta \log n)$-edge coloring.

Very recently, Ghaffari and Su[11] presented solutions for degree splitting without sacrificing any nodes, and used this to obtain the first polylog n round algorithm for $(2+o(1))\Delta$-edge coloring, improving on prior polylog(n)-round algorithms that used more colors: the algorithm of Barenboim and Elkin [1] for $\Delta \cdot \exp(O(\frac{\log \Delta}{\log\log \Delta}))$ colors, and the algorithm of Czygrinow et al. [8] for $O(\Delta \log n)$ colors. The degree splitting algorithm of Ghaffari and Su[11] obtains a discrepancy $\kappa = \varepsilon\Delta$ in $O((\Delta^2 \log^5 n)/\varepsilon)$ rounds. Their method is based on iterations of flipping augmenting paths (somewhat similar in style to blocking flows in classic algorithms for the maximum flow problem[9]) but the process of deterministically and distributedly finding enough disjoint augmenting paths is quite complex. Furthermore, that part imposes a crucial limitation on the method: it cannot obtain a discrepancy better than $\Theta(\log n)$. As such, this algorithm does not provide any meaningful solution in graphs with degree $o(\log n)$.

Our Contributions. Our main result is a deterministic distributed algorithm for degree splitting that improves on the corresponding result of [11]. The new algorithm is (1) simpler, (2) faster, and (3) it gives a splitting with a much lower discrepancy.

▶ **Theorem 1.** *For every $\varepsilon > 0$, there are deterministic $O\big(\varepsilon^{-1}\cdot\log\varepsilon^{-1}\cdot\big(\log\log\varepsilon^{-1}\big)^{1.71}\cdot\log n\big)$-round distributed algorithms for computing directed and undirected degree splittings with the following properties:*

(a) *For directed degree splitting, the discrepancy at each node v of degree $d(v)$ is at most $\varepsilon \cdot d(v) + 1$ if $d(v)$ is odd and at most $\varepsilon \cdot d(v) + 2$ if $d(v)$ is even.*
(b) *For undirected degree splitting, the discrepancy at each node v of degree $d(v)$ is at most $\varepsilon \cdot d(v) + 4$.*

An important corollary of this splitting result is a faster and simpler algorithm for $(2 + o(1))\Delta$-edge coloring, which improves on the corresponding result from [11]. The related proof is deferred to the full version [10].

▶ **Corollary 2.** *For every $\varepsilon > 1/\log \Delta$, there is a deterministic distributed algorithm that computes a $(2 + \varepsilon)\Delta$-edge coloring in $O\big(\log^2 \Delta \cdot \varepsilon^{-1} \cdot \log\log \Delta \cdot (\log\log\log \Delta)^{1.71} \cdot \log n\big)$ rounds.*

This is significantly faster than the $O(\log^{11} n/\varepsilon^3)$-round algorithm of [11]. Furthermore, we are hopeful that with the future improvements in edge coloring for low-degree graphs, this splitting result will play an even more important role. Ideally, in the ultimate solution for edge coloring, say with $(1 + o(1))\Delta$ colors, this splitting will be one half of the solution: This half brings down the degree to a small value, with a negligible $(1 + o(1))$ factor loss, and the other half would hopefully color those small degree graphs efficiently.

Theorem 1 has another fascinating consequence. Assume that we have a graph in which all nodes have an odd degree. If $\varepsilon < 1/\Delta$, we get a directed degree splitting in which each node v has outdegree either $\lfloor d(v)/2 \rfloor$ or $\lceil d(v)/2 \rceil$. Note that the number of nodes for which the outdegree is $\lfloor d(v)/2 \rfloor$ has to be exactly $n/2$. We therefore get an efficient distributed algorithm to exactly divide the number of nodes into two parts of equal size in any odd-degree graph. For bounded-degree graphs, the algorithm even runs in time $O(\log n)$.

Our Method in a Nutshell. The main technical contribution is a distributed algorithm that partitions the edge set of a given graph in *edge-disjoint short paths* such that each node is the start or end of at most δ paths. We call such a partition a *path decomposition* and δ its *degree*. Now if we orient each path of a path decomposition with degree δ consistently, we obtain an orientation of discrepancy at most δ. Moreover, such an orientation can be computed in time which is linear in the maximum path length.

To study path decompositions in graph G, it is helpful to consider an auxiliary graph H in which each edge $\{u, v\}$ represents a path from u to v in G; now δ is the maximum degree of graph H. To construct a low-degree path decomposition where δ is small, we can start with a trivial decomposition $H = G$, and then repeatedly join pairs of paths: we can replace the edges $\{u, v_1\}$ and $\{u, v_2\}$ in graph H with an edge $\{v_1, v_2\}$, and hence make the degree of u lower, at a cost of increasing the path lengths—this operation is called a *contraction* here.

If each node u simply picked arbitrarily some edges $\{u, v_1\}$ and $\{u, v_2\}$ to contract, this might result in long paths or cycles. The key idea is that we can use a *high-outdegree orientation* to select a good set of edges to contract: Assume that we have an orientation in H such that all nodes have outdegree at least $2k$. Then each node could select k pairs of outgoing edges to contract; this would reduce the maximum degree of H from δ to $\delta - 2k$ and only double the maximum length of a path.

In essence, this idea makes it possible to *amplify* the quality of an orientation algorithm: Given an algorithm A that finds an orientation with a large (but not optimal) outdegree, we can apply A repeatedly to reduce the maximum degree of H. This will result in a low-degree path decomposition of G, and hence also provide us with a well-balanced orientation in G.

Outline. In Section 2 we show how to partitioning graphs into edge-disjoint short paths. In Section 3 we use these results to prove our main result on the distributed computation of edge-splittings (Theorem 1). The proof of Corollary 2 is deferred to the full version of the paper [10].

2 Short Path Decompositions

The basic building block of our approach is to find consistently oriented and short (length $O(\Delta)$) paths in an oriented graph. The first crucial observation is that an oriented path going through a node v is "good" from the perspective of v in the sense that it provides exactly one incoming and one outgoing edge to v. Another important feature is that flipping a consistently oriented path does not increase the discrepancy between incoming and outgoing edges for any non-endpoint node along the path. Following these observations, we recursively decompose a graph into a set of short paths, and merge the paths to ensure that every node is at the end of only a few paths. If a node v is at the end of $\delta(v)$ paths an arbitrary orientation of these paths will provide a split with discrepancy at most $\delta(v)$ for v.

The recursive graph operations may turn graphs into multigraphs with self-loops. Thus throughout the paper a multigraph is allowed to have self-loops and the nodes of a path $v_1, \ldots, v_k$ do not need to be distinct; however, a path can contain each edge at most once. A self-loop at a node v contributes two to the degree of v.

2.1 Orientations and Edge Contractions

The core concept to merge many paths in parallel in one step of the aforementioned recursion is given by the concept of weak $k(v)$-orientations. We begin by extending and adapting prior work[11] on weak orientations to our needs.

▶ **Definition 3.** A *weak $k(v)$-orientation* of a multigraph $G = (V, E)$ is an orientation of the edges E such that each node $v \in V$ has outdegree at least $k(v)$.

Note that a weak 1-orientation is a *sinkless orientation*. By earlier work, it is known that a weak 1-orientation can be found in time $O(\log n)$ in simple graphs of minimum degree at least three.

▶ **Lemma 4** (sinkless orientation, [11])**.** *A weak 1-orientation can be computed by a deterministic algorithm in $O(\log n)$ rounds in simple graphs with minimum degree 3 (and by a randomised algorithm in $O(\log \log n)$ rounds in the same setting).*

In our proofs, we may face multigraphs with multiple self-loops and with nodes of degree less than three and thus, we need a slightly modified version of this result.

▶ **Corollary 5** (sinkless orientation, [11])**.** *Let $G = (V, E)$ be a multigraph and $W \subseteq V$ a subset of nodes with degree at least three. Then, there is a deterministic algorithm that finds an orientation of the edges such that every node in W has outdegree of at least one and runs in $O(\log n)$ rounds (and a randomised algorithm that runs in $O(\log \log n)$ rounds).*

Proof. For every multi-edge, both endpoints pick one edge and orient it outwards, ties broken arbitrarily. For every self-loop, the node will orient it arbitrarily. This way, every node with an incident multi-edge or self-loop will have an outgoing edge.

From here on, let us ignore the multi-edges and self-loops and focus on the simple graph H remaining after removing the multi-edges. For every node v with degree at most

two in H, we connect v to $3 - d(v)$ copies of the following gadget U. The set of nodes of $U = \{u_1, u_2, u_3, u_4, u_5\}$ is connected as a cycle. Furthermore, we add edges $\{u_2, u_4\}$ and $\{u_3, u_5\}$ to the gadget and connect u_1 to v. This way, the gadget is 3-regular.

In the simple graph constructed by adding these gadgets, we run the algorithm of Lemma 4. Thus, any node of degree at least three in the original graph that was not initially adjacent to a multi-edge or self-loop gets an outgoing edge. Since we know that every node incident to a multi-edge or self-loop in G also has an outgoing edge, the claim follows. ◀

The sinkless orientation algorithm from Corollary 5 immediately leads to an algorithm which finds a weak $\lfloor d(v)/3 \rfloor$-orientation in multigraphs in time $O(\log n)$.

▶ **Lemma 6** (weak $\lfloor d(v)/3 \rfloor$-orientation). *There is a deterministic algorithm that finds a weak $\lfloor d(v)/3 \rfloor$-orientation in time $O(\log n)$ in multigraphs.*

Proof. Partition node v into $\lceil d(v)/3 \rceil$ nodes and split its adjacent edges among them such that $\lfloor d(v)/3 \rfloor$ nodes have exactly three adjacent edges each and the remaining node, if any, has $d(v) \bmod 3$ adjacent edges. Note that the partitioning may cause self-loops to go between two different copies of the same node. Then, use the algorithm from Corollary 5 to compute a weak 1-orientation of the resulting multigraph where degree two or degree one nodes do not have any outdegree requirements. If we undo the partition but keep the orientation of the edges we have a weak $\lfloor d(v)/3 \rfloor$-orientation of the original multigraph. ◀

The techniques in this section need orientations in which nodes have at least two outgoing edges. Lemma 6 provides such orientations for nodes of degree at least six; but for nodes of smaller degree it guarantees only one outgoing edge. It is impossible to improve this for nodes with degree smaller than five (cf. [10, Theorem 7.1]) in time $o(n)$. But we obtain the following result for the nodes with degree five. Its proof relies on different techniques than the techniques in this section, and therefore it is deferred to the full version of the paper.

▶ **Lemma 7** (outdegree 2). *The following problem can be solved in time $O(\log n)$ with deterministic algorithms and $O(\log \log n)$ with randomised algorithms: given any multigraph, find an orientation such that all nodes of degree at least 5 have outdegree at least 2.*

The concept of weak orientations can be extended to both indegrees and outdegrees.

▶ **Definition 8.** A *strong $k(v)$-orientation* of a multigraph $G = (V, E)$ is an orientation of the edges E such that each node $v \in V$ has both indegree and outdegree at least $k(v)$.

2.2 Path Decompositions

We now introduce the concept of a path decomposition. The decomposition proves to be a strong tool due to the fact that it can be turned into a strong orientation (cf. Lemma 11).

▶ **Definition 9.** Given a multigraph $G = (V, E)$, a positive integer λ, and a function $\delta \colon V \to \mathbb{R}_{\geq 0}$, we call a partition $\mathcal{P}$ of the edges E into disjoint paths $P_1, \ldots, P_\rho$ a (δ, λ)-*path decomposition* if

- for every $v \in V$ there are at most $\delta(v)$ paths that start or end in v,
- each path P_i is of length at most λ.

For each path decomposition $\mathcal{P}$, we define the multigraph $G(\mathcal{P})$ as follows: the vertex set of $G(\mathcal{P})$ is V, and there is an edge between two nodes $u, v \in V$ if $\mathcal{P}$ has a path which starts at u and ends at v or vice versa. The *degree of v in $\mathcal{P}$* is defined to be its degree in $G(\mathcal{P})$ and the *maximum degree of the path decomposition $\mathcal{P}$* is the maximum degree of $G(\mathcal{P})$.

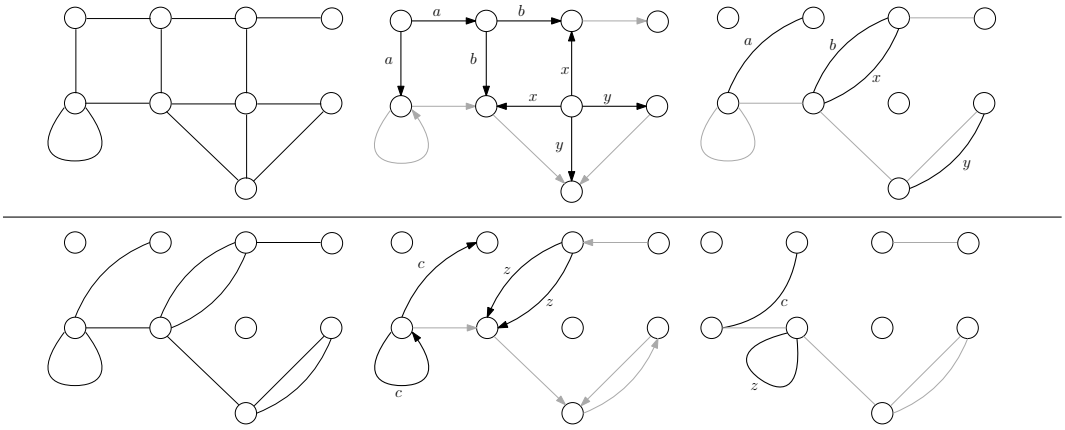


Figure 1 In two sequences of three illustrations this figure depicts two sets of contractions. In each line the first illustration is the situation before the contraction, the second one depicts the orientation and the selected outgoing edges which will be contracted in parallel and the third illustration shows the situation after the contraction where new edges are highlighted.

A contraction may produce isolated nodes, multi-edges and self loops. If a self loop $\{v, v\}$ is selected to be contracted with any other edge $\{v, w\}$ it simply results in a new edge $\{v, w\}$ as if the self loop was any other edge. Such a contraction still reduces the degree of v by two.

Note that we used a graph with small node degrees for illustration purposes. We cannot quickly compute an orientation with large outdegree for nodes with degree less than five.

Notice that $\delta(v)$ is an upper bound on the degree of v in $\mathcal{P}$ and $\max_{v \in V} \delta(v)$ is an upper bound on the maximum degree of the path decomposition. Note that $d_G(v) - d_{G(\mathcal{P})}(v)$ is always even. To make proofs more to the point instead of getting lost in notation, we often identify $G(\mathcal{P})$ with $\mathcal{P}$ and vice versa. A distributed algorithm has computed a path decomposition $\mathcal{P}$ if every node knows the paths of $\mathcal{P}$ it belongs to. Note that it is trivial to compute a $(d(v), 1)$-path decomposition in 1 round, because every edge can form a separate path.

Let $\lfloor \cdot \rfloor_*$ denote the function which rounds down to the previous even integer, that is, $\lfloor x \rfloor_* = 2\lfloor x/2 \rfloor$. The following virtual graph transformation, which we call *edge contraction*, is the core technical construction in this section.

Disjoint Edge Contraction

The basic idea behind edge contraction is to turn two incident edges $\{v, u\}$ and $\{v, w\}$ into a single edge $\{u, w\}$ by removing the edges $\{v, u\}$ and $\{v, w\}$ and adding a new edge $\{u, w\}$. We say that node v contracts when an edge contraction is performed on some pair of edges $\{v, u\}$ and $\{v, w\}$. When node v performs a contraction of edges $\{v, u\}$ and $\{v, w\}$, its degree $d(v)$ is reduced by two while maintaining the degrees of u and w. Furthermore, any set of nodes contracting any set of their incident edges at most doubles the distance between any pair of non-isolated nodes. Notice that adjacent nodes can only contract disjoint pairs of edges in parallel and a contraction may also produce isolated nodes, multi-edges and self-loops. If a self-loop $\{v, v\}$ is selected to be contracted with any other edge $\{v, w\}$ it simply results in a new edge $\{v, w\}$ as if the self-loop was any other edge. Such a contraction still reduces the degree of v by two as the self-loop was considered as both – an incoming and an outgoing edge of v. See Figure 1 for an illustration.

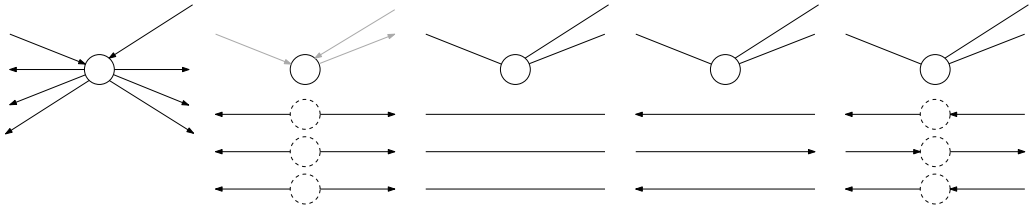


Figure 2 The first two illustrations show that selecting the outgoing edges for a contraction can be seen as dividing the node into a set of virtual nodes, each incident to two outgoing edges. Then, in the third illustration, the contraction is obtained by removing the virtual nodes but keeping the connection alive. The last two illustrations show how an orientation on contracted edges is used to orient the edges of the original graph such that virtual nodes obtain an equal split (and such that the original node obtains a good split).

Edge contractions can be used to compute path decompositions, e.g., an edge which is created through a contraction of two edges can be seen as a path of length two. If an edge $\{u, v\}$ represents a path from u to v in G, e.g., when recursively applying edge contractions on the graph $G(\mathcal{P})$ for some given path decomposition $\mathcal{P}$, each contraction merges two paths of $\mathcal{P}$. If each node simply picked arbitrarily some edges to contract, this might result in long paths or cycles. The key idea is to use orientations of the edges to find large sets of edges which can be contracted in parallel. If every node only contracts outgoing edges of a given orientation all contractions of all nodes can be performed in parallel.

If we start with a trivial decomposition, i.e., each edge is its own path, and perform k iterations of parallel contraction, where, in each iteration, each node contracts two edges, we obtain a $(d(v)-2k, 2^k)$-path decomposition. If we want the degrees $d(v)-2k$ to be constant we have to choose k, i.e., the number of iterations, in the order of Δ which implies exponentially long paths and runtime as the path lengths (might) double with each contraction.

The technical challenge to avoid exponential runtime is to achieve a lot of parallelism while at the same time reducing the degrees quickly. We achieve this with the help of weak orientation algorithms: An outdegree of $f(v)$ at node v allows the node to contract $\lfloor f(v) \rfloor_*$ edges at the same time and in parallel with all other nodes. If $f(v)$ is a constant fraction of $d(v)$ this implies that $O(\log \Delta)$ iterations are sufficient to reach a small degree. As the runtime is exponential in the number of iterations and the constant in the O-notation might be large, this is still not enough to ensure a runtime which is linear in Δ, up to polylogarithmic terms. Instead, we begin with the weak orientation algorithm from the previous section and iterate it until a path decomposition with a small (but not optimal!) degree is obtained. Then we use it to construct a better orientation algorithm. Then, we use this better orientation to compute an even better one and so on. Recursing with the correct choice of parameters leads to a runtime which is linear in Δ, up to polylogarithmic terms. We take the liberty to use the terms recursion and iteration interchangeably depending on which term is more suitable in the respective context. Refer to Figure 2 for an illustration of the edge contraction technique with a given orientation.

We will now apply a simple version of our contraction technique to obtain a fast and precise path decomposition algorithm in Δ-regular graphs for $\Delta = O(1)$. The result can also be formulated for non-regular graphs, but here we choose regular graphs to focus on the proof idea which is the key theme throughout most proofs of this section.

▶ **Theorem 10** ($(\Delta - 2k, 2^k)$-path decomposition)**.** *Let $G = (V, E)$ be a Δ-regular multigraph. For any positive integer $k \leq \Delta/2 - 2$ there is a deterministic distributed algorithm that computes a $(\Delta - 2k, 2^k)$-path decomposition in time $O(2^k \log n)$.*

Proof. We recursively compute k multigraphs $H_1, \ldots, H_k$ where H_k corresponds to the resulting path decomposition. To obtain H_1, we begin by computing a weak 2-orientation π of G with the algorithm from Lemma 6 (note that by assumption we have $k \geq 1$ and therefore $\Delta \geq 6$). Then, every node contracts a pair of outgoing incident edges. Notice that contractions of adjacent nodes are always disjoint. The degree of v is reduced to $\Delta - 2$ and each edge in the resulting multigraph H_1 consists of a path in G of length at most two.

Applying this method recursively with recursion depth k yields multigraphs $H_1, \ldots, H_k$ where the maximum degree of H_i is $\Delta - 2i$ and each edge in H_i corresponds to a path in G of length at most 2^i. Thus, H_k corresponds to a $(\Delta - 2k, 2^k)$-path decomposition. Note that there is one execution of Lemma 6 in each recursion level and it provides a weak 2-orientation of the respective graph because the degree of each node is at least six due to $i \leq k \leq \Delta/2 - 2$.

One communication round in recursion level i can be simulated in 2^i rounds in the original graph. Thus, the runtime is dominated by the application of Lemma 6 in recursion level k which yields a time complexity of $O(2^k \log n)$. ◀

Next, we show how to turn a (δ, λ)-path decomposition efficiently into a strong orientation. The strong orientation obtained this way has $\delta(v)$ as an upper bound on the discrepancy between in- and outdegree of node v.

▶ **Lemma 11.** *Let $G = (V, E)$ be a multigraph with a given (δ, λ)-path decomposition $\mathcal{P}$. There is a deterministic algorithm that computes a strong $\frac{1}{2}(d(v) - \delta(v))$-orientation of G in $O(\lambda)$ rounds.*

Proof. Let $H = G(\mathcal{P})$ be the virtual graph that corresponds to $\mathcal{P}$ and let π_H be an arbitrary orientation of the edges of H. Let (u, v) be an edge of H oriented according to π_H and let $P = u_1, \ldots, u_k$, where $u_1 = u$ and $u_k = v$, be the path in the original graph G that corresponds to edge (u, v) in H. Now, we orient the path P in a consistent way according to the orientation of (u, v), i.e., edge $\{u_i, u_{i+1}\}$ is directed from u_i to u_{i+1} for all $1 \leq i \leq k$. Since every edge in G belongs to exactly one path in the decomposition, performing this operation for every edge in H provides a unique orientation for every edge in G. Let us denote the orientation obtained this way by π_G.

Consider some node v and observe that orienting any path that contains v but where v is not either the start or the endpoint adds exactly one incoming edge and one outgoing edge for v. Therefore, the discrepancy of the indegrees and outdegrees of v in π_G is bounded from above by the discrepancy in π_H, which is at most $\delta(v)$ by the definition of a (δ, λ)-path decomposition. It follows that π_G is a strong $\frac{1}{2}(d(v) - \delta(v))$-orientation.

Finally, since the length of any path in $\mathcal{P}$ is bounded above by λ, consistently orienting the paths takes λ communication rounds finishing the proof. ◀

In the following, we formally use weak orientations to compute a path decomposition. This lemma will later be iterated in Corollary 13

▶ **Lemma 12.** *Assume that there exists a deterministic distributed algorithm that finds a weak $\left(\left(\frac{1}{2} - \varepsilon\right)d(v) - 2\right)$-orientation in time $T(n, \Delta)$.*

Then, there is a deterministic distributed algorithm that finds a $\left(\left(\frac{1}{2} + \varepsilon\right)d(v) + 4,\, 2\right)$-path decomposition in time $O(T(n, \Delta))$.

Proof. Let G be a multigraph with a weak $\left(\left(\frac{1}{2} - \varepsilon\right)d(v) - 2\right)$-orientation given by the algorithm promised in the lemma statement. Now every node v arbitrarily divides the outgoing edges into pairs and contracts these pairs yielding a multigraph with degree at most

$$d(v) - \left\lfloor \left(\tfrac{1}{2} - \varepsilon\right)d(v) \right\rfloor_* + 2 \leq \left(\tfrac{1}{2} + \varepsilon\right)d(v) + 4.$$

Observing that all of the chosen edge pairs are disjoint yields that the constructed multigraph is a $\big(\big(\frac{1}{2}+\varepsilon\big)d(v)+4,\, 2\big)$-path decomposition. The contraction operation requires one round of communication. ◀

In the following corollary we iterate Lemma 12 to obtain an even better path decomposition. Furthermore, more care is required in the details to avoid rounding errors and to obtain the correct result when the degrees get small. Corollary 13 will be applied many times in proceeding subsections.

▶ **Corollary 13.** *Let $0 < \varepsilon \leq 1/6$. Assume that $T(n,\Delta) \geq \log n$ is the running time of an algorithm $\mathcal{A}$ that finds a weak $\big((1/2-\varepsilon)d(v)-2\big)$-orientation. Then for any positive integer i, there is a deterministic distributed algorithm $\mathcal{B}$ that finds a $\big((1/2+\varepsilon)^i d(v)+4,\, 2^{i+5}\big)$-path decomposition $\mathcal{P}$ in time $O(2^i \cdot T(n,\Delta))$.*

Proof. Let i be a positive integer. We define algorithm $\mathcal{B}$ such that it uses algorithm $\mathcal{A}$ to recursively compute graphs $H_0, H_1, \ldots, H_i, H_{i+1}, \ldots, H_{i+5}$ and path decompositions $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_i, \mathcal{P}_{i+1}, \ldots, \mathcal{P}_{i+5}$. Let $G=(V,E)$ be a multigraph. For $j=0,\ldots,i-1$ we set $H_0 = G$ and $H_{j+1} = H_j(\mathcal{P}_{j+1})$, where $\mathcal{P}_{j+1}$ is the path decomposition which is returned by applying Lemma 12 with algorithm $\mathcal{A}$ on H_j. This guarantees that path decomposition $\mathcal{P}_i$ has maximum degree $(\frac{1}{2}+\varepsilon)^i d(v)+12$. The remaining five graph decompositions are computed afterward (see the end of this proof) and reduce the additive 12 to an additive 4.

Properties of $\mathcal{P}_1, \ldots, \mathcal{P}_i$. We first show that for $j=1,\ldots,i$ the path decomposition $\mathcal{P}_j$ is a $(z_j(v), 2^j)$-path decomposition with

$$z_j(v) = \big(\tfrac{1}{2}+\varepsilon\big)^j d(v) + 4\sum_{k=0}^{j-1}\big(\tfrac{1}{2}+\varepsilon\big)^k.$$

With every application of Lemma 12 the length of the paths at most double in length which implies that the path length of $\mathcal{P}_j$ is upper bounded by 2^j. We now prove by induction that the variables $z_j(v)$, $j=1,\ldots i$ behave as claimed:

- *Base case:* $z_1(v) = \big(\frac{1}{2}+\varepsilon\big)d(v)+4$ follows from the invocation of Lemma 12 with $\mathcal{A}$ on $H_0 = G$.
- *Inductive step:* Using the properties of Lemma 12 we obtain

$$\begin{aligned} z_{j+1}(v) &= \big(\tfrac{1}{2}+\varepsilon\big)z_j(v)+4 \leq \big(\tfrac{1}{2}+\varepsilon\big)\left(\big(\tfrac{1}{2}+\varepsilon\big)^j d(v) + 4\sum_{k=0}^{j-1}\big(\tfrac{1}{2}+\varepsilon\big)^k\right)+4 \\ &= \big(\tfrac{1}{2}+\varepsilon\big)^{j+1} d(v) + 4\sum_{k=0}^{j}\big(\tfrac{1}{2}+\varepsilon\big)^k. \end{aligned}$$

Using the geometric series to bound the last sum and then $\varepsilon \leq 1/6$ we obtain that

$$z_i(v) \leq \big(\tfrac{1}{2}+\varepsilon\big)^i d(v) + 12.$$

Reducing the Additive Term. Now, we compute the five further path decompositions $\mathcal{P}_{i+1}, \ldots, \mathcal{P}_{i+5}$ to reduce the additive term in the degrees of the path decomposition from 12 to 4; in each path decomposition this additive term is reduced by two for certain nodes. In each of the first four path decompositions nodes with degree at least six in the current path decomposition reduce the additive term by at least two: we compute a weak $\lfloor d(v)/3 \rfloor$-orientation (using Lemma 6) and then every node with degree at least six contracts two

outgoing edges. In the last path decomposition we compute an orientation in which every node with degree at least five in the current path decomposition has two outgoing edges (using Lemma 7) and then each of them contracts two incident edges. Thus in the last path decomposition the additive term of nodes with degree five is reduced by two.

To formally prove that we obtain the desired path decomposition let $x_{i+j}(v)$ be the actual degree of node v in $G(\mathcal{P}_{i+j})$ for $j = 0, \ldots, 5$. First note that the degree of a node never increases due to an edge contraction, not even due to an edge contraction which is performed by another node.

Constructing $\mathcal{P}_{i+1}, \ldots, \mathcal{P}_{i+4}$. To compute path decomposition $\mathcal{P}_{i+j+1}$, $j = 0, \ldots, 3$, we compute an orientation of $G(\mathcal{P}_{i+j})$ in which every node v with $x_{i+j}(v) \geq 6$ has outdegree at least two (one can use the algorithm described in Lemma 6). Then $\mathcal{P}_{i+j+1}$ is obtained if every node with $x_{i+j}(v) \geq 6$ contracts two of its incident outgoing edges. So, whenever $x_{i+j}(v) \geq 6$ we obtain that $x_{i+j+1}(v) = x_{i+j}(v) - 2$, that is $x_{i+j+1} \leq z_i(v) - 2(j+1)$. If $x_{i+j}(v) \geq 6$ for all $j = 0, \ldots, 3$ we have

$$x_{i+5}(v) \leq x_{i+4}(v) \leq (1/2 + \varepsilon)^i d(v) + 4.$$

Otherwise, for some $j = 0, \ldots, 3$, we have $x_{i+j}(v) \leq 5$, that is, $x_{i+4}(v) \leq 4$ or $x_{i+4}(v) = 5$. If $x_{i+4}(v) \leq 4$ we have

$$x_{i+5}(v) \leq x_{i+4}(v) \leq 4 \leq (1/2 + \varepsilon)^i d(v) + 4.$$

Constructing $\mathcal{P}_{i+5}$. For nodes with $x_{i+4}(v) = 5$ we compute one more path decomposition. We use Lemma 7 to compute an orientation of $G(\mathcal{P}_4)$ in which each node with degree at least five has two outgoing edges; then each node with at least two outgoing edges contracts one pair of its incident outgoing edges. Thus the degree of nodes with degree five reduces by two and we obtain that the path decomposition $\mathcal{P}_{i+5}$ is a $\big((\frac{1}{2} + \varepsilon)^i d(v) + 4,\ 2^{i+5}\big)$-path decomposition.

Running Time. The time complexity to invoke algorithm $\mathcal{A}$ or the algorithms from Lemma 6 or Lemma 7 on graph H_j is $O(2^j T(n, \Delta))$ because the longest path in H_j has length 2^j and $T(n, \Delta) \geq \log n$. Thus, the total runtime is

$$O\Big(\sum_{j=0}^{i+5} 2^j T(n, \Delta)\Big) = O\big(2^i T(n, \Delta)\big).$$ ◀

2.3 Amplifying Weak Orientation Algorithms

Now, we use Corollary 13 to iterate a given weak orientation algorithm $\mathcal{A}$ to obtain a new weak orientation algorithm $\mathcal{B}$. The goal is that $\mathcal{B}$ has an outdegree guarantee which is much closer to $(1/2)d(v)$ than the guarantee provided by algorithm $\mathcal{A}$.

Let $0 < \varepsilon_2 < \varepsilon_1 \leq \frac{1}{6}$, $\alpha = \frac{1}{2} - \varepsilon_1$, and $\beta = \frac{1}{2} + \varepsilon_1$. The roadmap for the proofs of this section is as follows:

- In the proof of Lemma 15:
  1. Execute i iterations of a weak $(\alpha d(v) - 2)$-orientation algorithm, for an i that will be chosen later, and after each iteration, perform disjoint edge contractions. Thus, obtain a $\big(\beta^i d(v) + 4,\ 2^{i+5}\big)$-path decomposition using Corollary 13.
  2. Apply Lemma 11 to obtain a weak $\big(\frac{1}{2}(1 - \beta^i)d(v) - 2\big)$-orientation.

3. By setting $i = \log(\varepsilon_2)/\log(\beta)$ we get that $\beta^i = \varepsilon_2$ and the running time of steps 1–2 is

$$O(2^i T(n,\Delta)) = O\big(\varepsilon_2^{\log_2^{-1}\beta} \cdot T(n,\Delta)\big) = O\big(\varepsilon_2^{-(1+24\varepsilon_1)} \cdot T(n,\Delta)\big),$$

where $T(n,\Delta)$ is the runtime of the weak $(\alpha d(v)-2)$-orientation algorithm. The last equality holds because with Lemma 14, we obtain that $-\log_2^{-1}\beta \le 1+24\varepsilon_1$ when $\varepsilon_1 \le 1/6$.

- In the proof of Theorem 16:

4. Use Lemma 11 with $\varepsilon_1 = 1/6$ and $\varepsilon_2 = 1/\log\log\Delta$ to obtain an algorithm which computes a weak $\big(\big(\frac{1}{2} - 1/\log\log\Delta\big)d(v) - 2\big)$-orientation and runs in time $O((\log\log\Delta)^{1.71} \cdot \log n)$. In this step, we plug in $\varepsilon_1 = 1/6$ to obtain the exponent

$$-\log_2^{-1}\beta = -\log_2^{-1}\big(\tfrac{1}{2} + \tfrac{1}{6}\big) < 1.71.$$

5. Using the construction twice more, once with $\varepsilon_1 = 1/\log\log\Delta$ and $\varepsilon_2 = 1/\log\Delta$ and once with $\varepsilon_1 = 1/\log\Delta$ and $\varepsilon_2 = 1/\Delta$, yields a weak $\big(\big(\frac{1}{2} - \frac{1}{\Delta}\big)\, d(v) - 2\big)$-orientation algorithm that runs in time $O(\Delta \cdot \log\Delta \cdot (\log\log\Delta)^{1.71} \cdot \log n)$.

The following technical result is used to simplify running times; it is proved in the full version of the paper with a Taylor expansion.

▶ **Lemma 14.** *Let $0 < \varepsilon \le 1/6$. Then, $-\log_2^{-1}\big(\frac{1}{2} + \varepsilon\big) \le 1 + 24\varepsilon$.*

In the following lemma we perform steps 1–3 of the aforementioned agenda.

▶ **Lemma 15.** *Let $0 < \varepsilon_2 < \varepsilon_1 \le \frac{1}{6}$. Assume that there is a deterministic algorithm $\mathcal{A}$ which computes a weak $\big(\big(\frac{1}{2} - \varepsilon_1\big)\, d(v) - 2\big)$-orientation and runs in time $T(n,\Delta)$. Then there is a deterministic weak $\big(\big(\frac{1}{2} - \varepsilon_2\big)\, d(v) - 2\big)$-orientation algorithm $\mathcal{B}$ with running time*

$$O\Big(\varepsilon_2^{\log_2^{-1}(\frac{1}{2}+\varepsilon_1)} \cdot T(n,\Delta)\Big) = O\Big(\varepsilon_2^{-(1+24\varepsilon_1)} \cdot T(n,\Delta)\Big). \tag{1}$$

Proof. Let $i = \log_2(\varepsilon_2)/\log_2(1/2+\varepsilon_1)$. By Lemma 14, we get that $i \le (1+24\varepsilon_1)\log_2(1/\varepsilon_2)$; thus it is sufficient to show the left hand side of (1). By applying Corollary 13 with parameter i and algorithm $\mathcal{A}$, we get a distributed algorithm that finds a $\big((1/2+\varepsilon_1)^r\, d(v) + 4,\ 2^{i+5}\big)$-path decomposition in time

$$O\big(2^i \cdot T(n,\Delta)\big) = O\Big(\varepsilon_2^{\log_2^{-1}(\frac{1}{2}+\varepsilon_1)} \cdot T(n,\Delta)\Big).$$

The degree of node v in the path decomposition is upper bounded by $\big(\frac{1}{2}+\varepsilon_1\big)^i d(v) + 4 = \varepsilon_2 d(v) + 4$. Now Lemma 11 yields a weak $\big(\frac{1}{2}\big(1-\varepsilon_2\big)d(v) - 2\big)$-orientation algorithm with the same running time; in particular, this is a weak $\big(\big(\frac{1}{2} - \varepsilon_2\big)d(v) - 2\big)$-orientation algorithm. ◀

We close the section by performing steps 4–5 of the agenda. Note that the theorem is more general than the outlined agenda as it contains an additional parameter δ which can be used to tune the running time at the cost of the quality of the weak orientation algorithm.

▶ **Theorem 16.** *Let δ be a positive integer. There exist the following deterministic weak orientation algorithms.*

(a) *$\mathcal{A}$: weak $\big(\big(\frac{1}{2} - 1/\log\log\frac{\Delta}{\delta}\big)\, d(v) - 2\big)$-orientation in time $O\big(\big(\log\log\frac{\Delta}{\delta}\big)^{1.71} \cdot \log n\big)$.*

(b) *$\mathcal{B}$: weak $\big(\big(\frac{1}{2} - 1/\log\frac{\Delta}{\delta}\big)\, d(v) - 2\big)$-orientation in time $O\big(\log\frac{\Delta}{\delta} \cdot \big(\log\log\frac{\Delta}{\delta}\big)^{1.71} \cdot \log n\big)$.*

(c) *$\mathcal{C}$: weak $\big(\big(\frac{1}{2} - \frac{\delta}{\Delta}\big)\, d(v) - 2\big)$-orientation in time $O\big(\frac{\Delta}{\delta} \cdot \log\frac{\Delta}{\delta} \cdot \big(\log\log\frac{\Delta}{\delta}\big)^{1.71} \cdot \log n\big)$.*

Proof. Each statement is proven by applying Lemma 15 with different values for ε_1 and ε_2.

(a) We obtain the algorithm $\mathcal{A}$ by applying Lemma 15 with the weak $\lfloor\Delta/3\rfloor$-orientation algorithm from Lemma 6, that is with $\varepsilon_1 = 1/6$, and with $\varepsilon_2 = 1/\log\log(\Delta/\delta)$.
(b) Algorithm $\mathcal{B}$ is obtained by applying Lemma 15 with $\mathcal{A}$, that is $\varepsilon_1 = 1/\log\log(\Delta/\delta)$ and $\varepsilon_2 = 1/\log(\Delta/\delta)$.
(c) Algorithm $\mathcal{C}$ is obtained by applying Lemma 15 with $\mathcal{B}$, that is $\varepsilon_1 = 1/\log(\Delta/\delta)$ and $\varepsilon_2 = 1/(\Delta/\delta) = \delta/\Delta$. ◀

2.4 Short and Low Degree Path Compositions Fast

Our higher level goal is to compute a path decomposition where the degree is as small as possible to obtain a directed split with the discrepancy as small as possible (with methods similar to Lemma 11, also see the proof of Theorem 1). As we will show in the next theorem, with the methods introduced in this section and the appropriate choice of parameters, we can push the maximum degree of the path decomposition down to $\varepsilon d(v) + 4$ for any $\varepsilon > 0$. This is the true limit of this approach because we cannot compute weak 2-orientations of 4-regular graphs in sublinear time (see [10, Theorem 7.1]).

▶ **Theorem 17.** *Let $G = (V, E)$ be a multigraph with maximum degree Δ. For any $\varepsilon > 0$ there is a deterministic distributed algorithm which computes a $(\delta(v), O(1/\varepsilon))$-path decomposition in time $O\big(\alpha \cdot \log\alpha \cdot (\log\log\alpha)^{1.71} \cdot \log n\big)$, where $\alpha = 2/\varepsilon$ and $\delta(v) = \varepsilon d(v) + 3$ if $\varepsilon d(v) \geq 1$ and $\delta(v) = 4$ otherwise.*

Proof. Apply Corollary 13 with algorithm $\mathcal{B}$ from Corollary 16, $\delta = \Delta/\alpha$, and

$$i = \frac{\log \alpha^{-1}}{\log(1/2 + 1/\log(\alpha))}.$$

This implies a path decomposition with degrees $\lfloor \alpha^{-1} d(v) + 4 \rfloor = \lfloor \varepsilon d(v)/2 + 4 \rfloor$. If $\varepsilon d(v) \geq 1$ this is smaller than $\varepsilon d(v) + 3$. If $\varepsilon d(v) < 1$ this is at most 4. The length of the longest path is upper bounded by $O(2^i) = O\big(\alpha^{1+24/\log\alpha}\big) = O(\alpha)$ where we used Lemma 14. The runtime is bounded by $O\big(2^i \cdot T_{\mathcal{B}}(n, \Delta)\big) = O\big(\alpha \cdot \log\alpha \cdot (\log\log\alpha)^{1.71} \cdot \log n\big)$, where $T_{\mathcal{B}}(n, \Delta)$ is the running time of algorithm $\mathcal{B}$. ◀

Choosing $\varepsilon = 1/(2\Delta)$ in Lemma 17 yields the following corollary.

▶ **Corollary 18** (constant degree path decomposition)**.** *There is a deterministic algorithm which computes a $(4, O(\Delta))$-path decomposition in time $O\big(\Delta \cdot \log\Delta \cdot (\log\log\Delta)^{1.71} \cdot \log n\big)$.*

▶ Remark. For any positive integer k smaller than $\log^*(\alpha) \pm O(1)$ one can improve the runtime of Lemma 17 to $O\big(\alpha \cdot (\log^{(k)}\alpha)^{0.71} \cdot \log n \cdot \Pi_{j=1}^{k} \log^{(j)}\alpha\big)$, where $\log^{(j)}(\cdot)$ denotes the j times iterated logarithm, $\alpha = 2/\epsilon$ and the constant in the O-notation grows exponentially in k. This essentially follows from a version of Theorem 16 that turns a weak $\big((1/2 - 1/\log^{(k)}\alpha)d(v) - 2\big)$-orientation algorithm into a weak $\big((1/2 - 1/\log\alpha)d(v) - 2\big)$-orientation algorithm in $k - 1$ iterations.

3 Directed and Undirected Splits

First note that an arbitrary consistent orientation of the paths in the best path decomposition of Section 2 would result in a splitting in which each node v has discrepancy at most $\varepsilon \cdot d(v) + 4$. In the case of directed splitting we slightly tune this by consistently orienting the paths in

such a way that each node has at least one outgoing and one incoming path. As the graph corresponding to the path decomposition is a low degree graph this is the same as finding sinkless and sourceless orientations in low-degree graphs; the following corollary states that these orientations can be computed efficiently. Its proof can be found in the full version of the paper [10].

▶ **Corollary 19** (sinkless and sourceless orientation)**.** *The following problem can be solved in time $O(\log n)$ with deterministic algorithms and $O(\log\log n)$ with randomised algorithms: given any multigraph, find an orientation such that all nodes of degree at least 3 have outdegree and indegree at least 1.*

We are now ready to prove our main result:

▶ **Theorem 1.** *For every $\varepsilon > 0$, there are deterministic $O\big(\varepsilon^{-1}\cdot\log\varepsilon^{-1}\cdot\big(\log\log\varepsilon^{-1}\big)^{1.71}\cdot\log n\big)$-round distributed algorithms for computing directed and undirected degree splittings with the following properties:*

- **(a)** *For directed degree splitting, the discrepancy at each node v of degree $d(v)$ is at most $\varepsilon\cdot d(v)+1$ if $d(v)$ is odd and at most $\varepsilon\cdot d(v)+2$ if $d(v)$ is even.*
- **(b)** *For undirected degree splitting, the discrepancy at each node v of degree $d(v)$ is at most $\varepsilon\cdot d(v)+4$.*

Proof. For both parts apply Lemma 17, which provides a $\big(\delta(v), O(1/\varepsilon)\big)$-path decomposition $\mathcal{P}$ with $\delta(v) = \varepsilon d(v)+3$ if $\varepsilon d(v)\geq 1$ and $\delta(v)=4$ otherwise.

Proof of (b) Nodes color each path of $\mathcal{P}$ alternating with red and blue. Because the length of a path in $\mathcal{P}$ is bounded by $O(1/\varepsilon)$ this can be done in $O(1/\varepsilon)$ rounds.

Consider some node v and observe that v has one red and one blue edge for any path where v is not a startpoint or endpoint. Thus the discrepancy of node v is bounded above by $\delta(v)\leq\varepsilon d(v)+4$.

Proof of (a) Use Corollary 19 to compute an orientation $\pi_{\mathcal{P}}$ of $G(\mathcal{P})$ in which all nodes which have degree at least three in $G(\mathcal{P})$ have at least one incoming and one outgoing edge. Then orient paths in the original graph according to $\pi_{\mathcal{P}}$ as in the proof of Lemma 11 and denote the resulting orientation of the edges of G with π_G.

Consider some node v and observe that orienting any path that contains v but where v is not a startpoint or endpoint adds exactly one incoming edge and one outgoing edge for v. Therefore, the discrepancy of the indegrees and outdegrees of v in $\pi_{\mathcal{P}}$ bounds from above the discrepancy of the indegrees and outdegrees in π_G. The goal is to upper bound this discrepancy as desired.

Therefor let $d_{\mathcal{P}}(v)$ denote the degree of v in $G(\mathcal{P})$. If $d_{\mathcal{P}}(v)$ is at least three then its discrepancy in $\pi_{\mathcal{P}}$ is bounded by $d_{\mathcal{P}}(v)-2$ as the algorithm from Corollary 19 provided one incoming and one outgoing edge for v in $G(\mathcal{P})$. Furthermore we obtain that $d_{\mathcal{P}}(v)$ and $d(v)$ have the same parity because $d(v) = d_{\mathcal{P}}(v)+2x$ holds where x is the number of paths that contain v but where v is neither a startpoint nor an endpoint. We have the following cases.

- $d_{\mathcal{P}}(v)\geq 3$:
  - $\varepsilon d(v)\geq 1$: v's discrepancy in π_G is bounded by $d_{\mathcal{P}}(v)-2\leq\varepsilon d(v)+1$.
  - $\varepsilon d(v)<1$, $d(v)$ even: v's discrepancy in π_G is bounded by $d_{\mathcal{P}}(v)-2\leq 2$.
  - $\varepsilon d(v)<1$, $d(v)$ odd: As $d_{\mathcal{P}}(v)$ has to be odd and $3\leq d_{\mathcal{P}}(v)\leq\delta(v)=4$ holds we have $d_{\mathcal{P}}(v)=3$. Thus v's discrepancy in π_G is bounded by $d_{\mathcal{P}}(v)-2\leq 1$.

- $d_{\mathcal{P}}(v) < 3$:
  - $d(v)$ even: We have $d_{\mathcal{P}} \in \{0, 2\}$ and v's discrepancy in π_G is also 0 or 2.
  - $d(v)$ odd: We have $d_{\mathcal{P}} = 1$ and v's discrepancy in π_G is also 1.

In all cases we have that the discrepancy of node v is upper bounded by $\varepsilon d(v) + 1$ if $d(v)$ is even and by $\varepsilon d(v) + 2$ if $d(v)$ is even, which proves the result. ◀

References

1 Leonid Barenboim and Michael Elkin. Distributed deterministic edge coloring using bounded neighborhood independence. In *Proc. PODC 2011*, pages 129–138, 2011. doi:10.1007/s00446-012-0167-7.

2 József Beck and Tibor Fiala. "Integer-making" theorems. *Discrete Applied Mathematics*, 3(1):1–8, 1981. doi:10.1016/0166-218X(81)90022-6.

3 Debe Bednarchak and Martin Helm. A note on the beck-fiala theorem. *Combinatorica*, 17(1):147–149, 1997.

4 Sebastian Brandt, Orr Fischer, Juho Hirvonen, Barbara Keller, Tuomo Lempiäinen, Joel Rybicki, Jukka Suomela, and Jara Uitto. A lower bound for the distributed Lovász local lemma. In *Proc. STOC 2016*, pages 479–488, 2016. doi:10.1145/2897518.2897570.

5 Boris Bukh. An improvement of the Beck–Fiala theorem. *Combinatorics, Probability & Computing*, 25(03):380–398, 2016. doi:10.1017/S0963548315000140.

6 Yi-Jun Chang, Tsvi Kopelowitz, and Seth Pettie. An exponential separation between randomized and deterministic complexity in the local model. In *Proc. FOCS 2016*, pages 615–624, 2016. doi:10.1109/FOCS.2016.72.

7 Bernard Chazelle. *The Discrepancy Method: Randomness and Complexity*. Cambridge University Press, 2000.

8 Andrzej Czygrinow, Michał Hańćkowiak, and Michał Karoński. Distributed $O(\Delta \log n)$-edge-coloring algorithm. In *Proc. ESA 2001*, pages 345–355, 2001. doi:10.1007/3-540-44676-1_29.

9 Yefim Dinitz. Dinitz' algorithm: The original version and Even's version. In *Theoretical Computer Science, Essays in Memory of Shimon Even*, pages 218–240. Springer, 2006. doi:10.1007/11685654_10.

10 M. Ghaffari, J. Hirvonen, F. Kuhn, Y. Maus, J. Suomela, and J. Uitto. Improved Distributed Degree Splitting and Edge Coloring. *ArXiv e-prints*, June 2017. arXiv:1706.04746.

11 Mohsen Ghaffari and Hsin-Hao Su. Distributed degree splitting, edge coloring, and orientations. In *Proc. SODA 2017*, pages 2505–2523, 2017.

12 Michał Hańćkowiak, Michał Karoński, and Alessandro Panconesi. On the distributed complexity of computing maximal matchings. *SIAM J. Discrete Math.*, 15(1):41–57, 2001. doi:10.1137/S0895480100373121.

13 Amos Israeli and Yossi Shiloach. An improved parallel algorithm for maximal matching. *Inf. Process. Lett.*, 22(2):57–60, 1986. doi:10.1016/0020-0190(86)90141-9.

14 Howard J. Karloff and David B. Shmoys. Efficient parallel algorithms for edge coloring problems. *J. Algorithms*, 8(1):39–52, 1987. doi:10.1016/0196-6774(87)90026-5.

15 Nathan Linial. Distributive graph algorithms—global solutions from local data. In *Proc. FOCS 1987*, pages 331–335, 1987. doi:10.1109/SFCS.1987.20.

16 Moni Naor and Larry Stockmeyer. What can be computed locally? In *Proc. STOC 1993*, pages 184–193, 1993. doi:10.1145/167088.167149.

17 David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. SIAM, 2000. doi:10.1137/1.9780898719772.

Simple and Near-Optimal Distributed Coloring for Sparse Graphs

Mohsen Ghaffari[1] **and Christiana Lymouri**[2]

1 **ETH Zürich, Switzerland**
`ghaffari@inf.ethz.ch`
2 **ETH Zürich, Switzerland**
`lymouric@student.ethz.ch`

Abstract

Graph coloring is one of the central problems in *distributed graph algorithms*. Much of the research on this topic has focused on coloring with $\Delta+1$ colors, where Δ denotes the maximum degree. Using $\Delta+1$ colors may be unsatisfactory in sparse graphs, where not all nodes have such a high degree; it would be more desirable to use a number of colors that improves with sparsity. A standard measure that captures sparsity is *arboricity*, which is the smallest number of forests into which the edges of the graph can be partitioned.

We present simple randomized distributed algorithms that, with high probability, color any n-node α-arboricity graph:

- using $(2+\varepsilon)\cdot\alpha$ colors, for constant $\varepsilon>0$, in $O(\log n)$ rounds, if $\alpha=\tilde{\Omega}(\log n)$, or
- using $O(\alpha\log\alpha)$ colors, in $O(\log n)$ rounds, or
- using $O(\alpha)$ colors, in $O(\log n\cdot\min\{\log\log n,\ \log\alpha\})$ rounds.

These algorithms are nearly-optimal, as it is known by results of Linial [FOCS'87] and Barenboim and Elkin [PODC'08] that coloring with $\Theta(\alpha)$ colors, or even poly(α) colors, requires $\Omega(\log_\alpha n)$ rounds. The previously best-known $O(\log n)$-time result was a deterministic algorithm due to Barenboim and Elkin [PODC'08], which uses $\Theta(\alpha^2)$ colors. Barenboim and Elkin stated improving this number of colors as an open problem in their Distributed Graph Coloring Book.

1998 ACM Subject Classification G.2.2 Graph Algorithms

Keywords and phrases Distributed Graph Algorithms, Graph Coloring, Arboricity

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.20

1 Introduction and Related Work

Graph coloring is one of the central and well-studied problems in *distributed graph algorithms*, and it has a wide range of applications in networks and distributed systems, prototypically in scheduling conflicting tasks, e.g., transmission in a wireless network. Much of the focus in the area has been on obtaining fast distributed algorithms that compute a $(\Delta+1)$-coloring, where Δ denotes the maximum degree of the graph, see e.g. [1, 2, 4, 6, 9, 11, 12, 13, 14, 17, 18, 22, 23, 25, 26, 27, 28].

For a vast range of "sparse" graphs, using $\Delta+1$ colors is rather unsatisfactory. To take the point to the extreme, coloring a tree – which is obviously 2-colorable – using $\Delta+1$ colors seems quite wasteful. Generally, it is more desirable to obtain colorings in which the number of colors improves if the graph is sparse (everywhere).

In this paper, we present simple and near-optimal randomized distributed algorithms that compute a coloring of the graph with a number of colors that depends on its (everywhere) sparsity, formally the *arboricity* of the graph. We next review the related definitions and discuss the known results. Then, we state our contributions.


© Mohsen Ghaffari and Christiana Lymouri;
licensed under Creative Commons License CC-BY

1.1 Definitions and Setup

Graph Arboricity. A standard measure of (everywhere) sparsity of an undirected graph $G = (V, E)$ is its *arboricity*, defined as

$$\alpha(G) = \max\left\{ \lceil \frac{|E(V')|}{|V'|-1} \rceil \;\middle|\; V' \subseteq V, |V'| > 2 \right\},$$

that is, roughly speaking, the maximum ratio of the number of edges to the number of vertices, among all subgraphs of G. By a beautiful result of Nash-Williams [21], an alternative equivalent formulation is as follows: arboricity $\alpha(G)$ is the minimum number of edge-disjoint forests to which one can partition the edges of G.

The Distributed Model. As standard in distributed graph algorithms, we work with the LOCAL model of distributed computation [18, 24]: The network is abstracted as an undirected graph $G = (V, E)$, with $n = |V|$. Communication happens in synchronous message-passing rounds, and per-round, each node can send one message to each of its neighbors. We note that all of our algorithms work also in the more restricted variant of the model, known as CONGEST [24] model, where each message can contain at most $O(\log n)$ bits. Initially, nodes do not know the topology of the graph, except for knowing the arboricity of the graph $\alpha(G)$. At the end, each node should know its own part of the output, e.g., its own color in a coloring.

1.2 Known Results and Open Problems

Existential Aspects. Any graph G admits a $2\alpha(G)$-coloring, and this bound is tight. For the former, note that one can easily arrange vertices as $v_1, \ldots, v_n$ so that each v_i has at most $2\alpha(G) - 1$ neighbors v_j with higher index $j > i$. Then, one can greedily color this list from v_n to v_1, using $2\alpha(G)$ colors. For the latter, note that a graph made of several disjoint cliques, each with 2α vertices, has arboricity α, and chromatic number 2α.

Known Lower Bounds for Distributed Algorithms. By a classic observation of Linial [18], it is well-understood that having a small arboricity is not a local characteristic of graphs, and any distributed algorithm for coloring with $2\alpha(G)$ colors, or anything remotely close to it, needs $\Omega(\log n)$ rounds. Concretely, Linial [18] pointed out that there exists a graph with girth $\Omega(\log_\Delta n)$ and chromatic number $\Omega(\Delta/\log\Delta)$ [10][1] and thus also arboricity $\alpha = \Omega(\Delta/\log\Delta)$. Graphs of girth $\Omega(\log_\Delta n)$ are indistinguishable from trees (which have arboricity $\alpha = 1$), for distributed algorithms with round complexity $o(\log_\Delta n)$. Hence, no distributed algorithm with round complexity $o(\log_\Delta n)$ can compute a coloring of a tree with maximum degree Δ – which clearly has arboricity $\alpha = 1$ – with less than $\Omega(\Delta/\log\Delta) \gg \text{poly}(\alpha)$ colors.

Barenboim and Elkin [3, 5, 7] presented a strengthening of this result and showed that for any α and $q < n^{1/4}/\alpha$, any distributed algorithm for $O(q \cdot \alpha)$-coloring graphs with arboricity α requires $\Omega(\log_{q\alpha} n)$ rounds.

Known Distributed Algorithms for $(\Delta + 1)$ Coloring. Distributed graph coloring started with Linial's seminal work [18, 19]. Linial's coloring algorithm is an $O(\log^* n)$-round deterministic distributed algorithm that computes an $O(\Delta^2)$-coloring of the input graph. This can

[1] In his original writing [18], Linial referred to such high-girth graphs with chromatic number $\Omega(\sqrt{\Delta})$, but he also added remarks that the bound can probably be improved to $\Omega(\Delta/\log\Delta)$.

be easily turned into a $\Delta+1$ coloring in $O(\Delta^2)$ additional rounds. In Section 2.2, we present a variation of Linial's algorithm due to Barenboim and Elkin [3, 5], which produces an $O(\alpha^2)$-coloring in $O(\log n)$ rounds of a graph G with arboricity α. Since Linial's algorithm, significant advances have been made in the area, which we briefly overview next.

On the side of deterministic algorithms, the best known $(\Delta+1)$-coloring distributed algorithm, in terms of dependency on n, is a $(2^{O(\sqrt{\log n})})$-round algorithm by Panconesi and Srinivasan [23]. In terms of dependency on the maximum degree Δ of the graph, the linear in Δ round complexity remained as the state of the art for deterministic $\Delta+1$-coloring [8], until very recently, when Barenboim [2] presented an $O(\Delta^{3/4}\log\Delta+\log^* n)$-round distributed $(\Delta+1)$-coloring algorithm. This was followed by a work of Fraigniaud, Heinrich, and Kosowski [13], which improved the round complexity to $O(\sqrt{\Delta}\log^{2.5}\Delta+\log^* n)$ rounds.

On the side of randomized algorithms, an $O(\log n)$-round algorithm follows from Luby's maximal independent set (MIS) algorithm [20]. A direct $O(\log n)$-round distributed algorithm was analyzed by Johansson [15]. The fastest known randomized algorithm for $(\Delta+1)$-coloring is due to a recent work of Harris et al. [14] which provides a $(\Delta+1)$-coloring in $O(\sqrt{\log\Delta})+2^{O(\sqrt{\log\log n})}$ rounds, with high probability.

Shortcomings of These Methods in Obtaining Arboricity-Dependent Coloring. All the aforementioned deterministic and randomized algorithms perform in iterations, where in each iteration the graph is colored partially and each node that remains uncolored removes from its palette the colors that are taken by its neighbors, until a proper $(\Delta+1)$-coloring of the whole graph is produced. This fundamental property makes these algorithms inappropriate for our setting of obtaining an arboricity-dependent coloring of the graph. In particular, in a graph G with arboricity α and maximum out-degree $\Delta \gg \alpha$, the above algorithms may fail to produce an $f(\alpha)$-coloring. Next, we present the known results on distributed graph coloring in which the number of colors depends on the arboricity of the graph.

Known Distributed Algorithms for Arboricity-Dependent Coloring. Barenboim and Elkin [3, 5] present a deterministic distributed algorithm that computes an $O(\alpha^2)$ coloring within $O(\log n)$ rounds – which is essentially the time that is proven to be necessary by the above lower bound. If one uses more colors, say $O(q\cdot\alpha^2)$ colors for some parameter $q\geq 1$, the algorithm can be made somewhat faster, running in $O(\log_q n+\log^* n)$ rounds. They also show that by spending more time, particularly $O(\alpha\log n)$ rounds, one can get close to the ideal number of colors and use $\lfloor(2+\varepsilon)\cdot\alpha+1\rfloor$ colors, for any constant $\varepsilon>0$. This can be turned into smoother trade-off, obtaining an $O(t\cdot\alpha)$-coloring, for any $t\in[1,\alpha]$, in $O(\frac{\alpha}{t}\cdot\log n+\alpha\log\alpha)$ rounds.

Kothapoli and Pemmaraju [16] study arboricity-dependent randomized distributed coloring algorithms, although targeting a very different range of parameters: they allow drastically more colors, but then their algorithms run very fast. In particular, they present randomized distributed algorithms for $O(\alpha\cdot n^{1/k})$-coloring in $O(k)$ rounds, when $k\in[\Omega(\log\log n,\sqrt{\log n})]$; see [16, Theorem 1.4] for the precise statements. They also present more detailed trade-offs in [16, Theorem 1.3], when a low out-degree orientation of the graph is provided. By the above lower bounds, we know that if we want something remotely close to 2α colors, or even poly(α) colors, we can allow $\Omega(\log_\alpha n)$ rounds for free. To the best of our understanding, the trade-offs of [16, Theorem 1.4] and [16, Theorem 1.3] are not suitable when $\Omega(\log_\alpha n)$ rounds are allowed, with only one exception: for $\alpha\geq 2^{\omega(\log^{1/3} n)}$, one can obtain an $O(\alpha)$-coloring in $O(\log n)$ rounds, by putting together [16, Theorem 1.3 (ii)] and H-partitions of [5].

Open Problem. Barenboim and Elkin ask in Open Problem 11.11 of their distributed graph coloring book [7]: "*Can one use significantly less than α^2 colors, and still stay within deterministic $O(\log n)$ time?*", immediately followed by adding that "*This question is open even for randomized algorithms*".

1.3 Our Contribution

We present very simple randomized distributed algorithms that make a significant progress on the above open problem:

▶ **Theorem 1.** *For any constant $\varepsilon > 0$, there are randomized distributed algorithms that on any n-node graph with arboricity α, with high probability[2], compute*

- *a $\left(\min\left\{(2+\varepsilon)\alpha + O(\log n \cdot \log\log n), O(\alpha \log \alpha)\right\}\right)$-coloring in $O(\log n)$ rounds,*
- *an $O(\alpha)$-coloring, in $O(\log n \cdot \min\{\log\log n,\ \log \alpha\})$ rounds.*

This theorem achieves a near-optimal coloring as a function of arboricity, with parameter trade-offs that compare favorably to the previous results provided by [5, 16]. In particular, so long as $\alpha = \Omega(\log n \cdot \log\log n)$, we get the almost best-possible $((2+\varepsilon)\cdot\alpha)$-coloring, for $\varepsilon > 0$, in $O(\log n)$ time. For graphs of lower arboricity, we can either spend an $O(\log \alpha) \leq O(\log\log n)$ factor more time and get an $O(\alpha)$-coloring in $O(\log n \cdot \log\log n)$ rounds, or we can use an $O(\log \alpha)$ factor more colors and get a coloring with $O(\alpha \log \alpha) \ll \alpha^2$ colors in $O(\log n)$ time.

2 Warm Up: Reviewing an Algorithm of Barenboim and Elkin [3, 5]

In this section, we review an $O(\log n)$-round deterministic distributed algorithm by Barenboim and Elkin [5] that produces an $O(\alpha^2)$-coloring of any n-node graph $G = (V, E)$ with arboricity α. We note that the paper [5] presents other trade-offs when more time is allowed, as overviewed in Section 1, e.g., $((2+\varepsilon)\cdot\alpha)$-coloring in $O(\alpha \log n)$ time, but these algorithms are less relevant for our target of $O(\log n)$-time algorithms (and also their aforementioned open problem in [7]).

The $O(\log n)$-time $O(\alpha^2)$-coloring algorithm of Barenboim and Elkin [3, 5] consists of two steps. In the first step, we use an algorithm, called H-partition, to compute an orientation of the edges in $O(\log n)$ rounds, such that each node has out-degree at most $O(\alpha)$. In the second step, we compute an $O(\alpha^2)$-coloring in $O(\log^* n)$ rounds, using the low out-degree orientation of step 1. Later in Sections 3 and 4, we will make use of this H-partition method.

2.1 Step 1: Low Out-Degree Orientation via H-partition

We now discuss a deterministic distributed algorithm that, given an n-node graph $G = (V, E)$ with arboricity α, in $O(\log_{1+\varepsilon/2} n)$ rounds, computes an acyclic orientation of the edges such that the maximum out-degree is at most $(2+\varepsilon)\cdot\alpha$, for a given parameter $\varepsilon > 0$.

The main idea behind the algorithm is to partition the nodes into $\ell = \lceil \log_{\frac{2+\varepsilon}{2}} n \rceil$ disjoint subsets $H_1, H_2, ...H_\ell$, such that every node $v \in H_j$ with $j \in \{1, 2, ..., l\}$, has at most $(2+\varepsilon)\cdot\alpha$ neighbors in subsets $\cup_{y=j}^{\ell} H_j$. We refer to partitions that satisfy this property as H-partitions with degree $d \leq (2+\varepsilon)\cdot\alpha$ and size $\ell = \lceil \log_{\frac{2+\varepsilon}{2}} n \rceil$. We refer to subsets $H_1, H_2, ..., H_\ell$ as layers of the H-partition. In Lemma 2 we sketch the algorithm for computing an H-partition.

[2] As standard, we use the phrase *with high probability* (w.h.p.) to indicate that an event happens with probability at least $1 - 1/n^c$, for a desirably large constant $c \geq 2$.

Once an H-partition is computed, we orient the edges that have endpoints in different layers H_j and H'_j, for $j' > j$, towards the higher layer H'_j, and orient the edges which have endpoints in the same layer towards the greater ID endpoint. This ensures that we have an acyclic orientation with maximum out-degree at most $d \leq (2+\varepsilon) \cdot \alpha$.

▶ **Lemma 2.** *For a graph G with arboricity α and a parameter $\varepsilon > 0$, there is a deterministic distributed algorithm that computes an H-partition of G with degree $d \leq (2+\varepsilon) \cdot \alpha$ and size $\ell = \lceil \log_{\frac{2+\varepsilon}{2}} n \rceil$ in $O(\log_{\frac{2+\varepsilon}{2}} n)$ rounds.*

Proof Sketch. A graph with arboricity α has at least $\frac{\varepsilon}{2+\varepsilon} \cdot n$ nodes with degree at most $(2+\varepsilon) \cdot \alpha$, as can be seen by a simple double-counting of edges. These nodes join layer H_1. In the subgraph $G \setminus H_1$, there are at least $\frac{\varepsilon}{2+\varepsilon} \cdot (n - |V(H_1)|)$ nodes with degree at most $(2+\varepsilon) \cdot \alpha$. These nodes join layer H_2. Iteratively, in the subgraph $G \setminus \cup_{y=1}^{j} H_y$ there are at least $\frac{\varepsilon}{2+\varepsilon} \cdot (n - \sum_{y=1}^{j} |V(H_y)|)$ nodes with degree at most $(2+\varepsilon) \cdot \alpha$; these nodes join layer H_{j+1}. This argument continues until all nodes have joined a layer, which happens after at most $\ell = \lceil \log_{\frac{2+\varepsilon}{2}} n \rceil$ rounds. ◀

2.2 Step 2: Coloring the Graph using the Low Out-Degree Orientation

We now employ the above low out-degree (acyclic) orientation to compute an $O(\alpha^2)$-coloring, in $O(\log^* n)$ additional rounds. The algorithm is based on (iterative applications of) a single-round coloring reduction, similar to Linial's Algorithm [18, 19].

Linial's Coloring Algorithm

Linial's coloring algorithm is an $O(\log^* n)$-round deterministic distributed algorithm that computes an $O(\Delta^2)$-coloring of the input graph, where Δ is the largest degree in the graph. In each round, a k-coloring is transformed to a k'-coloring, such that $k' = O(\Delta^2 \log_\Delta k)$. This is done by letting each node compute a set that is not a subset of the union of the sets of its neighbors. Then, it picks an arbitrary color from this set that is not in the union of its neighbors' sets. The existence of such a set relies on Lemma 3. The coloring is produced by iteratively applying the single-round color reduction. We start with the initial numbering of the vertices as a n-coloring. In a single round, we compute an $O(\Delta^2 \log_\Delta n)$-coloring. With another single-round color reduction, we get an $O(\Delta^2 \cdot (\log_\Delta \Delta + \log_\Delta \log_\Delta n))$ coloring. After $O(\log^* n)$ iterations, we end up with an $O(\Delta^2)$-coloring. The single-round reduction technique relies on the following lemma.

▶ **Lemma 3** (Linial [18, 19])**.** *For any k and Δ, there exists a Δ-cover free family of size k on a ground-set of size $k' = O(\Delta^2 \log_\Delta k)$ i.e., a family of sets $S_1, S_2, ..., S_k \in \{1, 2, ..., k'\}$ such that there is no set in the family that is a subset of the union of Δ other sets.*

Applying Linial's Algorithm to Low Out-Degree Graphs

Now, the second step of the $O(\alpha^2)$-coloring algorithm of Barenboim and Elkin [3, 5] is running a variation of Linial's algorithm where each node considers only the colors of its out-neighbors. In particular, each node computes a set that is not a subset of the union of the sets of its out-neighbors. Then, it picks an arbitrary color from this set that is not in the union of its out-neighbors' sets. This produces a proper coloring of the graph. Similar to Linial's algorithm, after $O(\log^* n)$ rounds, the number of colors is $O(\alpha^2)$.

3 Coloring for High-Arboricity Graphs

In this section, we present an $O(\log n)$-round randomized distributed algorithm that, with high probability computes, a $((2+\varepsilon)\cdot\alpha + O(\log n \cdot \log\log n))$-coloring of a graph G with arboricity $a = \Omega(\log n)$, for any desirably small constant $0 < \varepsilon \leq 1$.

Algorithm Outline. Our algorithm consists of two steps.

- In the first step, we perform an $O(\log n)$-round partial coloring that uses $(2+\frac{2}{3}\varepsilon)\cdot\alpha$ colors, in a manner that the remaining graph – i.e., the graph induced by the nodes that remain uncolored – has arboricity at most $\frac{\varepsilon}{144}\alpha$, with high probability.
- In the second step, we partially color the remaining graph of arboricity at most $\frac{\varepsilon}{144}\alpha$, in $O(\log n)$ rounds, using at most $\frac{\varepsilon}{3}\alpha$ new colors. This is done such that at the end of the second step, the subgraph induced by the uncolored nodes has arboricity at most $O(\log n)$, with high probability.

Overall, our algorithm runs in $O(\log n)$ rounds and uses $(2+\varepsilon)\cdot\alpha$ colors. Once we are done with this 2-step partial coloring, on the remaining graph, we apply the coloring algorithm of Lemma 8, which we present later in Section 4. This algorithm uses $O(\log n \cdot \log\log n)$ new colors to color the remaining uncolored nodes, in $O(\log n)$ rounds. Hence, overall, we obtain a proper $((2+\varepsilon)\cdot\alpha + O(\log n \cdot \log\log n))$-coloring of the whole graph, in $O(\log n)$ rounds, with high probability. If we omit the first step and apply directly the second step of the algorithm, an $O(\alpha)$ partial coloring is produced in $O(\log n)$ rounds. Overall, this would produce a proper $O(\alpha)$-coloring of the the whole graph, in $O(\log n)$ rounds, with high probability.

We note that if the input graph G has arboricity $\alpha \geq \log^2 n$, once we reach a remaining graph of arboricity $O(\log n)$, we can wrap up using a much simpler algorithm: we can color the remaining graph by applying the variation of Linial's algorithm explained in Section 2.2, which uses $O(\log^2 n)$ extra colors and colors all the remaining nodes in $O(\log^* n)$ extra rounds. Hence, in total, we would end up with a $((2+\varepsilon)\cdot\alpha + O(\log^2 n))$-coloring in $O(\log n)$ rounds.

3.1 Step 1: A First Partial Coloring of the Graph

Let $G = (V, E)$ be a graph with arboricity $\alpha = \Omega(\log n)$. In this section, we present an $O(\log n)$-round randomized distributed algorithm that partially colors G, using $(2+\frac{2\varepsilon}{3})\cdot\alpha$ colors, for a small constant $0 < \varepsilon \leq 1$, such that the remaining graph i.e., the graph induced by the remaining uncolored nodes, has arboricity at most $\frac{\varepsilon}{144}\alpha$. Next, for simplifying the notation, we use $\epsilon = \frac{\varepsilon}{3}$.

A first preparation step of the algorithm is to compute in $O(\log n)$ rounds an H-partition with degree $d \leq (2+\epsilon)\cdot\alpha$ and size $\ell = \lceil \frac{1}{\epsilon}\log n \rceil$, together with an acyclic orientation of the edges, such that the maximum out-degree is at most $d \leq (2+\epsilon)\cdot\alpha$. Then, it partially colors layers $H_1, H_2, ..., H_\ell$ gradually, starting from layer H_ℓ and proceeds backwards, ending with the first layer H_1. Each node receives a palette of size $(2+2\epsilon)\cdot\alpha$ and when we color layer H_j, $1 \leq j \leq \ell$, each (uncolored) node $v \in H_j$ performs the following algorithm.

First Random Partial Coloring Algorithm, run by each node $v \in H_j$:
In iteration $i \in \{1, 2, ..., \lceil \frac{1+\epsilon}{\epsilon} \rceil \cdot \log \frac{300}{\epsilon}\}$,

- Node v selects one random color x among colors $\{1, 2, ..., (2+2\epsilon)\cdot\alpha\}$.
- Node v sends the selected color x to its neighbors, and receives their selected colors.

- If no out-neighbor has selected x in this round, or picked x as its permanent color in the previous rounds, node v gets colored permanently with x, and informs its neighbors.

▶ **Lemma 4.** *After partially coloring the graph in $O(\log n)$ rounds, the remaining graph i.e., the graph induced by the uncolored nodes, has out-degree at most $\frac{\epsilon}{112}d$, with high probability.*

Proof. First, we discuss the time complexity of the algorithm. We have $\lceil\frac{1+\epsilon}{\epsilon}\rceil \cdot \log\frac{300}{\epsilon}$ iterations per layer of the H-partition and the H-partition has $\ell = \lceil\frac{1}{\epsilon}\log n\rceil$ total layers. Hence, the whole algorithm has round complexity $O(\log n)$.

We now argue that once the algorithm is completed, with high probability, the remaining graph has arboricity at most $\frac{\epsilon}{112}d$. Consider an arbitrary layer H_j, $1 \leq j \leq \ell$ of the H-partition. A node $v \in H_j$ has at most $d \leq (2+\epsilon)\cdot\alpha$ neighbors in the graph induced by layers $\cup_{y=j}^{\ell} H_y$. In each iteration i, each permanently colored out-neighbor of v, blocks at most one color from v's palette. Each out-neighbor that is in the same layer H_j and remains uncolored in iteration i, blocks at most 1 color from v's palette in iteration i. This implies that in any iteration i, v has at least $\epsilon\cdot\alpha$ colors that are not blocked by its out-neighbors. Therefore, the probability that v gets permanently colored with a color x in iteration i is at least $\frac{\epsilon\cdot\alpha}{(2+2\epsilon)\cdot\alpha}$. Moreover, this holds independently of the events of other nodes being colored.

In total, after $\lceil\frac{1+\epsilon}{\epsilon}\rceil \cdot \log\frac{300}{\epsilon}$ iterations we get that, independently of the events of other nodes being colored,

$$Pr[\text{v is not colored}] \leq (1-\frac{\epsilon\cdot\alpha}{(2+2\epsilon)\cdot\alpha})^{\lceil\frac{1+\epsilon}{\epsilon}\rceil\cdot\log\frac{300}{\epsilon}} \leq (\frac{1}{4})^{\frac{1}{2}\log\frac{300}{\epsilon}} \leq \frac{\epsilon}{300}.$$

After applying the partial coloring in layers $H_1, H_2, .., H_\ell$, each node remains uncolored with probability at most $\frac{\epsilon}{300}$.

At this point, the coloring process of the algorithm is completed. We now upper bound the arboricity of the remaining graph i.e., the graph induced by the uncolored nodes after applying the algorithm. Consider a node v that remains uncolored and let X be a random variable that represents the number of v's uncolored out-neighbors. Then,

$$E[X] \leq d\cdot\frac{\epsilon}{300}.$$

So long as the expected out-degree is $\Omega(\log n)$, we can apply the Chernoff bound and conclude that

$$Pr[X \geq d\cdot\frac{\epsilon}{112}] \leq \frac{1}{n^{10}}.$$

Hence, the remaining graph is an H-partition with degree $d \leq \frac{\epsilon}{112}(2+\epsilon)\alpha \leq \frac{\varepsilon}{336}(2+\frac{\varepsilon}{3})\alpha \leq \frac{\varepsilon}{144}\alpha$ and size $\ell = \lceil\frac{3}{\varepsilon}\log n\rceil$ and is oriented such that the out-degree of each remaining node is at most $d \leq \frac{\varepsilon}{144}\alpha$, with high probability. ◀

3.2 Step 2: A Second Partial Coloring of the Remaining Graph

Once the first step of the algorithm is completed, the remaining graph is an H-partition with degree $d \leq \frac{\varepsilon}{144}\alpha$ and size $\ell = \lceil\frac{3}{\epsilon}\log n\rceil$ and is oriented such that the out-degree is at most $d \leq \frac{\varepsilon}{144}\alpha$, with high probability.

In this section, we present an $O(\log n)$ randomized distributed algorithm that partially color this remaining graph using $48d \leq \frac{\varepsilon}{3}\alpha$ colors, in a manner that once the algorithm is completed, the graph induced by the remaining uncolored nodes has arboricity at most $O(\log n)$, with high probability.

▶ **Lemma 5.** *Given an H-partition with degree $d = \Omega(\log n)$ and size $O(\log n)$, there is an $O(\log n)$ randomized distributed algorithm that partially colors the graph using $48d$ colors, in a manner that the remaining graph has arboricity at most $O(\log n)$, with high probability.*

Proof. The algorithm consists of $\log^* n$ phases. In each phase i, for $i \in \{0, 1, ..., \log^* n\}$, we perform a partial coloring of the remaining graph as follows. The input of phase i is an H-partition of the remaining graph with degree $d_i \leq \frac{d}{i2}$ and size $O(\frac{\log n}{2^i})$. Here, the tetration ${}^{y}x$ expresses $x^{x^{\cdot^{\cdot^{x}}}}$, with y copies of x. In each phase i, we apply the $O(\frac{\log n}{2^i})$-round randomized distributed algorithm of Lemma 6, which we discuss later on this section. In phase i, we use $2Q_i = 2 \cdot \frac{12d}{2^i}$ colors and we partially color the graph such that at the end of phase i, the remaining graph is an H-partition with degree $d_{i+1} \leq \frac{d}{(i+1)2}$ and size $O(\frac{\log n}{2^{i+1}})$, with high probability. This is the input for the next phase.

After $\log^* n$ phases, the remaining nodes have out-degree at most $O(\log n)$, with high probability. Furthermore, the total number of rounds of the process is $\sum_{i=0}^{\log^* n} \frac{O(\log n)}{2^i} = O(\log n)$ and the total number of colors that it uses is $\sum_{i=0}^{\log^* n} 2Q_i \leq 48d$. ◀

The Coloring Algorithm for a Single Phase. For each phase i, we start with an H-partition of the remaining graph with degree $d_i \leq \frac{d}{i2}$ and size $O(\frac{\log n}{2^i})$. In the coloring part of this phase, we color some nodes in a manner that, among the nodes that remain uncolored, each node has out-degree at most $\frac{d}{(i+2)1.98 \cdot 20} \ll \frac{d}{i2}$.

The coloring process in phase i consists of two iterations, as follows: In each iteration, each remaining node receives a fresh palette of $Q_i = \frac{12d}{2^i}$ colors. We color the layers $H_1, H_2, .., H_\ell$ of the given H-partition gradually, starting from the last layer H_ℓ, and proceed backwards, ending with the first layer H_1. As we show next, one iteration is not enough to drop the maximum out-degree to the desired level. Repeating the algorithm for a second iteration, we end up with maximum out-degree at most $\frac{d}{(i+2)1.98 \cdot 20} \ll \frac{d}{i2}$, with high probability. We now focus on coloring an arbitrary layer H_j, $1 \leq j \leq \ell$. Each node v in layer H_j performs the following algorithm.

Single-Iteration of Second Partial Coloring Algorithm, run by each node $v \in H_j$

- Node v selects $f(i) = \frac{Q_i}{2d_i}$ colors at random from a new palette of Q_i colors.
- Node v sends the selected colors to its neighbors, and receives their selected colors.
- If there is a selected color x such that no out-neighbor has selected x in this round, or picked x as its permanent color in the previous rounds, node v gets colored permanently with x, and informs its neighbors.

▶ **Lemma 6.** *Given an H-partition with degree $d_i \leq \frac{d}{i2}$ and size $O(\frac{\log n}{2^i})$, there is an $O(\frac{\log n}{2^i})$-round randomized distributed algorithm that partially colors the graph with $2Q_i = 2 \cdot \frac{12d}{2^i}$ colors, such that in the same H-partition, with size $O(\frac{\log n}{2^i})$, the remaining graph has out-degree at most $\frac{d}{(i+2)1.98 \cdot 20}$, with high probability.*

Proof. First, we discuss the time complexity of the algorithm. We have two iterations, and each iteration takes $\ell = O(\frac{\log n}{2^i})$ rounds, one round per layer of the H-partition. Hence, the whole algorithm of this phase has round complexity $\ell = O(\frac{\log n}{2^i})$.

We now argue that at the end of the phase, with high probability, in the remaining graph induced by the uncolored nodes each node has out-degree at most $\frac{d}{(i+2)1.98 \cdot 20}$. We do the analysis of the two iterations separately, though they are similar.

Consider the first iteration of phase i and an arbitrary layer H_j, $1 \leq j \leq \ell$. A node $v \in H_j$ has at most d_i out-neighbors in the graph induced by layers $\cup_{y=j}^{\ell} H_y$. Each permanently colored out-neighbor of v blocks at most one color from v's palette. Each out-neighbor that belongs to the same layer H_j, blocks at most $f(i)$ colors from v's palette.

Thus, there are at most $f(i) \cdot d_i$ colors that are blocked by v's out-neighbors, which implies that v has at least $Q_i - f(i) \cdot d_i = \frac{Q_i}{2}$ colors that are not blocked, when we select random colors for v. Therefore, the probability that v gets permanently colored with a color x that it selects is at least $1/2$. Moreover, this holds independently of the events of other nodes being colored. In total, since v selects $f(i) = \frac{Q_i}{2d_i}$ colors independently, we get that independently of the events of other nodes being colored:

$$Pr[\text{v is not colored}] \leq 2^{-f(i)} = 2^{-\frac{i_{2 \cdot 6}}{2^i}}.$$

After applying the 1-round coloring in layers $H_1, H_2, .., H_\ell$, each node remains uncolored with probability at most $2^{-f(i)}$.

At this point, the coloring process of the first iteration is completed. We now upper bound the maximum out-degree of the remaining graph. Consider a node v that remains uncolored and let X be a random variable that represents the number of v's uncolored out-neighbors. Then,

$$E[X] \leq d_i \cdot 2^{-f(i)} \leq \frac{d}{i_2} \cdot 2^{-\frac{i_{2 \cdot 6}}{2^i}}.$$

As long as the new expected out-degree is $\Omega(\log n)$, we can apply the Chernoff bound and conclude that

$$Pr[X \geq \frac{d}{(i+1)1.99 \cdot 20}] \leq Pr[X \geq 3\frac{d}{i_{2 \cdot 64}} 2^{-\frac{i_2}{2^i}}] \leq \frac{1}{n^{10}}.$$

We now discuss the decrease in the out-degrees during the second iteration. At the beginning of the second iteration, in the remaining graph, each (uncolored) node has at most $\frac{d}{(i+1)1.99 \cdot 20}$ out-neighbors, with high probability. Similarly to the first iteration, each remaining node receives a fresh palette of size Q_i. Again, applying the same process, after we color layers $H_1, H_2, .., H_\ell$ in the second iteration, each node remains uncolored with probability at most $2^{-f(i)}$. With a similar analysis, we conclude that in the graph induced by nodes that remain uncolored at the end of the second iteration, each node has out-degree at most $\frac{d}{(i+2)1.98 \cdot 20}$, with high probability. ◀

Re-computing the H-partition. At this point, we are done with the coloring of phase i. As a preparation step for phase $i+1$, we compute a new H-partition of the graph induced by the uncolored nodes. The new H-partition has degree $d_{i+1} \leq \frac{d}{(i+1)_2}$ and size $O(\frac{\log n}{2^{i+1}})$.

▶ **Lemma 7.** *Given an H-partition with degree at most $\frac{d}{(i+2)1.98 \cdot 20}$ and size $O(\frac{\log n}{2^i})$, there is an $O(\frac{\log n}{2^{i+1}})$-round deterministic distributed algorithm that computes an H-partition with degree at most $\frac{d}{(i+1)_2}$ and size $O(\frac{\log n}{2^{i+1}})$.*

Proof. We set the parameter $\varepsilon > 0$ of the H-partition of Lemma 2, to a value such that the degree of the H-partition is $(2+\varepsilon)\frac{d}{(i+2)1.98 \cdot 20} \leq \frac{d}{(i+1)_2}$ and the size of the H-partition is $\ell = \frac{\log n}{\log \varepsilon} \leq \frac{\log n}{2^{i+1}}$. In particular, we set $\varepsilon = 16\frac{(i+2)}{(i+1)_2}\frac{1.98}{2}$, and compute an H-partition with degree $d_{i+1} \leq \frac{d}{(i+1)_2}$ and size $\ell \leq \frac{\log n}{2^{i+1}}$. The round complexity of recomputing the H-partition is at most $O(\frac{\log n}{2^{i+1}})$, as explained in Lemma 2. ◀

4 Coloring for Low-Arboricity Graphs

In this section, we present two randomized distributed algorithms that on any n-node graph with arboricity α, with high probability, compute respectively

- an $O(\alpha \log \alpha)$-coloring in $O(\log n)$ rounds, and
- an $O(\alpha)$-coloring in $O(\log n \cdot \log \alpha)$ rounds.

In particular, we prove the following two lemmas in Section 4.1 and Section 4.2, respectively.

▶ **Lemma 8.** *There is an $O(\log n)$-round randomized distributed algorithm that partially colors any n-node graph with arboricity α, using $O(\alpha \log \alpha)$ colors, in a manner that the remaining graph has no path longer than $O(\log n)$, with high probability.*

▶ **Lemma 9.** *There is an $O(\log n \cdot \log \alpha)$-round randomized distributed algorithm that partially colors any n-node graph with arboricity α, using $(2+\varepsilon) \cdot \alpha$ colors, for a constant $0 < \varepsilon \leq 1$, in a manner that the remaining graph has no path longer than $O(\log n)$, with high probability.*

After partially coloring the graph with the algorithms of Lemma 8 or Lemma 9, we apply the $O(\log n)$-round deterministic distributed algorithm of Lemma 13, to color the remaining graph using $O(\alpha)$ extra colors.

We note that the algorithms we present in this section are more interesting for coloring graphs with arboricity at most $O(\log n)$, since for graphs with larger arboricity, we can apply the algorithm of Section 3 to obtain a $((2+\varepsilon) \cdot \alpha + O(\log n \cdot \log \log n))$-coloring in $O(\log n)$ rounds.

4.1 A Randomized $O(\alpha \log \alpha)$ Partial Coloring in $O(\log n)$ rounds

Let G be a n-node graph with arboricity α. In this section, we provide an $O(\log n)$-round randomized distributed algorithm that partially colors the graph with $O(\alpha \log \alpha)$ colors, in a manner that the remaining graph has no path longer than $O(\log n)$, with high probability.

A first preparation step of the algorithm is to compute in $O(\log n)$ rounds an H-partition with degree $d \leq 3\alpha$ and size $O(\log n)$, together with an acyclic orientation of the edges, such that the maximum out-degree is at most $d \leq 3\alpha$.

The algorithm colors layers $H_1, H_2, ..., H_\ell$ gradually, starting from layer H_ℓ and proceeds backwards, ending with the first layer H_1. Initially, each node receives a palette of $d \log d$ colors. When layer H_j, $1 \leq j \leq \ell$ is colored, each remaining node $v \in H_j$ performs the following algorithm.

Low-Arb Coloring Algorithm, run by each node $v \in H_j$
In iteration $i \in \{1, 2, 3, 4\}$:

- Node v selects $\frac{\log d}{2}$ random colors among $d \log d$ colors.
- Node v sends the selected colors to the neighbors, and receives their selected colors.
- If there is a selected color x such that no out-neighbor has selected x in this round, or picked x as its permanent color in the previous rounds, node v gets colored permanently with x, and informs its neighbors.

▶ **Lemma 10.** *After partially coloring the graph in $O(\log n)$ rounds, each node $v \in V$ remains uncolored with probability at most d^{-2}. Furthermore, this holds independently of the events of other nodes being colored.*

Proof. First, we discuss the time complexity of the algorithm. The H-partition has $O(\log n)$ layers and we have 4 iterations per layer of the H-partition. Hence, the whole algorithm has round complexity $O(\log n)$.

We now argue that once the algorithm is completed, in the remaining graph, each node $v \in V$ remains uncolored with probability at most d^{-2}, independently of the events of other nodes being colored in the graph.

Consider an arbitrary layer H_j, $1 \leq j \leq \ell$ of the H-partition. A node $v \in H_j$ has at most d out-neighbors in the graph induced by layers $\cup_{y=j}^{\ell} H_y$. In each iteration i, each permanently colored out-neighbor of v blocks at most one color from v's palette. Each out-neighbor that is in the same layer H_j and remains uncolored in iteration i, blocks at most $\frac{\log d}{2}$ colors from v's palette.

Thus, v has at least $\frac{d \log d}{2}$ colors that are not blocked by its out-neighbors, when we select random colors for v. Therefore, the probability that v gets permanently colored with a color x that it selects in iteration i is at least $1/2$. Moreover, this holds independently of the events of other nodes being colored. This implies that in each iteration, independently of the events of other nodes being colored, we have

$$Pr[\text{v is not colored}] \leq 2^{-\frac{\log d}{2}} = 1/\sqrt{d}.$$

In total, after 4 iterations we get that, independently of the events of other nodes being colored, we have

$$Pr[\text{v is not colored}] \leq (1/\sqrt{d})^4 = d^{-2}.$$ ◀

Next, we prove that in the remaining graph, there exists no path longer than $O(\log n)$, with high probability. This allows us to color the remaining graph deterministically in $O(\log n)$ rounds, using $d+1$ extra colors, as we explain in Lemma 13.

▶ **Lemma 11.** *The remaining graph has no directed path longer than $O(\log n)$, w.h.p.*

Proof. There are at most $n \cdot d^{\log n}$ different ways to select a path of length $\log n$. For each such path, the probability that all of its nodes stay is at most $d^{-2\log n}$. By a union bound over all such paths, we conclude that with probability $1 - n \cdot d^{\log n} \cdot d^{-2\log n} \geq 1 - n^{-10}$, no such path exists. ◀

4.2 A Randomized $O(\alpha)$ Partial Coloring in $O(\log n \cdot \log \alpha)$ Rounds

In this section, we present an $O(\log n \cdot \log \alpha)$-round randomized distributed algorithm that colors a graph G with arboricity α, using $(2+\varepsilon)$ colors, for a small constant $0 < \varepsilon \leq 1$, in a manner that the remaining graph has no path longer than $O(\log n)$, with high probability.

The algorithm is similar to the randomized distributed algorithm of Section 4.1. More specifically, it first computes an H-partition with degree $d \leq (2+\frac{\varepsilon}{2}) \cdot \alpha$ and size $\ell = \lceil \frac{1}{\varepsilon} \log n \rceil$. Each node receives a palette of size $(2+\varepsilon) \cdot \alpha$ and when we color layer H_j, $1 \leq j \leq \ell$, each (uncolored) node performs the following algorithm.

Tradeoff-Low-Arb Coloring Algorithm, run by each node $v \in H_j$:
In iteration $i \in \{1, 2, ..., \lceil \frac{2\cdot(2+\varepsilon)}{\varepsilon} \rceil \cdot \log d\}$,
- Node v selects one random color x among $(2+\varepsilon) \cdot \alpha$ colors.
- Node v sends the selected color x to its neighbors, and receives their selected colors.

- If no out-neighbor has selected x in this round, or picked x as its permanent color in the previous rounds, node v gets colored permanently with x, and informs its neighbors.

▶ **Lemma 12.** *After partially coloring the graph in $O(\log n \cdot \log \alpha)$ rounds, each node $v \in V$ remains uncolored with probability at most d^{-2}. Furthermore, this holds independently of the events of other nodes being colored.*

Proof. First, we discuss the time complexity of the algorithm. The H-partition has $O(\log n)$ layers and we have $\lceil \frac{2\cdot(2+\varepsilon)}{\varepsilon} \rceil \cdot \log d = O(\log \alpha)$ iterations per layer of the H-partition. Hence, the whole algorithm has round complexity $O(\log n \cdot \log \alpha)$.

We now argue that once the algorithm is completed, in the remaining graph, each node $v \in V$ remains uncolored with probability at most d^{-2}, independently of the events of other nodes being colored in the graph.

Consider an arbitrary layer H_j, $1 \leq j \leq \ell$ of the H-partition. A node $v \in H_j$ has at most $d \leq (2 + \frac{\varepsilon}{2}) \cdot \alpha$ neighbors in the graph induced by layers $\cup_{y=j}^{\ell} H_y$. In any iteration i, each permanently colored out-neighbor of v, blocks at most one color from v's palette. Each out-neighbor that is in the same layer H_j and remains uncolored in iteration i, blocks at most one color from v's palette. Thus, in any iteration i, node v has at least $\frac{\varepsilon}{2}\alpha$ colors that are not blocked by its out-neighbors. Therefore, the probability that v gets permanently colored with a color x in iteration i is at least $\frac{\varepsilon \cdot \alpha}{2(2+\varepsilon)\cdot\alpha}$. Moreover, this holds independently of the events of other nodes being colored.

In total, after $\lceil \frac{2\cdot(2+\varepsilon)}{\varepsilon} \rceil \cdot \log d$ iterations, we get that (independently of the events of other nodes being colored), we have

$$Pr[\text{v is not colored}] \leq (1 - \frac{\varepsilon \cdot \alpha}{2(2+\varepsilon)\cdot\alpha})^{\lceil \frac{2\cdot(2+\varepsilon)}{\varepsilon} \rceil \log d} \leq (\frac{1}{4})^{\log d} \leq d^{-2}.$$ ◀

At this point, we apply Lemma 11 to conclude that in the remaining graph there is no path longer than $O(\log n)$, with high probability. Then, we apply the $O(\log n)$-round deterministic algorithm of Lemma 13, to color the remaining graph with $d+1$ extra colors.

4.3 Deterministic Coloring

After we partially color the input graph G with either of the algorithms of Section 4.1 and Section 4.2, in the remaining graph there is no path longer than $O(\log n)$, with high probability.

In this section, we color deterministically the remaining graph as follows. Each remaining (uncolored) node receives $d+1$ new colors and performs the following algorithm.

Low-Arb Deterministic Coloring Algorithm, run by each uncolored node v:

- Node v waits for all its remaining out-neighbors to be colored and removes their colors from its palette.
- It gets permanently colored with one remaining color x, and informs its neighbors.

▶ **Lemma 13.** *After $O(\log n)$ rounds, every node is colored, with high probability.*

Proof. Consider a remaining (uncolored) node v that runs the above algorithm. Since it has at most d remaining out-neighbors, there is always at least one available color to select

the moment that we color node v. Furthermore, by Lemma 11, there is no path longer than $O(\log n)$ in the remaining graph, with high probability; this implies that with high probability, v does not wait more than $O(\log n)$ rounds until it gets permanently colored. ◀

References

1. Baruch Awerbuch, M Luby, AV Goldberg, and Serge A Plotkin. Network decomposition and locality in distributed computation. In *Foundations of Computer Science, 1989., 30th Annual Symposium on*, pages 364–369, 1989.
2. Leonid Barenboim. Deterministic (Δ+ 1)-coloring in sublinear (in Δ) time in static, dynamic, and faulty networks. *Journal of the ACM (JACM)*, page 47, 2016.
3. Leonid Barenboim and Michael Elkin. Sublogarithmic distributed mis algorithm for sparse graphs using nash-williams decomposition. In *Proceedings of the Twenty-seventh ACM Symposium on Principles of Distributed Computing*, PODC'08, pages 25–34, 2008.
4. Leonid Barenboim and Michael Elkin. Distributed (Δ+ 1)-coloring in linear (in Δ) time. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 111–120, 2009.
5. Leonid Barenboim and Michael Elkin. Sublogarithmic distributed MIS algorithm for sparse graphs using nash-williams decomposition. *Distributed Computing*, 22(5-6):363–379, 2010.
6. Leonid Barenboim and Michael Elkin. Deterministic distributed vertex coloring in polylogarithmic time. *Journal of the ACM (JACM)*, page 23, 2011.
7. Leonid Barenboim and Michael Elkin. Distributed graph coloring: Fundamentals and recent developments. *Synthesis Lectures on Distributed Computing Theory*, 4(1):1–171, 2013.
8. Leonid Barenboim, Michael Elkin, and Fabian Kuhn. Distributed (Δ+1)-coloring in linear (in Δ) time. *SIAM Journal on Computing*, 43(1):72–95, 2014.
9. Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The locality of distributed symmetry breaking. In *Foundations of Computer Science (FOCS) 2012*, pages 321–330, 2012.
10. Béla Bollobás. Chromatic number, girth and maximal degree. *Discrete Mathematics*, 24(3):311–314, 1978.
11. Kai-Min Chung, Seth Pettie, and Hsin-Hao Su. Distributed algorithms for the lovász local lemma and graph coloring. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, pages 134–143, 2014.
12. Richard Cole and Uzi Vishkin. Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 206–219. ACM, 1986.
13. Pierre Fraigniaud, Marc Heinrich, and Adrian Kosowski. Local conflict coloring. In *Foundations of Computer Science (FOCS), 2016 IEEE 57th Annual Symposium on*, pages 625–634. IEEE, 2016.
14. David G Harris, Johannes Schneider, and Hsin-Hao Su. Distributed (Δ+ 1)-coloring in sublogarithmic rounds. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing*, pages 465–478, 2016.
15. Öjvind Johansson. Simple distributed Δ+ 1-coloring of graphs. *Information Processing Letters*, 70(5):229–232, 1999.
16. Kishore Kothapalli and Sriram Pemmaraju. Distributed graph coloring in a few rounds. In *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC'11, pages 31–40, 2011.
17. Fabian Kuhn. Weak graph colorings: Distributed algorithms and applications. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, pages 138–144, 2009.

18 Nathan Linial. Distributive graph algorithms global solutions from local data. In *Proc. of the Symp. on Found. of Comp. Sci. (FOCS)*, pages 331–335. IEEE, 1987.

19 Nathan Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992.

20 Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM journal on computing*, 15(4):1036–1053, 1986.

21 CSJA Nash-Williams. Decomposition of finite graphs into forests. *Journal of the London Mathematical Society*, 1(1):12–12, 1964.

22 Alessandro Panconesi and Romeo Rizzi. Some simple distributed algorithms for sparse networks. *Distributed computing*, 14(2):97–100, 2001.

23 Alessandro Panconesi and Aravind Srinivasan. Improved distributed algorithms for coloring and network decomposition problems. In *Proc. of the Symp. on Theory of Comp. (STOC)*, pages 581–592. ACM, 1992.

24 David Peleg. *Distributed Computing: A Locality-sensitive Approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

25 Seth Pettie and Hsin-Hao Su. Distributed coloring algorithms for triangle-free graphs. *Information and Computation*, 243:263–280, 2015.

26 Johannes Schneider and Roger Wattenhofer. A new technique for distributed symmetry breaking. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 257–266, 2010.

27 Johannes Schneider and Roger Wattenhofer. Distributed coloring depending on the chromatic number or the neighborhood growth. In *International Colloquium on Structural Information and Communication Complexity*, pages 246–257. Springer, 2011.

28 Márió Szegedy and Sundar Vishwanathan. Locality based graph coloring. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, pages 201–207, 1993.

Near-Optimal Distributed DFS in Planar Graphs

Mohsen Ghaffari[1] and Merav P. Parter[2]

1 **ETH Zürich, Switzerland**
`ghaffari@inf.ethz.ch`
2 **CSAIL, MIT, Cambridge, USA**
`parter@mit.edu`

Abstract

We present a randomized distributed algorithm that computes a Depth-First Search (DFS) tree in $\widetilde{O}(D)$ rounds, in any planar network $G = (V, E)$ with diameter D, with high probability. This is the first sublinear-time distributed DFS algorithm, improving on a three decades-old $O(n)$ algorithm of Awerbuch (1985), which remains the best known for general graphs. Furthermore, this $\widetilde{O}(D)$ round complexity is nearly-optimal as $\Omega(D)$ is a trivial lower bound.

A key technical ingredient in our results is the development of a distributed method for (recursively) computing a *separator path*, which is a path whose removal from the graph leaves connected components that are all a constant factor smaller. We believe that the general method we develop for computing path separators recursively might be of broader interest, and may provide the first step towards solving many other problems.

1998 ACM Subject Classification G.2.2 Graph Algorithms

Keywords and phrases DFS, planar graphs, CONGEST, separator

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.21

1 Introduction and Related Work

Depth First Search (DFS) is "*one of the most versatile sequential algorithm techniques known for solving graph problems*" [38]. Along with its cousin BFS, these two have a long history: DFS dates back to the 19th century [10], and BFS dates back to the 1950s [40]. Both were first used for solving different kinds of mazes, but are nowadays among basic building blocks in graph algorithms, covered in elementary courses, and with a wide range of applications.

In the centralized setting, computing BFS and DFS are straightforward. However, in the distributed setting, there is a stark contrast, and DFS turns out to be much harder. Let us first recall the definition of the distributed model.

Throughout, we use a standard message passing model of distributed computing called CONGEST [36]. The network is abstracted as an n-node graph $G = (V, E)$, with one processor on each network node. Initially, these processors do not know the graph. They solve the given graph problems via communicating with their neighbors. Communications happen in synchronous rounds. Per round, each processor can send one $O(\log n)$-bit message to each of its neighboring processors.

Distributedly computing both BFS and DFS need $\Omega(D)$ rounds, in graphs of diameter D. BFS can be computed easily in $O(D)$ rounds, in any graph with diameter D. In contrast, the best known distributed algorithm for DFS takes $O(n)$ rounds, regardless of how small diameter D is; see e.g., [36, Section 5.4] and [4]. We note that designing algorithms with complexity $o(n)$, when $D = o(n)$, and ideally close to $O(D)$, has become the target of essentially all the distributed graph algorithms for global optimization problems, since the pioneering work of


© Mohsen Ghaffari and Merav Parter;
licensed under Creative Commons License CC-BY

Garay, Kutten, and Peleg [13, 27] which gave an $O(D + n^{0.61})$ round algorithm for minimum spanning tree. See, e.g., [5, 7, 8, 12, 14, 15, 18, 19, 24, 25, 28, 29, 32, 33, 34].

Despite this, there has been no progress on the problem over the last three decades, and no sublinear-time distributed algorithm for DFS is known. This, and especially the lack of any sub-linear time distributed DFS, is certainly not for the lack of trying. It is widely understood that DFS is not easy to parallelize/decentralize; it has even been called "*inherently sequential*" [38] and "*a nightmare for parallel processing*" [31].

1.1 Our Contribution

In this paper, making a first step of progress on the distributed complexity of this classical problem, we present a randomized distributed algorithm that computes a DFS in $O(D \cdot \text{poly}\log n)$ rounds in planar graphs, with high probability. This time complexity is nearly optimal as it matches the trivial $\Omega(D)$ lower bound up to poly-logarithmic factors.

▶ **Theorem 1.** *There is a randomized distributed algorithm that computes a DFS in any n-node planar network with diameter D in $O(D \cdot \text{poly}\log n)$ rounds, with high probability.*

Turning to general graphs, we note that the parallel algorithms by Aggarwal and Anderson [1] and Goldberg, Plotkin and Vaidya [20] can be adapted to give an $\widetilde{O}(\sqrt{Dn}+n^{3/4})$-round DFS algorithm for graphs with diameters D. Therefore, a DFS can be computed in sublinear number of rounds for graphs with sublinear diameter $D = o(n)$. This simple corollary of [1] and [20] is shown in the full version. Improving the bound for general graphs remains an important open problem.

1.2 A High-Level Discussion of Our Method

Our method relies on *separator paths*. Generally, *separators* have been a key tool in working with planar graphs, starting with the seminal work of Lipton and Tarjan [30]. In a rough sense, separators are subgraphs whose removal from the graph leaves connected components that are all a constant factor smaller than the initial graph. Typically, one desires the separator to be small. However, unlike [30], we do not insist on a small separator, but instead it is crucial for us that the separator is a simple *path*. This allows us to use the separator path, with some iterations of massaging and modifications in the style of [1], as a part of a partial DFS. See Sec. 3 for the explanations. Now that this separator is put in the partial DFS, the left over graph is made of a number of connected components, each a constant factor smaller than the initial graph. Hence, we would have the hope to be able to solve each of these subproblems recursively, and moreover, to do that simultaneously for all the subproblems.

But two key issues remain: (1) How do we compute the separator path distributedly? This itself is the main technical contribution of our paper, and is explained in Sec. 4. But a crucial part of the challenge of that lies in the next point. (2) How do we recurse and most importantly, how do we compute the separator path throughout the recursions? Once we remove the first separator, the left over components are smaller in size, but they may have considerably larger diameter, even up to $\Theta(n)$. This large diameter can be a major obstacle for distributed algorithms. For instance, we cannot even assume that we can compute a BFS of each component. More generally, if we are to have a fast separator path algorithm, we cannot confine the algorithm for each subproblem to stay within the connected component of that subproblem. On the other hand, allowing the algorithm to use the other parts creates

the possibility of *congestion* as now many subproblems may need to use the same edge, perhaps many times each.

Our solution uses a number of novel algorithmic ideas. It would be hard to summarize these ideas out of context, and thus we refer the interested reader to the technical sections. One key tool from prior work, which is worth pointing out and makes our life significantly easier, is *low-congestion shortcucts* for planar graphs, developed by Ghaffari and Haeupler [17]. In a very rough sense, this tool opens the road for working on many disjoint potentially large-diameter subgraphs of the base graph G at the same time, and still enjoying the small diameter of the base graph G. Though, this is possible only in certain conditions and only for a very limited class of problems. We usually need quite some work to break our problems into modules that fit these conditions and classifications.

Separators have a wide range of applications, in centralized algorithms for planar graphs. Though, computing the separators distributedly and especially computing them recursively in the distributed setting when the remaining components have large diameter is highly non-trivial. We thus hope that the methods developed in this paper may open the road for recursive computation of separators in the distributed setting, and thus be a first step towards solving many other problems.

1.3 Related Work

Distributed Graph Algorithms in Sublinear Time. Over the past two decades, starting with the seminal work of Garay, Kutten, and Peleg [13, 27], there has a been a big body of work presenting sublinear-time distributed algorithms for various graph problems (for graphs with diameter $D = o(n)$, as otherwise that is impossible). See for instance[1] [5, 7, 8, 12, 14, 15, 18, 19, 24, 25, 28, 29, 32, 33, 34]. There are also lower bounds [6, 9, 37] which for instance show that in general graphs, computing minimum spaning trees requires $\tilde{\Omega}(D + \sqrt{n})$ rounds, hence ruling out the possibility of $\tilde{O}(D)$ round MST algorithms. A similar lower bound holds for many other problems, even when approximating, e.g., min-cut, shortest paths, min-cost connected dominated set etc. See [6]. By now, most of the classical graph problems are known to have sublinear-time distributed algorithms, at least when relaxing the problem to allow some approximation. A prominent exception is DFS!

Distributed Graph Algorithms in Planar Networks. Starting with the work of Ghaffari and Haeupler [16, 17], some attention has been paid to developing more efficient distributed algorithms for (global) network optimizations on planar or near-planar networks. This was in part motivated by trying to circumvent the aforementioned $\tilde{\Omega}(D + \sqrt{n})$ general-graph lower bound. Another motivation was also to bring in the vast array of the techniques and methodologies developed for efficient centralized algorithms for planar networks to the distributed domain.

In [17], the aformentioned lower bound was ruled out for planar networks by showing an $\tilde{O}(D)$ algorithm for MST in planar networks. A key tool in this MST algorithm was the concept of *low-congestion shortcuts*, which was introduced in [17]. An algorithm for computing this structure was also given in [17], which as one of its subroutines made use of the planar embedding algorithm of [16]. It was shown later by Haeupler, Izumi and Zuzic [21] that even without having the embedding, one can compute an approximate version of the low-congestion shortcuts, which is good enough for many applications. Furthermore, low

[1] This is merely a sample, and is by no means exhaustive.

congestion shortcuts were later extended to other special graph families such as those with fixed tree-width or genus [22].

Parallel DFS Algorithms. DFS has received vast attention in the parallel literature. It is known that computing the lexicographically-first DFS —where the smallest ID unvisited neighbor should be visited first — is P-complete [38], and is thus unlikely to admit an efficient parallel algorithm. This was the reason that DFS was deemed *"inherently sequential"* [38]. However, over the years, several sophisticated but efficient parallel algorithms were developed for DFS, which compute some depth first search tree (not necessarily the lexicographically-first one). We here review the related work on only undirected graphs. Smith [41] gave an $O(\log^3 n)$-time parallel DFS algorithm for planar graphs. Shannon [39] improved this to an $O(\log^2 n)$-time parallel DFS algorithm for planar graphs, while also using only linear number of processors. Anderson gave a $\tilde{O}(\sqrt{n})$-time [2] and then a $2^{O(\sqrt{\log n})}$-time [3] parallel DFS algorithm for general graphs. Aggarwal and Anderson [1] gave the first poly-logarithmic time parallel algorithm for DFS in general undirected graphs. Kao [26] gave the first deterministic NC algorithm for DFS in planar networks. Then Hagerup [23] gave an $O(\log n)$-time randomized parallel DFS algorithms for planar networks. Finding a deterministic NC algorithm for DFS in general graphs remains open, though a quasi-NC algorithm was given very recently in [11].

We note that our distributed DFS algorithm for planar graphs is quite different than the parallel DFS algorithms for planar graphs (e.g., [39, 41]), mainly because we do not compute a separator *cycle* distributedly. Our algorithm is morally closer to the methodology of Aggarwal and Anderson (for general graphs) [1] which can work with (collections of) separator *paths*.

2 Preliminaries

Basic Notions. Let $G = (V, E)$ be a simple undirected planar graph. Given a tree $T \subseteq G$ and a non-tree edge $e = (v, u) \notin T$, the cycle formed by e and the tree path connecting v to u is called the *fundamental cycle* defined by e. Let $\mathcal{F}(G) = \{F_1, \ldots, F_k\}$ be the faces of the planar graph G. Let $G' = (V', E')$ be the *dual graph* of G, defined by including one node $v'_i \in V'$ for each face $F_i \in \mathcal{F}$ and connecting two nodes $v'_i, v'_j \in V'$ if their corresponding faces share an edge[2]. We may interchange between the dual-nodes v'_i and the faces F_i. A superface $\mathcal{F}$ is a collection of faces whose boundary is a simple cycle.

Dual Tree and its Distributed Representation. Given a spanning tree T of G, we define its dual-tree in the dual-graph G' as follows: Let $\phi_F : \mathcal{F}(G) \to V'$ be the bijection between the faces of G and the dual-nodes of G'. The nodes of dual tree T' are the faces of G, and two dual-nodes v'_i and v'_j are connected iff the two faces $\phi_F^{-1}(v'_i)$ and $\phi_F^{-1}(v'_j)$ share an non-tree edge $e \notin T$. We define a bijection $\phi_E : E \setminus E(T) \to E(T')$ between the non-tree edges $G \setminus T$ and the dual-tree edges of T', where in the aforementioned example, we have $\phi_E(e) = \{v'_i, v'_j\}$.

In the distributed representation of this dual-tree, the leader $\ell(e')$ of a dual-edge $e' \in T'$ is the higher-ID endpoint of the edge $\phi_E^{-1}(e') = \{u, v\}$. The leader $\ell(v')$ of the dual-node v' is the node in the face $\phi_F(v')$ of maximum ID. The dual-tree is known in a distributed

[2] In-fact, the dual-graph is a multigraph as there might be many edges between two dual-nodes

manner where for every edge $e' \in T'$, its leader $\ell(e')$ knows that this edge belongs to T'. The nodes $v' \in V(T')$ and dual edges $e' \in E(T')$ will be simulated by their leader nodes $\ell(v')$ and $\ell(e')$ respectively.

Planar Embeddings. The geometric planar embedding of graph G is a drawing of G on a plane so that no two edges intersect. A combinatorial planar embedding of G determines the clockwise ordering of the edges of each node $v \in G$ around that node v such that all these orderings are consistent with a plane drawing (i.e., geometric planar embedding) of G. Ghaffari and Haeupler [16] gave a deterministic distributed algorithm that computes a combinatorial planar embedding in $O(D \min\{\log n, D\})$ rounds, where each node learns the clockwise order of its own edges.

Low-Congestion Shortcuts. In a subsequent paper [17], Ghaffari and Haeupler introduced the notion of *low-congestion shortcuts* which plays a key role in several algorithms for planar graphs (e.g., MST, min-cut). We will also make frequent use of this tool. The definition is as follows.

▶ **Definition 2** (α-congestion β-dilation shortcut)**.** Given a graph $G = (V, E)$ and a partition of V into disjoint subsets $S_1, \ldots, S_N \subseteq V$, each inducing a connected subgraph $G[S_i]$, we call a set of subgraphs $H_1, \ldots, H_N \subseteq G$, where H_i is a supergraph of $G[S_i]$, an α-congestion β-dilation shortcut if we have the following two properties:

1. For each i, the diameter of the subgraph H_i is at most β, and
2. for each edge $e \in E$, the number of subgraphs H_i containing e is at most α.

Ghaffari and Haeupler [17] proved that any partition of a D-diameter planar graph into disjoint subsets $S_1, \ldots, S_N \subseteq V$, each inducing a connected subgraph $G[S_i]$, admits an α-congestion β-dilation shortcut where $\alpha = O(D \log D)$ and $\beta = O(D \log D)$. They also gave a randomized distributed algorithm that computes such a shortcut in $\tilde{O}(D)$ rounds, with high probability. We will make black-box use of this result, frequently.

3 Outline of the Depth First Search Tree Construction

Towards proving Thm. 1, in this section, we explain the outline of our $\widetilde{O}(D)$-round algorithm for computing a *Depth-First Search* (DFS) tree. Detailed steps are explained in later sections.

We compute a DFS tree of a graph $G = (V, E)$ rooted in a given node $s \in V$. The algorithm is based on a *divide-and-conquer* style approach. A key technical ingredient is a *separator path* algorithm, which we use for *dividing* the problem into independent sub-problems of constant factor smaller size. We describe this separator algorithm in the next section. In this section, we explain how via recursive (black-box) applications of a separator path subroutine, we compute a DFS.

We note that our approach is inspired by an idea of Aggarwal and Anderson [1]. However, the overall method is quite different. On one hand, we have an easier case here because we need to deal with only a *single path* instead of a large collection of them, thanks to the nice structure of planar graphs. On the other hand, computing this single path, and especially being able to do it recursively, has its own challenges, as we discuss in the next section. We will have to deal with a number of difficulties that are unique to the distributed setting, as we will point out.

High-Level Outline. The high-level outline of the approach is as follows. The method is recursive. In each (independent) branch of the recursion, we have a connected induced subgraph $\mathcal{C} \subseteq G$ and a root $r \in \mathcal{C}$ and we need to compute a DFS of $\mathcal{C}$ rooted in r. In the beginning, we simply have $\mathcal{C} = G$ and $r = s$. Furthermore, in each step of recursion, we will assume that $\mathcal{C}$ is *biconnected*, that is, removing any single node $v \in \mathcal{C}$ from $\mathcal{C}$ leaves a connected subgraph $\mathcal{C} \setminus \{v\}$. Notice that this may not hold at the beginning, that is, G may have some cut-nodes. We will later discuss how to deal with cut nodes, by dividing the problem further into a number of independent subproblems, one for each biconnected component. For now, we assume that $\mathcal{C}$ is biconnected.

The Framework of One Recursion Level. We wish to compute a partial DFS $\mathcal{T}$ of $\mathcal{C}$ rooted at r such that each connected component of $\mathcal{C} \setminus \mathcal{T}$ has size at most $2|\mathcal{C}|/3$. This partial DFS $\mathcal{T}$ of $\mathcal{C}$ is such that it can be completed to a full DFS of $\mathcal{C}$ rooted at r. In particular, it has the following *validity* property: there are no two branches of $\mathcal{T}$ which are connected to each other via a path with all its internal nodes in $\mathcal{C} \setminus \mathcal{T}$. In other words, for each connected component $\mathcal{C}'_i$ of $\mathcal{C} \setminus \mathcal{T}$, all neighbors of $\mathcal{C}'_i$ in $\mathcal{T}$ are in one branch (i.e., rooted path) of $\mathcal{T}$. Once we compute this partial DFS $\mathcal{T}$, we can then recurse on each of those remaining connected components of $\mathcal{C} \setminus \mathcal{T}$, all in parallel. As the component size decreases by a $2/3$ factor per level of recursion, the recursion has depth $O(\log n)$.

The Procedure for One Recursion Level. Thus, the key is to grow a partial DFS $\mathcal{T}$ of $\mathcal{C}$ in $\widetilde{O}(D)$ rounds, in a way that each connected component of $\mathcal{C} \setminus \mathcal{T}$ has size at most $2|\mathcal{C}|/3$. We will do this in $\tilde{O}(D)$ rounds. For that purpose, we compute a *separator path* $\mathcal{P} \subseteq \mathcal{C}$ of $\mathcal{C}$. That is, each connected component of the graph $\mathcal{C} \setminus \mathcal{P}$ has size at most $2|\mathcal{C}|/3$. We explain this subroutine in the next section. For now, let us assume that such a path $\mathcal{P}$ is computed.

Let $\mathcal{Q}$ be a simple path that connects the root r to some node in $\mathcal{P}$ (and is otherwise disjoint from $\mathcal{P}$). Let v be the endpoint of $\mathcal{Q}$ in path $\mathcal{P}$ and suppose that $\mathcal{P} = u_1, u_2 \ldots, u_\ell, v, w_1, w_2, \ldots, w_{\ell'}$. Let $\mathcal{P}_1 = u_1, u_2 \ldots, u_\ell, v$ and $\mathcal{P}_2 = v, w_1, w_2, \ldots, w_{\ell'}$. We will use the longer one of $\mathcal{P}_1$ and $\mathcal{P}_2$ and append it to the path $\mathcal{Q}$ connecting the root r to v. Without loss of generality, suppose that the longer subpath is $\mathcal{P}_2$. We add the path $Q \cup \mathcal{P}_2$ as the first branch of the DFS $\mathcal{T}$. Moreover, we update the separator $\mathcal{P}$ to be the remaining part of the separator path, concretely $\mathcal{P}_1$ in the assumed case. We note that this is the idea that we borrow from Aggarwal and Anderson [1]. We have two important properties: (1) the new path $\mathcal{P}$ is still a separator of $\mathcal{C} \setminus \mathcal{T}$, and (2) the length of the new separator path $\mathcal{P}$ is at most half of the length of the previous separator.

Thanks to these two properties, we have the means to continue and exhaust the separator path in $O(\log n)$ repetitions. Each time we grow the partial DFS $\mathcal{T}$ further. Let us explain one step of this repetition. Fig. 1 illustrates an example for this step. We find the deepest node r' in the current partial DFS $\mathcal{T}$ rooted at r that is directly or indirectly connected to a node in the current separator $\mathcal{P}$, in the graph $\mathcal{C} \setminus \mathcal{T}$. Notice that this deepest node is unique, due to the validity of the current partial DFS, as all neighbors of the connected component of $\mathcal{C} \setminus \mathcal{T}$ containing $\mathcal{P}$ are in one branch of $\mathcal{T}$. We then find a path Q as above starting from r' and connecting to some node v in $\mathcal{P}$. This is done with the help of an $\widetilde{O}(D)$ round *minimum spanning tree* (MST) algorithm of Ghaffari and Haeupler [17], as we outline next.

In particular, let each node of $\mathcal{T}$ send its DFS depth to its neighbors in $\mathcal{C} \setminus \mathcal{T}$. Then, we run a connected component identification algorithm of [17] on the subgraph $\mathcal{C}' = \mathcal{C} \setminus \mathcal{T}$ of G. In identifying the connected components of the graph $\mathcal{C}' = \mathcal{C} \setminus \mathcal{T}$, the component leader is chosen according to having a $\mathcal{T}$-neighbor with deepest DFS depth. This finds the deepest

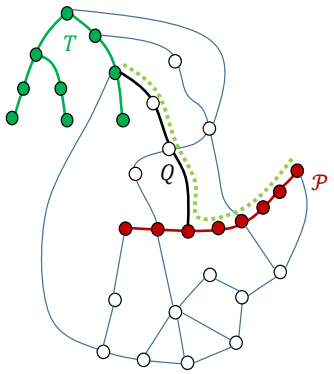


Figure 1 Growing the partial DFS tree. The green tree shows the current partial DFS $\mathcal{T}$, and the rest of the nodes are those of $\mathcal{C} \setminus \mathcal{T}$. The red path shows the current separator path $\mathcal{P}$. The black path is $\mathcal{Q}$, which connects the deepest point of $\mathcal{T}$ to some node in $\mathcal{P}$. The green dotted line indicated the path that will be added to the DFS, which is composed of $\mathcal{Q}$ and the longer half of $\mathcal{P}$ from the point of intersection with $\mathcal{Q}$. After this DFS growing, the leftover separator $\mathcal{P}$ will be the single edge at the left endpoint of $\mathcal{P}$. This is still a separator path of $\mathcal{C} \setminus \mathcal{T}$, for the new $\mathcal{T}$.

node r' of $\mathcal{T}$ that is in the connected component of $\mathcal{P}$ and thus has a path to $\mathcal{P}$. Furthermore, we can find a path $\mathcal{Q} \subset \mathcal{C} \setminus \mathcal{T}$ connecting r' to $\mathcal{P}$ in a similar manner.

Let us explain this step for finding path $\mathcal{Q}$, in $\tilde{O}(D)$ rounds. On the graph $\mathcal{C}'$, give an edge weight of 0 to each $\mathcal{P}$ edge and edge weight of 1 to each $\mathcal{C}' \setminus \mathcal{P}$ edge. Then, compute an MST of $\mathcal{C}'$ according to these weights using the algorithm of [17]. The unique path of weight-1 edges in the MST that connects node r' to a node $v \in \mathcal{P}$ is our desired path $\mathcal{Q}$. This path $\mathcal{Q}$ can be identified in $\widetilde{O}(D)$ rounds. One endpoint of it r' is clear by now. We first identify the other endpoint v, as follows: Discard all the zero-weight edges of the MST. Then, with another iteration of [17] on the subgraph edge-induced by weight-1 edges of the MST, we can identify the node $v \in \mathcal{P}$ who is the endpoint of the path $\mathcal{Q}$ connecting r' to $\mathcal{P}$. This is the only $\mathcal{P}$-node in the same component with r'. Now that we have the two endpoints r' and v of our path $\mathcal{Q}$, which is a part of the computed MST, we can fully mark this path $\mathcal{Q}$ in $\widetilde{O}(D)$ rounds, easily. We defer the details of that step to the full version, where we explain a routine for marking a tree-path connecting two nodes.

Now that we have found a path $\mathcal{Q}$ connecting the deepest possible node r' of $\mathcal{T}$ to a node $v \in \mathcal{P}$, we work as before. We break $\mathcal{P}$ at v, as depicted in Fig. 1, and append the longer half to Q, and then add the resulting path to the DFS $\mathcal{T}$, essentially hanging it from node $r' \in \mathcal{T}$. This is the dotted green path in Fig. 1. One can see that, as we chose v to be the deepest $\mathcal{T}$-node with a connection to $\mathcal{P}$, the resulting new tree $\mathcal{T}$ preserves the validity property. That is, each remaining connected component of $\mathcal{C} \setminus \mathcal{T}$ has neighbors in only one branch of the this new DFS tree $\mathcal{T}$. This is because each newly added node is connected to the deepest possible point in the DFS. After each such repetition, the length of the remaining separator path P decreases by a 1/2 factor. Hence, after $O(\log n)$ iterations, we exhaust $\mathcal{P}$. At that point, we have a valid partial DFS $\mathcal{T}$ rooted at r and furthermore, $\mathcal{C} \setminus \mathcal{T}$ is made of connected components, each of which has size at most $2|\mathcal{C}|/3$.

Preparation for Next Recursions. At this point, we are almost ready for recursing on the connected components of $\mathcal{C} \setminus \mathcal{T}$, each as a subproblem of its own. Though, we should do a preparation step so that each subproblem is in the format that we assumed above, while describing the recursive step. In particular, we should identify the connected components

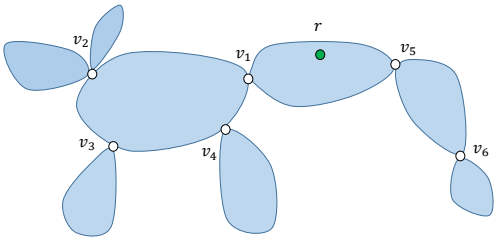


Figure 2 An induced connected subgraph $\mathcal{C}$ of G, depicted with its DFS root r, as well as its biconnected components and the corresponding cut-nodes v_1 to v_6.

$C_1, C_2, \ldots, C_\ell$ of $\mathcal{C} \setminus \mathcal{T}$, by giving a connected component identifier to each of them, and more importantly, we should declare a DFS root for each of them. Let each node in $\mathcal{T}$ sends its DFS depth to each of its neighbors in $\mathcal{C} \setminus \mathcal{T}$. Then, for each component C_i, we define the component leader and also the DFS root r_i to be a node $v \in C_i$ that received the greatest DFS depth from its $\mathcal{T}$ neighbors (breaking ties based on the id of v). Notice that for each component, this greatest depth $\mathcal{T}$-node is uniquely defined, because of the validity of the partial DFS. Moreover, this is a valid DFS root, in the sense that adding a DFS of C_i rooted at r_i to the current partial DFS $\mathcal{T}$ would be a correct partial DFS. These component leaders (i.e., component-wise DFS roots r_i) can be identified for all the connected components in parallel in $\widetilde{O}(D)$ rounds, using the connected component identification algorithm of Ghaffari and Haeupler [17] for planar graphs. It is crucial to note that here D is the diameter of the very base graph G and not just $\mathcal{C}$. See [17] for details.

Dealing with Cut Nodes. Finally, we come back to the assumption of the connected component $\mathcal{C}$ being biconnected, and we address the possibility of having cut nodes. Fig. 2 illustrates an example for this case, where a connected component $\mathcal{C}$ is drawn which has several cut nodes. In this case, we break the problem into several independent DFS problems that can be solved independently. In particular, we will partition the graph into edge-disjoint parts, each being one of the biconnected components of $\mathcal{C}$, and we solve a rooted DFS problem in each of these biconnected components. The root of the biconnected component containing root r is node r itself. For each other biconnected component C, the DFS root is the cut node of C that lies on the shortest path to the root r. It is easy to see that if we compute these rooted DFSs and glue them together in the natural way—hanging the DFS of each biconnected component C from its root as a subtree of DFS of the neighboring biconnected component closer to the node r — we get a DFS of $\mathcal{C}$. Computing a rooted DFS in each of these biconnected component is performed using the recursive method explained above. So, what remains to be explained is identifying two things (1) the biconnected components of $\mathcal{C}$, and (2) the corresponding DFS roots. We describe these components in the full version.

4 Computing A Separator Path

4.1 Method Outline and Challenges

A celebrated result of Lipton and Tarjan [30] demonstrates the existence of a *separator path* in planar graphs. Their proof shows that

> Any spanning tree T in a planar graph $G = (V, E)$ contains a tree path $P \subseteq T$ which is a *separator path*. That is, each connected component of $G \setminus P$ contains at most $2|V|/3$ nodes.

If one takes T to be a BFS (i.e., shortest path tree) of G, then the separator consists of at most two shortest paths. Hence, in this case, the separator path also has a small length of $O(D)$. For our purposes in this section, we do not need a small separator. Moreover, for reasons that shall become clear during the recursive steps, we will not be able to pick our separators based on BFS trees (of the remaining components). We will work with more general trees, and thus will not insist on the separator path being small. As a side remark, we note that if we did not need the separator to be a path, then there would be ways for having it be also small (even throughout the recursions).

In most applications of separators, we need to compute the separators not *once* but rather many times, recursively. That is, after computing a separator path in G, the separator is removed and the graph breaks into connected components; then in each component, we compute a separator and recurse. The first recursion level where we compute the separator in G may be delusively simple. This is because, whereas the diameter of G is D, in later levels, we need to compute the separator in connected induced subgraphs $\mathcal{C}$, which potentially may have much larger diameter than D.

Throughout this section, we describe how to compute a separator for a given induced subgraph $\mathcal{C} \subseteq G$, which is biconnected, but may have diameter much larger than D. We note that in reality, there are potentially many subgraphs $\mathcal{C}_1$, $\mathcal{C}_2$, $\ldots$, $\mathcal{C}_N$ for which we are computing separators, at the same time. Our description focuses on just one of these. Dealing with all these disconnected subgraphs in parallel will follow by standard usage of low-congestion shortcuts.

To avoid cumbersome notation, let us abuse notation and use n as the size of the subgraph $\mathcal{C}$. Our algorithm will compute a path P that breaks $\mathcal{C} \setminus P$ into components of size $[n/(3(1+\epsilon)), 2(1+\epsilon)n/3]$, for a small constant $\epsilon > 0$, say $\epsilon = 0.01$.

Algorithm Outline. Here, we describe a high-level outline of the algorithm for finding a separator path. We start by computing a spanning tree T in $\mathcal{C}$. This is done using the MST algorithm of [17], in $\widetilde{O}(D)$ rounds, where D is the diameter of the base graph G rather then the diameter of the subgraph $\mathcal{C}$. Our separator path will correspond to a fundamental cycle of the MST tree T in $\mathcal{C}$. Picking this primal tree T also leads to defining a *dual tree* T', containing the *dual edges* of the non T-edges, as described in Sec. 2. See Fig. 3. In this dual-tree T', each two faces who share a non-tree edge $e \notin T$ are adjacent. We will use this dual tree T' to find a collection of faces, i.e., dual nodes, that can be merged into a superface whose boundary can be used as a separator.

To choose a separator path on the tree T, we introduce the notion of *weight* for the dual-tree T'. We define the *weight* of a superface to be the number of nodes on the superface boundary plus the number of nodes inside the superface. Let $\mathcal{F}_i$ denote the superface corresponding to the dual-node v'_i, obtained by merging the faces of all dual-nodes in the subtree $T'(v'_i)$, i.e., the subtree of the dual tree T' rooted in v'_i. The weight of the subtree $T'(v'_i)$ is the weight of the superface $\mathcal{F}_i$.

Our algorithm would not be able to compute the exact weights and instead it would compute a $(1+\epsilon)$-approximation of these weights. Using these approximated weights, we explain how the algorithm chooses a separator path. First, the algorithm attempts to find

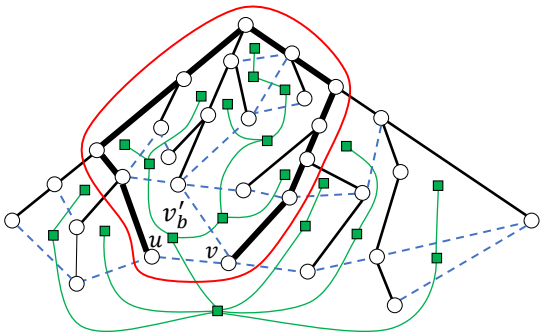


Figure 3 Shown is a planar graph and its path separator as computed by our algorithm. Solid edges are T-edges and the dashed blue edges are the non T-edges. These non-tree edges define the edges of the dual-tree T'. The dual-nodes are depicted as squares and the dual-edges of T' are the curved green edges in the figure. The dual-node v'_b is a balanced dual-node as the total weight of its superface (shown in the figure) is in $[n/(3(1+\epsilon)), 2(1+\epsilon)n/3]$. The boundary of the superface of v'_b—i.e., the subtree of dual rooted in v'_b—consists of one non T-edge $e = \{u, v\}$ and a T-path. The path-separator, indicated via thick black edges, is the T-path between u and v.

an (approximate) *balanced* dual-node v'_b such that the weight of its subtree $T'(v'_b)$ is in $[n/(3(1+\epsilon)), 2(1+\epsilon)n/3]$. If such a dual-node exists, then the boundary of the corresponding superface—obtained by merging all the faces in the dual subtree of v'_b—is a cycle separator. It is indeed a fundamental cycle of T. See Fig. 3 for an illustration. Otherwise, if no balanced dual-node exists, there must be a dual-node v'_c such that the weight of its subtree $T'(v'_c)$ is larger than $2(1+\epsilon)n/3$ but the weight of each of its descendants sub-trees is less than $n/(3(1+\epsilon))$. We call v'_c a *critical* dual-node.

In the case that we have a *critical* dual-node, we will compute a separator path slightly differently. This will be essentially by mimicking the separator computation of Lipton and Tarjan in the triangulated version of G. In fact, it will suffice to triangulate only the face corresponding to the dual-node v'_c. We note that generally, it is unclear how to efficiently simulate triangulation in distributed manner as this requires simulating many virtual edges. Our construction, however, only uses triangulation implicitly in the analysis. That is, we compute a separator and then show that it is the same as computed by the algorithm of Lipton and Tarjan on the triangulated version of $\mathcal{C}$.

Challenges and Our Approach for Overcoming Them. Our goal is to implement the above algorithm in $\widetilde{O}(D)$ rounds, where D is the diameter of the base graph G. Note that the diameter of $\mathcal{C}$ might be as large as $\Theta(n)$. We face two key challenges: (CI) we need to simulate each dual-node in a distributed manner. Note that a dual node is made of a face, which can be long, and it may interact with other faces through far apart parts of this face. (CII) More severely, we need to implement communications on the dual tree. The nodes and edges of this tree are not real nodes and edges of the graph. Even simulating each node of it is not straightforward, and is the challenge mentioned before. To add insult to injury, the diameter of the dual tree (even in terms of dual-nodes) can be much larger than D. For instance, it is possible that the primal graph has diameter $D = O(1)$ and yet, the diameter of the dual graph is $\Theta(n)$. We next briefly outline the methods we use for overcoming these two challenges.

To deal with challenge (CI), we use the low-congestion shortcuts of [17], as defined in Def. 2, one shortcut for each of the faces. This application is not straightforward because an

important requirement for low-congestion shortcuts is not met in our setting. To use the low-congestion shortcuts of [17], the collection of subsets $S_1, \ldots, S_k$ must be node-disjoint. In our case, however, the S_i sets are the nodes of faces. Hence, these sets are *not node-disjoint*; in fact, a node may belong to several different faces. We bypass this obstacle by transforming the graph G into an auxiliary graph $\widehat{G}$, in which the sets S_i, that correspond to the faces of $\mathcal{C}$, are mapped to node-disjoint connected sets. We then show that the auxiliary graph $\widehat{G}$ can be simulated efficiently in the original graph G.

To deal with challenge (CII), our approach is inspired by a method of [17, Section 5] for aggregating information on a tree with large diameter in planar graphs with low diameter. They used this method for aggregating information on the MST. Though, we need to adjust this method to suit our case. A straightforward combination would suggest a round complexity of $\tilde{O}(D^2)$. This is because, our method for communication inside faces (i.e., dual-nodes) itself takes $\tilde{O}(D)$ rounds, and on top of that, the method of [17, Section 5] for dealing with large-diameter trees needs $\tilde{O}(D)$ iterations of communicating on the dual-nodes. Thus, the naive combination would be $\tilde{O}(D^2)$. We will however be able to put the ideas together in a way that leads to a round complexity of $\tilde{O}(D)$.

Roadmap. In Sec. 4.2, we present the basic computational tools for efficient distributed communication inside a dual-node and on a dual tree, i.e., dealing with challenges (CI) and (CII) respectively. Then, we present our algorithm for computing a path-separator in an arbitrary (biconnected) induced subgraph $\mathcal{C} \subseteq G$, using the tools explained in Sec. 4.2. The related analysis and smaller subroutines appear in the full version.

4.2 Key Tools

We begin by explaining how to preform communication inside nodes of each face, and later how to perform communication on the dual tree.

4.2.1 Tool (I): Communication Inside Dual-Nodes

To simulate communication inside the dual-nodes, we consider two basic tasks.

(T1) Face identification: Assign each face F_i in $\mathcal{C}$ a unique ID, $ID(F_i)$, such that each node knows the IDs of the faces to which it belongs. In addition, for each edge $\{u, v\} \in \mathcal{C}$, the endpoints of this edge should know the two face IDs, $(ID(F_i), ID(F_j))$, to which the edge $\{u, v\}$ belongs.

(T2) Low-Congestion Shortcuts for all Faces: Let S_i denote the nodes of face F_i. Compute an (α, β) low-congestion shortcuts H_i for the S_i sets, for $\alpha, \beta = \widetilde{O}(D)$.

To tackle both of these tasks, we transform the original planar graph G into a virtual planar graph $\widehat{G}$ in which the subsets of nodes belonging to the faces of $\mathcal{C}$ are mapped to *node-disjoint* subsets $\widehat{S}_i$ for which low-congestion shortcuts can be computed. We then show that any r-round algorithm for $\widehat{G}$ can be simulated in G using $2r$ rounds.

The virtual graph $\widehat{G}$ is defined as follows. See Fig. 4. First, it contains all edges of $G \setminus \mathcal{C}$. The edges of $\mathcal{C}$ are transformed in the following manner. Consider a node v that belongs to y faces $F_{i_1}, \ldots, F_{i_y}$ in $\mathcal{C}$, ordered in a clockwise manner. Then, v creates y many virtual copies of itself named $v^1, \ldots, v^y$. In $\widehat{G}$, the identifier of the ℓ^{th} copy of v is (ID_v, ℓ). By computing the embedding of the original graph G, each node v knows the clockwise ordering of all its edges in G. This can be used to deduce the clockwise ordering[3] of its edges in $\mathcal{C}$.

[3] Note that since the diameter of $\mathcal{C}$ can be larger than D, we cannot afford computing the embedding for $\mathcal{C}$ from scratch, via communicating only inside $\mathcal{C}$.

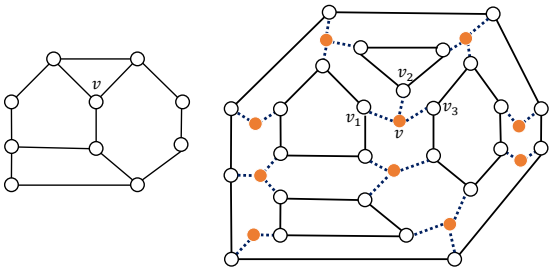


Figure 4 The transformation from G to $\widehat{G}$, which maps faces to node-disjoint connected subsets. The left figure depicts the graph before the transformation, and the right one depicts it after the transformation. The dotted links show the star-edge E_S. Notice that in the graph $\widehat{G}$ after the transformation, if we remove the star-edges, we get a collection of connected components, each corresponding to a face of $\mathcal{C}$.

The clockwise ordering of the edges of v in $\mathcal{C}$ imposes a local numbering of its faces in $\mathcal{C}$, each two consecutive edges in the clock-wise order define one new face. On each edge $\{u, v\}$, the nodes u and v exchange their local face numberings for that edge. Since a given edge appears in at most two faces, this can be done in 2 rounds. In the graph $\widehat{G}$, we connect v to y copies $v^1, \ldots, v^y$, one per face in $\mathcal{C}$. In addition, for each edge $\{v, u\} \in \mathcal{C}$ belonging to the i^{th} face of v and the j^{th} face of u, we connect v^i to u^j. We use E_S to denote the set of star-edges $\{v, v_i\}$ in $\widehat{G}$.

The graph $\widehat{G}$ is planar. Furthermore, it has the additional benefit that the nodes corresponding to the faces of the $\mathcal{C}$ are now node-disjoint subsets, while still each face induces a connected subgraph. Hence, one can construct low-congestion shortcuts for these node sets in the graph $\widehat{G}$. Notice that $\widehat{G}$ has diameter at most $3D$. This is because every edge $\{u, v\} \in \mathcal{C}$ becomes a path $(u - u^i - v^j - v)$ in $\widehat{G}$, and every edge not in $\mathcal{C}$ is unchanged. Since each edge belongs to two faces, we have:

▶ **Lemma 3** (Simulation of $\widehat{G}$ in G)**.** *Any r-round algorithm $\mathcal{A}$ in $\widehat{G}$ can be implemented in G within at most $2r$ rounds.*

Proof. The edges of $\mathcal{C}$ are transformed into two types of edges in $\widehat{G}$: star-edges between v and its copies, whose simulation requires no real communication in G, and face-edges $\{v^j, u^i\}$. Since each edge $\{u, v\}$ in G simulates the communication of two edges, namely, $\{u^{i_1}, v^{j_1}\}$ and $\{u^{i_2}, v^{j_2}\}$ in $\widehat{G}$, every round r of $\mathcal{A}$ in $\widehat{G}$ can be implemented in G using two rounds. ◀

From now on, it suffices to consider algorithms in $\widehat{G}$. Since the node faces are the connected components of $\widehat{G} \setminus E_S$, we have:

▶ **Lemma 4.** *The Faces Identification task can be solved in $\widetilde{O}(D)$ rounds.*

Proof. Let $\mathcal{C}$ be the induced subgraph of G, which is biconnected, and for which we are computing a separator path. Let $\widehat{\mathcal{C}}$ be the subgraph of $\widehat{G}$. We employ the $\widetilde{O}(D)$-round connectivity algorithm of [17] in the graph $\widehat{G} \setminus E_S$ but only for the nodes of $\mathcal{C}$. Recall that E_S denotes the star edges in $\widehat{G}$. By using Lemma 3, this algorithm can be simulated in G in $\widetilde{O}(D)$ rounds. Let the ID of each connected component of $\widehat{\mathcal{C}} \setminus E_S$ be the node with maximum ID in the component. Since each connected component of $\widehat{\mathcal{C}} \setminus E_S$ corresponds to a face of $\mathcal{C}$, each node now knows the IDs of its faces, in particular, it knows the face IDs of each of its

copies in $\widehat{\mathcal{C}}$. In addition, each node $v \in \mathcal{C}$ also learns the IDs of the two faces $ID(F_i)$ and $ID(F_j)$ of each of its edges $\{u, v\}$ in $\mathcal{C}$. The lemma follows. ◀

Turning to the second task of computing low congestion shortcuts for each face F_i, we have:

▶ **Lemma 5.** *Let $S_1, \ldots, S_N$ be the nodes on faces $F_1, \ldots, F_N$ of the graph $\mathcal{C}$. W.h.p., one can construct in $\widetilde{O}(D)$ rounds, an (α, β) low-congestion shortcut graphs $H_1, \ldots, H_N$ for $\alpha, \beta = O(D \log D)$.*

Proof. Consider the algorithm $\mathcal{A}$ of [17] for constructing the low-congestion shortcuts in $\widehat{G}$. By Lemma 3, Algorithm $\mathcal{A}$ can be simulated in G in $\widetilde{O}(D)$ rounds. Let $\widehat{H}_i$ be the $(\alpha, \beta/2)$ low-congestion shortcuts computed for the sets $\widehat{S}_i$ in $\widehat{G}$. Let H_i be obtained from $\widehat{H}_i$ by omitting star-edges $\{v, v^j\}$ and replacing $\{u^i, v^j\}$ edges with $\{u, v\}$ edges. The subgraphs H_i are (α, β) low-congestion shortcuts for the sets S_i in $\mathcal{C}$. ◀

▶ **Corollary 6.** *One can compute any aggregate function, which has $O(\log n)$-bit size values, in all faces of $\mathcal{C}$ in parallel in $\widetilde{O}(D)$ rounds.*

4.2.2 Tool (II): Communication on the Dual Tree

In tool (I), we described how to perform efficient communication within each face, that is, inside each node of the dual tree. We now explain how to perform efficient communication on the dual tree T' of a spanning tree T of the subgraph $\mathcal{C}$. We mainly need to solve the following two computational tasks in the dual tree T': (D1) Edge Orientation: orienting the dual-edges towards a given dual root, and (D2) Subset-OR: given a rooted dual tree T', and initial binary input values $x(v')$ for each dual-node v', the leader node $\ell(v')$ of the dual node v' should learn the OR of its subtree, that is, the value $y(v') = \vee_{u' \in T'(v')} x(u')$.

An important tool for both of these tasks is a recursive fragment merging process, which we describe next. In Subsec. 4.2.2, we then describe how to use this recursive merging to solve the two tasks (D1) and (D2).

Recursive Face-Merging Process. To avoid computation in time $O(Diam(T'))$, we employ an idea of [17, Section 5]. It is worth noting that this idea itself is inspired by merges in the style of Boruvka's classical minimum spanning tree algorithm [35].

We have $O(\log n)$ levels of merging faces, where each merge happens along some edge of the dual-tree node T'. The faces involved in each merge correspond to a connected subgraph of the dual tree, which we will call a *fragment* or a *face-fragment*, stressing that it is a merge of some faces. The dual-tree gets partitioned into fragments in a hierarchical fashion, where the fragments of level i are formed by merging fragments of level $i-1$. See Fig. 5 for an illustration. Considering that the dual-tree nodes are faces of the primal graph, the fragments of the i^{th}-level are obtained by merging the (sets of) faces corresponding to the fragments of level $i-1$. The $O(\log n)$ levels of face-fragment merging of the dual tree T' are implemented by using low-congestion shortcuts in G, as described next. For every fragment j in level i, let $S_{i,j}$ be the set of nodes appearing on the faces of fragment j. By using the tools provided in Subsec. 4.2.1 and mainly Lemma 5, we construct low-congestion shortcut subgraphs for each set $S_{i,j}$ (i.e., despite the fact that these sets are not disjoint). Here, we slightly change the definition of the auxiliary graph $\widehat{G}$ that was defined in Subsec. 4.2.1. For simplicity, consider the first level of the face merging process where two faces of $\mathcal{C}$, say F_j and F_k, are merged. Let $e = \{u, v\}$ be a common edge of F_j and F_k. The endpoint u

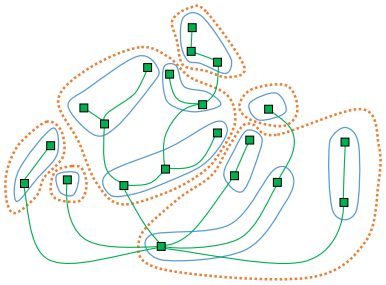


Figure 5 The fragmentation of the dual-tree from Fig. 3. Shown are the first three levels of merging. As each dual-node corresponds to a face in G, the merged fragment of the dual-tree is formed by a merging of faces.

indicates the merging of these faces in the auxiliary graph $\widehat{G}$, by adding an edge between its copies u^j and u^k corresponding to the merged faces F_j and F_k. As a result, the nodes on the faces F_j and F_k now belong to the same connected component in the graph $\widehat{G} \setminus E_S$ (where E_S are the star-edges $\{u, u^j\}$). Since the face identification is done by identifying the connected components of $\widehat{G} \setminus E_S$, this step ensures that F_j and F_k would be identified as one merged face.

Equipped with the low-congestion shortcut subgraphs for each face-fragment (i.e., the node sets $S_{i,j}$), all nodes inside each fragment can communicate in their fragment in parallel, for all fragments in level i, in $\widetilde{O}(D)$ rounds. Hence, the $O(\log n)$ face merging process can be done in $\widetilde{O}(D)$ rounds. A detailed description of the face merging process is described in the full version. We conclude by presenting a concise description of the entire algorithm, its detailed description and analysis is deferred to the full version.

Algorithm ComputePathSep

Input: A n-node biconnected induced subgraph $\mathcal{C}$ of a planar graph G with diameter D, approximation parameter $\epsilon \in (0, 1/2)$.
Output: A separator path P in $\mathcal{C}$, so that each component of $C \setminus P$ has size at most $2(1+\epsilon)n/3$.

- **Step (S1): Computing the Dual Tree T'**
  - Compute an MST T in $\mathcal{C}$. Non T-edges of $\mathcal{C}$ correspond to the edges of dual-tree T'.
- **Step (S2): Orienting the Dual Tree T' Towards a Root**
  - This step is done via a recursive face-fragment merging process.
- **Step (S3): Computing the Weights of the Dual Nodes in T'**
  - For each $i \in \{1, \ldots, \log_{1+\epsilon} n\}$, we have $N_\epsilon = O_\epsilon(\log n)$ *experiments*, as follows:
    - Sample each of the nodes of $\mathcal{C}$ with probability $1/(1+\epsilon)^i$.
    - Use Subset-OR to inform each dual-node if there is a marked node in its subtree.
  - Using these experiments, dual-nodes deduce a $(1+\epsilon)$ approximation of their weight.
  - Detect a balanced dual-node, i.e., a dual-node with weight in $[n/(3(1+\epsilon)), 2(1+\epsilon)n/3]$.
  - If there is no balanced dual-node, detect a critical dual-node, that is, a dual-node with weight at least $2(1+\epsilon)n/3$ but each of its children has weight less than $n/(3(1+\epsilon))$.
- **Step (S4): Marking the Separator Path**
  - For balanced dual node: mark the tree path connecting the boundary of its superface.
  - For critical dual-node: mark a tree path by simulating Lipton-Tarjan on its superface.

References

1. A. Aggarwal and R. Anderson. A random nc algorithm for depth first search. In *STOC*, pages 325–334, 1987.
2. Richard Anderson. A parallel algorithm for the maximal path problem. *Combinatorica*, 7(4):315–326, 1987.
3. RJ Anderson. A parallel algorithm for depth-first search. 1986. In *Extended abstract*, 1986.
4. Baruch Awerbuch. A new distributed depth-first-search algorithm. *Information Processing Letters*, 20(3):147–150, 1985.
5. Keren Censor-Hillel, Mohsen Ghaffari, and Fabian Kuhn. Distributed connectivity decomposition. In *PODC*, 2014.
6. 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. In *STOC*, pages 363–372, 2011.
7. Atish Das Sarma, Danupon Nanongkai, and Gopal Pandurangan. Fast distributed random walks. In *PODC*, pages 161–170, 2009.
8. Atish Das Sarma, Danupon Nanongkai, Gopal Pandurangan, and Prasad Tetali. Efficient distributed random walks with applications. In *PODC*, pages 201–210, 2010.
9. Michael Elkin. Unconditional lower bounds on the time-approximation tradeoffs for the distributed minimum spanning tree problem. In *STOC*, pages 331–340, 2004.
10. Shimon Even. *Graph algorithms*. Cambridge University Press, 2011.
11. Stephen Fenner, Rohit Gurjar, and Thomas Thierauf. Bipartite perfect matching is in quasi-nc. In *STOC*, pages 754–763. ACM, 2016.
12. Silvio Frischknecht, Stephan Holzer, and Roger Wattenhofer. Networks cannot compute their diameter in sublinear time. In *SODA*, pages 1150–1162, 2012.
13. J.A. Garay, S. Kutten, and D. Peleg. A sub-linear time distributed algorithm for minimum-weight spanning trees. In *FOCS*, 1993.
14. M. Ghaffari and F. Kuhn. Distributed minimum cut approximation. In *DISC*, pages 1–15, 2013.
15. Mohsen Ghaffari. Near-optimal distributed approximation of minimum-weight connected dominating set. In *International Colloquium on Automata, Languages, and Programming*, pages 483–494. Springer, 2014.
16. Mohsen Ghaffari and Bernhard Haeupler. Distributed algorithms for planar networks i: Planar embedding. In *PODC*, pages 29–38, 2016.
17. Mohsen Ghaffari and Bernhard Haeupler. Distributed algorithms for planar networks ii: Low-congestion shortcuts, mst, and min-cut. In *SODA*, pages 202–219, 2016.
18. Mohsen Ghaffari, Andreas Karrenbauer, Fabian Kuhn, Christoph Lenzen, and Boaz Patt-Shamir. Near-optimal distributed maximum flow. In *PODC*, pages 81–90. ACM, 2015.
19. Mohsen Ghaffari and Christoph Lenzen. Near-optimal distributed tree embedding. In *International Symposium on Distributed Computing*, pages 197–211. Springer, 2014.
20. Andrew V Goldberg, Serge A Plotkin, and Pravin M Vaidya. Sublinear-time parallel algorithms for matching and related problems. In *Foundations of Computer Science, 1988., 29th Annual Symposium on*, pages 174–185. IEEE, 1988.
21. Bernhard Haeupler, Taisuke Izumi, and Goran Zuzic. Low-congestion shortcuts without embedding. In *PODC*, pages 451–460. ACM, 2016.
22. Bernhard Haeupler, Taisuke Izumi, and Goran Zuzic. Near-optimal low-congestion shortcuts on bounded parameter graphs. In *International Symposium on Distributed Computing*, pages 158–172. Springer, 2016.
23. Torben Hagerup. Planar depth-first search in o(\logn) parallel time. *SIAM Journal on Computing*, 19(4):678–704, 1990.

24 Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. A deterministic almost-tight distributed algorithm for approximating single-source shortest paths. In *STOC*, pages 489–498, 2016.

25 Stephan Holzer and Roger Wattenhofer. Optimal distributed all pairs shortest paths and applications. In *PODC*, pages 355–364, 2012.

26 Ming-Yang Kao. All graphs have cycle separators and planar directed depth-first search is in dnc. In *Aegean Workshop on Computing*, pages 53–63. Springer, 1988.

27 Shay Kutten and David Peleg. Fast distributed construction of k-dominating sets and applications. In *PODC*, pages 238–251, 1995.

28 Christoph Lenzen and Boaz Patt-Shamir. Fast routing table construction using small messages: Extended abstract. In *STOC*, pages 381–390, 2013.

29 Christoph Lenzen and Boaz Patt-Shamir. Improved distributed steiner forest construction. In *PODC*, 2014.

30 Richard J Lipton and Robert Endre Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

31 Kurt Mehlhorn and Peter Sanders. Graph traversal. *Algorithms and Data Structures: The Basic Toolbox*, pages 175–189, 2008.

32 Danupon Nanongkai. Distributed approximation algorithms for weighted shortest paths. In *STOC*, 2014.

33 Danupon Nanongkai, Atish Das Sarma, and Gopal Pandurangan. A tight unconditional lower bound on distributed randomwalk computation. In *PODC*, pages 257–266, 2011.

34 Danupon Nanongkai and Hsin-Hao Su. Almost-tight distributed minimum cut algorithms. In *International Symposium on Distributed Computing*, pages 439–453. Springer, 2014.

35 Jaroslav Nešetřil, Eva Milková, and Helena Nešetřilová. Otakar boruvka on minimum spanning tree problem translation of both the 1926 papers, comments, history. *Discrete Mathematics*, 233(1):3–36, 2001.

36 David Peleg. *Distributed Computing: A Locality-sensitive Approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

37 David Peleg and Vitaly Rubinovich. A near-tight lower bound on the time complexity of distributed MST construction. In *FOCS*, pages 253–, 1999.

38 John H Reif. Depth-first search is inherently sequential. *IPL*, 20(5):229–234, 1985.

39 Gregory E Shannon. A linear-processor algorithm for depth-first search in planar graphs. *IPL*, 29(3):119–123, 1988.

40 Steven S Skiena. *The algorithm design manual: Text*, volume 1. Springer Science & Business Media, 1998.

41 Justin R Smith. Parallel algorithms for depth-first searches i. planar graphs. *SIAM Journal on Computing*, 15(3):814–830, 1986.

Dynamic Analysis of the Arrow Distributed Directory Protocol in General Networks*

Abdolhamid Ghodselahi[1] and Fabian Kuhn[2]

1 **Department of Computer Science, University of Freiburg, Germany**
`hghods@cs.uni-freiburg.de`
2 **Department of Computer Science, University of Freiburg, Germany**
`kuhn@cs.uni-freiburg.de`

Abstract

The Arrow protocol is a simple and elegant protocol to coordinate exclusive access to a shared object in a network. The protocol solves the underlying distributed queueing problem by using path reversal on a pre-computed spanning tree (or any other tree topology simulated on top of the given network).

It is known that the Arrow protocol solves the problem with a competitive ratio of $O(\log D)$ on trees of diameter D. This implies a distributed queueing algorithm with competitive ratio $O(s \log D)$ for general networks with a spanning tree of diameter D and stretch s. In this work we show that when running the Arrow protocol on top of the well-known probabilistic tree embedding of Fakcharoenphol, Rao, and Talwar [STOC 03], we obtain a randomized distributed online queueing algorithm with expected competitive ratio $O(\log n)$ against an oblivious adversary even on general n-node network topologies. The result holds even if the queueing requests occur in an arbitrarily dynamic and concurrent fashion and even if communication is asynchronous. The main technical result of the paper shows that the competitive ratio of the Arrow protocol is constant on a special family of tree topologies, known as hierarchically well separated trees.

1998 ACM Subject Classification F.1.2 Online computation, F.2.2 Computations on discrete structures, G.2.2 Network problems

Keywords and phrases Arrow protocol, competitive analysis, distributed queueing, shared objects, tree embeddings

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.22

1 Introduction

Coordinating the access to shared data is a fundamental task that is at the heart of almost any distributed system. For example, when implementing a distributed shared memory system on top of a message passing system, each shared register has to be kept in a coherent state despite possibly a large number of concurrent requests to read or write the shared register. In a distributed transactional memory system, each transaction might need to operate on several shared objects, which need to be kept in a consistent state [17, 26, 30]. When implementing a shared object on top of large-scale network, a *distributed directory protocol* can be used to improve scalability of the system [1, 2, 4, 6, 7, 17, 26]. When a network node requires access to a shared object, the directory moves a copy of the object

* A full version of this paper is available at `https://arxiv.org/abs/1705.07327` [11].


© Abdolhamid Ghodselahi and Fabian Kuhn;
licensed under Creative Commons License CC-BY

to the node requesting the object. If the node changes the state of the shared object, the directory protocol has to make sure that all existing copies of the object are kept in a consistent state.

Distributed Queueing. At the core of many distributed directory implementations is the following basic *distributed queueing problem* that allows the system to order potential concurrent access requests to a shared object [15]. The nodes of a network issue queueing requests (e.g., requests to access a shared object) in a completely dynamic and possibly arbitrarily concurrent manner. A queueing protocol needs to globally order all the requests so that they can be acted on consecutively. Formally, each request has to find its *predecessor request* in the order. That is, when enqueueing a request r issued by some node v, a queueing protocol needs to find the request r' that currently forms the tail of the queue and inform the node v' of request r' about the new request r. The cost of enqueueing request r after request r' is defined as the delay from when request r is issued until when the node v' knows that r is the successor request of r'.

The Arrow Protocol. A particularly simple and elegant solution for this distributed queueing problem is given by the Arrow protocol, which was introduced independently (in slightly different forms) by Naimi and Trehel, Raymond, as well as van de Snepscheut in the context of distributed mutual exclusion [21, 24, 29]. The Arrow protocol operates on a directed tree topology $T = (V, E)$. In a quiescent state, the tree is rooted at the node u of the current tail of the queue, i.e., all edges of T are directed towards u. When a new queueing request is issued at a node v, the directions of the edges on the path between v and the previous tail u are reversed so that the tree is now rooted at v. For a precise description of the protocol, we refer to Section 2. It has been shown in [8] that the Arrow protocol correctly solves the queueing problem even in an asynchronous system even if the requests are issued in a completely dynamic and possibly concurrent way. Moreover, the Arrow protocol guarantees that every request finds the node of its predecessor on a direct path (i.e., within D time units if D is the diameter of T). The overall cost of some queueing algorithm is the sum of delays between when each request is issued and when its predecessor in the final order knows about it. In [14], it was further shown that on a tree T, the overall cost of the Arrow protocol for ordering a dynamic set of queueing requests is within a factor $O(\log D)$ of the cost of an optimal offline queueing algorithm, which knows the request sequence in advance.[1]

Contribution. In the present paper, we strengthen the result of [14] and we show that when run on the right underlying tree, the Arrow protocol is $O(\log n)$-competitive even on general n-node network topologies. The competitive ratio achieved by the Arrow protocol is the worst case ratio between the cost of Arrow and the cost of an optimal offline ordering strategy. The best previously known competitive ratio for the distributed queueing problem with arbitrarily dynamically injected requests on general graphs is $O(\log^2 n \cdot \log D)$ as shown in [27] for the hierarchical schemes defined of [4, 26]. This shows that (under some assumptions), the simple and elegant Arrow protocol outperforms all existing significantly more complicated distributed queueing protocols. We note that our protocol is based on a randomized tree construction and its competitive ratio is w.r.t. an oblivious adversary (i.e., the adversary needs to determine the sequence of requests before the construction of the tree). Other

[1] Note that this implies a competitive ratio of $O(s \cdot \log D)$ for general graphs if a spanning tree T of diameter D and stretch s is given.

protocols with polylogarithmic competitive ratio are deterministic and they therefore also work in the presence of an adaptive adversary. For a more detailed comparison of our results with existing protocols, we refer to the discussion in Section 1.1.

More specifically, as our main technical result, we show that the Arrow protocol is $O(1)$-competitive when it is run on a special class of trees known as *hierarchically well separated trees* [5]. A hierarchically well separated tree (in the following referred to as an HST) with parameter $\alpha > 1$ is a weighted rooted tree, where on each level, all the nodes are at the same distance to the root (the distance to the root depend on α and on the level) and all the leaves are on the same level (and thus also at the same distance to the root). Further, the edge lengths decrease exponentially (by a factor α per level) when going from the root towards the leaves. The properties of HSTs as well as the way we utilize HSTs are formally described in Section 2. When running Arrow on an HST T, we assume that all requests are issued at the leaves of T. We show that the total cost of an Arrow execution on an HST T is within a constant factor of the total cost of an optimal offline algorithm for the given set of requests. Our result even holds if the communication on T is asynchronous.

▶ **Theorem 1.** *Assume that we are given an HST T with parameter $\alpha = 2$ and queueing requests R that arrive in an arbitrarily dynamic manner at the leaves of T. When using the* Arrow *protocol on tree T, the total cost for ordering the requests in R is within a constant factor of the cost of an optimal offline algorithm for ordering the requests R on T. This even holds if communication is asynchronous.*

▶ Remark. Because the statement of the theorem applies to the general asynchronous case, it also captures a synchronous scenario, where the delay on each edge is fixed, but might be smaller than the actual weight of the edge in the HST. Such executions are relevant because an HST T is often built as an overlay graph on top of an underlying network graph G, where each edge of weight w in T is mapped to a path of length at most w in G and thus even in a synchronous execution, the delay when sending a message across the edge of T might be smaller than the weight of the edge.

For a precise description of the Arrow protocol and the definition of queueing cost, we refer to Section 2. When combining Theorem 1 with the celebrated probabilistic tree embedding of Fakcharoenphol, Rao, and Talwar [9], we get our main result for general graphs. In [9], it is shown that there is a randomized algorithm that given an arbitrary n-point metric (X, d) constructs an HST T such all points X are mapped to leaves of T, all distances in (X, d) are upper bounded by the respective distances in T, and the expected distance between any two leaves in T is within an $O(\log n)$ factor of the distance between the corresponding two points in X. When constructing such an HST T for a given graph G and when assuming an oblivious adversary[2], this implies that the expected total cost of Arrow on T is within an $O(\log n)$ factor of the optimal offline queueing cost on G. We also note that an efficient distributed construction of the HST embedding of [9] has been given in [10].

▶ **Theorem 2.** *Assume that we are given an arbitrary graph $G = (V, E)$ and queueing requests R that arrive in an arbitrarily dynamic manner at the nodes of G. There is a randomized construction of an HST T that can be simulated on G such that when running* Arrow *on T, we get a distributed queueing algorithm for G with expected competitive ratio at most $O(\log n)$ against an oblivious adversary providing the sequence of requests. This even holds if communication is asynchronous.*

[2] That is, when assuming that the sequence of requests is statistically independent of the randomness used to construct the HST T or equivalently, if the adversary determines the sequence of requests before the tree T is constructed.

Organization of the Paper. The remainder of the paper is organized as follows. Section 2 formally defines the queueing problem, the Arrow protocol, as well as the cost model used in our paper. The section also contains some lemmas that establish some basic properties that are needed for the rest of the paper. Section 3 analyzes the cost of an optimal offline algorithm on an HST T by relating it to the total weight of an MST defined on the set of requests. In Section 4, we introduce a general framework to analyze the queueing cost of distributed queueing algorithms on an HST T and the framework is applied to synchronous executions of the Arrow protocol. For the analysis of asynchronous executions, we refer to the full version of the paper [11]. Due to lack of space, we also need to omit most proofs throughout the technical sections of the paper. All the missing proofs can also be found in the full version [11].

1.1 Related Work

The Arrow protocol has been introduced (in somewhat different forms) by Naimi and Trehel, Raymond, as well as van de Snepscheut [24] as a way to solve the mutual exclusion problem in a network. The protocol was later reinvented by Demmer and Herlihy [8], who used Arrow to implement a distributed directory [6]. Over the years, Arrow has been used and analyzed in different contexts [14, 16, 18, 19, 22, 28]. The protocol has been implemented as a part of Aleph Toolkit [16] and shown to outperform centralized schemes significantly in practice [19]. Several other tree-based distributed queueing protocols that are similar to the Arrow protocol have also been proposed in the literature. A protocol that combines the ideas of Arrow with path compression has been implemented in the Ivy system [20]. The amortized cost to serve a single request is only $O(\log n)$ [12], however the protocol needs a complete graph as the underlying network topology. There are also other similar protocols that operate on fixed trees. The Relay protocol [30] has been introduced as a distributed transactional memory protocol. It is run on top of a fixed spanning tree similar to Arrow, however to more efficiently deal with aborted transactions, it does not always move the shared object to the node requesting it. Further, in [2], a distributed directory protocol called Combine has been proposed. Combine runs on a fixed overlay tree and it is in particular shown in [2] that Combine is starvation-free.

The first paper to study the competitive ratio of concurrent executions of a distributed queueing protocol is [15]. The paper shows that in synchronous executions of Arrow on a tree T, if all requests are issued at time 0 (known as one-shot executions), the total cost of Arrow is within a factor $O(\log |R|)$ compared with the optimal queueing cost on tree T. This analysis has later been extended (and slightly strengthened) to the general concurrent setting where requests are issued in an arbitrarily dynamic fashion. In [14], it is shown that in this case, the total cost of Arrow is within a factor $O(\log D)$ of the optimal cost on the tree T. Later, the same bounds have also been proven for the Relay protocol [30] and the Combine protocol [2]. Typically, these protocols are run on a spanning tree or an overlay tree on top of an underlying general network topology. While the cost of all these protocols is small when compared with the optimal queueing cost on the tree, the cost of the protocols might be much larger when compared with the optimal cost on the underlying topology. In this case, the competitive ratio becomes $O(s \cdot \log D)$, where s is the stretch of the tree. There are underlying graphs (e.g., cycles) for which every spanning tree and even every overlay tree has stretch $\Omega(n)$ [13, 23]. The fact that even the best spanning tree might have large stretch initiated the work on distributed queueing protocols that run on more general hierarchical structures. In [17], a protocol called Ballistic is introduced and analyzed for the sequential and the one-shot case. Ballistic has competitive ratio $O(\log D)$, however the protocol requires

the underlying distance metric to have bounded doubling dimension and it thus cannot be applied in general networks. The best protocol known for general networks is Spiral, which was introduced in [26]. Spiral is based on a hierarchy of overlapping clusters that cover the graph. It's general structure is thus somewhat resembling the classic sparse partitions and mobile objects solutions by Awerbuch and Peleg [3, 4]. The competitive ratio of Spiral is shown to be $O(\log^2 n \cdot \log D)$ for sequential and one-shot executions in [26]. In [27], a general framework to analyze the cost of concurrent executions of hierarchical queueing and directory protocols has been presented. In particular, in [27], the competitive analysis of Spiral and also of the classic mobile object algorithm of Awerbuch and Peleg [3, 4] has been extended to the dynamic setting. In [14], it is sketched how the competitive analysis for Arrow generalizes to the asynchronous case.

2 Model, Problem Statement, and Preliminaries

Communication Model. We consider a standard message passing model on a network modeled by a graph $G = (V, E)$. In some cases, the edges of G have weights $w : E \to \mathbb{R}_{>0}$, which are assumed to be normalized such that $w(e) \geq 1$ for all $e \in E$. We distinguish between synchronous and asynchronous executions. In a *synchronous execution*, the delay for sending a message from a node u to a node v over an edge e connecting u and v is exactly 1 if the edge is unweighted and exactly $w(e)$ otherwise. In an *asynchronous execution*, message delays are arbitrary, however when analyzing an asynchronous execution, we assume that the message delay over an edge e is upper bounded by the edge weight $w(e)$ (or by 1 in the unweighted case).

The Distributed Queueing Problem. In the *distributed queueing problem* on a graph $G = (V, E)$, a set R of queueing requests $r_i = (v_i, t_i)$ are issued at the nodes of V (every node can issue multiple requests) in an arbitrarily dynamic fashion. The goal of a queueing algorithm is to order all the requests. Specifically, if a request $r_i = (v_i, t_i)$ is issued at node v_i at time $t_i \geq 0$, the algorithm needs to enqueue the request r_i by informing the node v_j of the predecessor request $r_j = (v_j, t_j)$ in the constructed global order. For this purpose, every queueing algorithm in particular has to send (possibly indirectly) a respective message from node v_i to v_j. We assume that at time 0, when an execution starts, the tail of the queue is at a given node $v_0 \in V$. Formally, this is modeled as a request $r_0 = (v_0, 0)$ which has to be ordered first by any queueing protocol. We sometimes refer to r_0 as the dummy request. For a set R' of queueing requests (and sometimes by overloading notation also for a set of request indexes), we define $t_{\min}(R')$ and $t_{\max}(R')$ to be the minimum and the maximum issue time t of any request $r = (v, t) \in R'$, respectively.

The Arrow Protocol. The Arrow protocol [24] is a distributed queueing protocol that operates on a tree network $T = (V, E)$. At each point in time, each node $v \in V$ has exactly one outgoing link (arrow) pointing either to one of the neighbors of v or to the node v itself. In a quiescent state, the arrow of the node of the request at the tail of the queue points to itself and all other arrows point towards the neighbor on the path towards the tail of the queue (i.e., the tree is directed towards the current tail). When a new request at a node $v \in V$ occurs, a "find predecessor" message is sent along the arrows until it finds the predecessor request. While following the path to the direction of the arrows are reversed. More formally, a request r at node v is handled as follows.

1. If the arrow of v points to v itself, r is queued directly behind the previous request issued at v. Otherwise if the arrow points to neighbor u, *atomically*, a "find predecessor" message (including the information about request r) is sent to u and the arrow of v is redirected to v itself.
2. If a node u receives a "find predecessor" message for request r from a neighbor w, if the arrow of u points to itself, *atomically*, the request r is queued directly behind the last request issued by node u and the arrow of u is redirected to node w. Otherwise, if the arrow of u points to neighbor x, *atomically*, the "find predecessor" message is forwarded to node x and the arrow of node u is redirected to node w.

For a more detailed description of the Arrow protocol and of how Arrow handles concurrent requests, we refer the reader to [8, 14]. It was shown in [8] that the Arrow protocol correctly orders a given sequence of requests even in an asynchronous network. Moreover as shown in [8, 14], when operating on tree T, the protocol always finds the predecessor of a request on the direct path on T. As a result, if two requests r' and r are at distance d on T and if r' is the predecessor of r in the queueing order, the "find predecessor" message initiated by request r finds the node of request r' in time exactly d in the synchronous setting and in time at most d in the asynchronous model. Further, it is shown in [14] that the successor request of a request r at node v in the queue is always the remaining request r'' that first reaches v on a direct path. This "greedy" nature of the Arrow ordering was used in [15], where it was shown that in the one-shot case when all requests occur at time 0, the Arrow order corresponds to a greedy (nearest neighbor) TSP path through requests, whereas an optimal offline algorithm corresponds to an optimal TSP path on the request set. The competitive ratio on trees then follows from the fact that the nearest neighbor heuristic provides a logarithmic approximation of the TSP problem [25]. In [14], this analysis was extended and it was shown that even in the fully dynamic case, it is possible to reduce the problem to a (generalized) TSP nearest neighbor analysis. Formally, the greedy nature of the Arrow protocol in the synchronous setting is captured by Lemma 7 in Section 3.

Hierarchically Well Separated Trees. The notion of a *hierarchically well separated tree* (HST) was defined by Bartal in [5]. Given a parameter $\alpha > 1$, an HST of depth h is a rooted tree with the following properties. All children of the root are at distance α^{h-1} from the root. Further, every subtree of the root is an HST of depth $h-1$ that is characterized by the same parameter α (i.e., the children 2 hops away from the root are at distance α^{h-2} from their parents). The probabilistic tree embedding result of [9] shows that for every metric space (X, d) with minimum distance normalized to 1 and for every constant $\alpha > 1$, there is a randomized construction of an HST T with a bijection f of the points in X to the leaves of T such that for every $x, y \in X$, $d(x, y) \leq d_T(f(x), f(y))$ and such that the expected tree distance $\mathbb{E}\big[d_T(f(x), f(y))\big] = O(\log |X|) \cdot d(x, y)$. Further, an efficient distributed implementation of the construction of [9] for the distances of a given network graph was given in [10].

The main technical result of this paper is an analysis of Arrow on an HST T if all requests are issued at leaves of T. Throughout the paper, the HST parameter α is set to $\alpha = 2$. For convenience, we number the levels of an HST T of depth h from 0 to h, where the level 0 nodes are the leaves and the single level h node is the root. For $\ell \in \{0, \ldots, h\}$, $\delta(\ell) := 2^{\ell+1} - 2$ denotes the distance between two leaves for which the least common ancestor is on level ℓ.

Cost Model. Assume when applying some queueing algorithm ALG to the dynamic set of requests R, the requests are ordered according to the permutation π_{ALG} such that the request ordered at position i in the order is $r_{\pi_{\mathsf{ALG}}(i)}$. For every $i \in \{1, \ldots, |R| - 1\}$, we define

the *cost of ordering* $r_{\pi_{\mathsf{ALG}}(i)}$ *after* $r_{\pi_{\mathsf{ALG}}(i-1)}$ as the time it takes the queueing algorithm ALG to enqueue the request $r_{\pi_{\mathsf{ALG}}(i)}$ as the successor of $r_{\pi_{\mathsf{ALG}}(i-1)}$. More specifically, we assume that request $r_{\pi_{\mathsf{ALG}}(i)}$ can be enqueued as soon as the predecessor request $r_{\pi_{\mathsf{ALG}}(i-1)}$ is in the system and as soon as node $v_{\pi_{\mathsf{ALG}}(i-1)}$ knows about request $r_{\pi_{\mathsf{ALG}}(i)}$. Assume that algorithm ALG informs node $v_{\pi_{\mathsf{ALG}}(i-1)}$ (through a message) about $r_{\pi_{\mathsf{ALG}}(i)}$ at time $t_{\mathsf{ALG}}(i)$. The cost (latency) $L_{\mathsf{ALG}}(r_{\pi_{\mathsf{ALG}}(i-1)}, r_{\pi_{\mathsf{ALG}}(i)})$ incurred for enqueueing request $r_{\pi_{\mathsf{ALG}}(i)}$ and the overall cost (latency) $cost_{\mathsf{ALG}}$ of ALG are then defined as follows.

$$L_{\mathsf{ALG}}(r_{\pi_{\mathsf{ALG}}(i-1)}, r_{\pi_{\mathsf{ALG}}(i)}) := \max\left\{t_{\mathsf{ALG}}(i), t_{\pi_{\mathsf{ALG}}(i-1)}\right\} - t_{\pi_{\mathsf{ALG}}(i)}, \tag{1}$$

$$cost_{\mathsf{ALG}}(\pi_{\mathsf{ALG}}) := \sum_{i=1}^{|R|-1} L_{\mathsf{ALG}}(r_{\pi_{\mathsf{ALG}}(i-1)}, r_{\pi_{\mathsf{ALG}}(i)}). \tag{2}$$

We next specify the above cost more concretely for Arrow and for an optimal offline algorithm. Assume that we have an execution $\mathcal{A}$ of the Arrow protocol that operates on a tree T. Let $\pi_{\mathcal{A}}$ be the ordering induced by the Arrow execution $\mathcal{A}$. When the "find predecessor" message of a request $r_{\pi_{\mathcal{A}}(i)}$ arrives at the node of the predecessor request $r_{\pi_{\mathcal{A}}(i-1)}$, clearly the request $r_{\pi_{\mathcal{A}}(i-1)}$ has already occurred and thus we always have $L_{\mathcal{A}}(r_{\pi_{\mathcal{A}}(i-1)}, r_{\pi_{\mathcal{A}}(i)}) = t_{\mathcal{A}}(i) - t_{\pi_{\mathcal{A}}(i)}$ for any Arrow execution. Further note, that in a synchronous execution of arrow on tree T, because Arrow always finds the predecessor on the direct path, this latency cost is always equal to the distance between the respective nodes in T.

When studying the cost of an optimal offline queueing algorithm $\mathcal{O}$, we assume that $\mathcal{O}$ knows the whole sequence of requests in advance. However, $\mathcal{O}$ still needs to send messages from each request to its predecessor request. The message delays are not under the control of the optimal offline algorithm. When lower bounding the cost of $\mathcal{O}$, we can therefore assume that all communication is synchronous even in the asynchronous case. Note that a synchronous execution is a possible strategy of the asynchronous scheduler. When operating on a graph G, the latency cost of $\mathcal{O}$ for ordering a request r_j as the successor of a request r_i is then exactly $L^G_{\mathcal{O}}(r_i, r_j) = \max\{t_i - t_j, d_G(v_i, v_j)\}$. As we analyze Arrow on an HST T that is simulated on top of an underlying network G, we directly define the optimal offline w.r.t. synchronous executions on the tree T as follows.

$$L^T_{\mathcal{O}}(r_{\pi^T_{\mathcal{O}}(i-1)}, r_{\pi^T_{\mathcal{O}}(i)}) := \max\left\{d_T(v_{\pi^T_{\mathcal{O}}(i-1)}, v_{\pi^T_{\mathcal{O}}(i)}), t_{\pi^T_{\mathcal{O}}(i-1)} - t_{\pi^T_{\mathcal{O}}(i)}\right\}, \tag{3}$$

$$cost^T_{\mathcal{O}}(\pi^T_{\mathcal{O}}) := \sum_{i=1}^{|R|-1} L^T_{\mathcal{O}}(r_{\pi^T_{\mathcal{O}}(i-1)}, r_{\pi^T_{\mathcal{O}}(i)}). \tag{4}$$

The ordering $\pi_{\mathcal{O}}$ is chosen such that the total cost $cost^T_{\mathcal{O}}(\pi_{\mathcal{O}})$ in (4) is minimized. The next lemma shows that when using the randomized HST construction of [9], the cost (4) is within a logarithmic factor of the optimal offline cost on the underlying network graph G.

▶ **Lemma 3.** *Assume T is an HST that is constructed on top of an n-node network graph G by using the randomized algorithm of [9] and assume that there is a dynamic set of queueing requests issued at the nodes of G. If the sequence of requests is independent of the randomness of the randomized HST construction, the expected optimal total cost on T (as defined in (4)) is within a factor $O(\log n)$ of the optimal offline queueing cost on G.*

Given Theorem 1 (which will be proven as the main technical result of the paper) and Lemma 3, we immediately get Theorem 2. We note in light of the remark following the statement of Theorem 1 in Section 1, the statement of Theorem 2 is also true for synchronous executions on the underlying graph G.

Manhattan Cost. In the dynamic competitive analysis of Arrow on general trees in [14], it has been shown that it is useful to study the optimal ordering w.r.t. to the following *Manhattan cost* on a tree T between two queueing requests $r_i = (v_i, t_i)$ and $r_j = (v_j, t_j)$.

$$c_{\mathcal{M}}^T(r_i, r_j) := d_T(v_i, v_j) + |t_i - t_j|. \tag{5}$$

As the cost function $c_{\mathcal{M}}(r_i, r_j)$ defines a metric space on the request set, the problem of finding an optimal ordering w.r.t. the cost $c_{\mathcal{M}}(r_i, r_j)$ is a metric TSP problem.[3] As a result, we will for example use that the total weight of an MST on the set of request w.r.t. the weight function $c_{\mathcal{M}}(r_i, r_j)$ is within a factor 2 of the cost of an optimal TSP path. The following definition is inspired by Lemma 3.12 in [14].

▶ **Definition 4** (Condensed Request Set)**.** A set R of queueing requests $r_i = (v_i, t_i)$ on a tree T is called *condensed* if for any two requests $r_i = (v_i, t_i)$ and $r_j = (v_j, t_j)$ that are consecutive w.r.t. time of occurrence, there exists requests $r_a = (v_a, t_a)$ and $r_b = (v_b, t_b)$ such that $t_a \leq t_i$, $t_b \geq t_j$, and $d_T(v_a, v_b) \geq t_b - t_a$.

It is shown in [14] that for condensed request sets, the total optimal Manhattan cost is within a constant factor of the optimal offline queueing cost.

▶ **Lemma 5** (Lemma 3.17 in [14] rephrased)**.** *If the request set R is condensed, then on any tree T and for every ordering π on the requests, it holds that*

$$\sum_{i=1}^{|R|-1} c_{\mathcal{M}}^T(r_{\pi(i-1)}, r_{\pi(i)}) \leq 12 \cdot \sum_{i=1}^{|R|-1} L_{\mathcal{O}}^T(r_{\pi(i-1)}, r_{\pi(i)}).$$

For synchronous executions on trees, it is also shown in [14] that every request set R can be transformed into a condensed request set without changing the ordering (and the cost) of Arrow and without increasing the optimal offline cost.

▶ **Lemma 6** (Lemma 3.11 in [14] rephrased)**.** *Let R be a set of queueing requests issued on a tree T and let $r_i = (v_i, t_i)$ and $r_j = (v_j, t_j)$ be two requests of R that are consecutive w.r.t. time of occurrence. Further, choose two requests $r_a = (v_a, t_a)$ with $t_a \leq t_i$ and $r_b = (v_b, t_b)$ with $t_b \geq t_j$ minimizing $\delta := t_b - t_a - d_T(v_a, v_b)$. If $\delta > 0$, every request $r = (v, t)$ with $t \geq t_j$ can be replaced by a request $r' = (v, t - \delta)$ without changing the synchronous* Arrow *order and without increasing the optimal offline cost.*

Lemma 6 implies that every request set R can be transformed into a condensed set R' without changing the synchronous order of Arrow and without increasing the optimal offline cost. For the analysis of Arrow in synchronous systems, we can thus w.l.o.g. assume that the request set is condensed.

3 Analysis of the Optimal Offline Cost

This and the next section discuss the main technical contribution of the paper and analyzes the total cost of a synchronous Arrow execution when run on an HST T. Throughout this section, we assume that a fixed HST T, a set of dynamic requests R placed at the leaves of T, and a synchronous execution of Arrow with request set R on T are given. For convenience, we

[3] The relation of Arrow and the TSP problem was already exploited in [14] when analyzing Arrow on general trees.

relabel the requests in R so that they are ordered according to the queueing order resulting from the given Arrow execution on T. That is, we assume that for all $i \in \{0, \ldots, |R|-1\}$, request $r_i = (v_i, t_i)$ is the i^{th} request in Arrow's order. Note that $r_0 = (v_0, 0)$ is still the dummy request defining the initial tail of the queue. As discussed in Section 2, the Arrow order can be seen as a greedy ordering in the following sense. Given the first $i-1$ requests in the order, the i^{th} request r_i is a request $r = (v, t)$ from the subset of the remaining requests that can reach the node v_{i-1} of request r_{i-1} first immediately sending a message at time t from node v to node v_{i-1}. This greedy behavior is captured by the following basic lemma. For a more thorough discussion, we refer to [14].

▶ **Lemma 7.** *Consider a synchronous execution of* Arrow *on tree T and consider two arbitrary requests r_i and r_j for which $1 \leq i < j$ (i.e., r_j is ordered after r_i by* Arrow*). Then it holds that*

1. $t_i + d_T(v_{i-1}, v_i) \leq t_j + d_T(v_{i-1}, v_j)$ *and*
2. $t_i \leq t_j + d_T(v_i, v_j)$.

Before delving into the details of the analysis, we give a short outline. In the first step in Section 3.1, we study the ordering generated by Arrow in more detail and show that it implies a hierarchical partition of the requests R in a natural way. To simplify the next step, Section 3.2 transforms the given HST T into a new tree such that inside each subtree, if ordering the request by time of occurrence, the gap between the times of consecutive requests cannot be too large (whenever such a gap is too large, we split the corresponding subtree into two trees). Section 3.3 then shows that the optimal offline cost can be characterized by the total Manhattan cost of a spanning tree that respects the hierarchical structure of the HST T in a best given way. Finally, in Section 4, we give a general framework to compare the queueing cost of an online distributed algorithm on an HST T to the optimal offline cost on T and we apply this method to synchronous Arrow executions.

3.1 Characterizing Arrow By A Hierarchical Partition of R

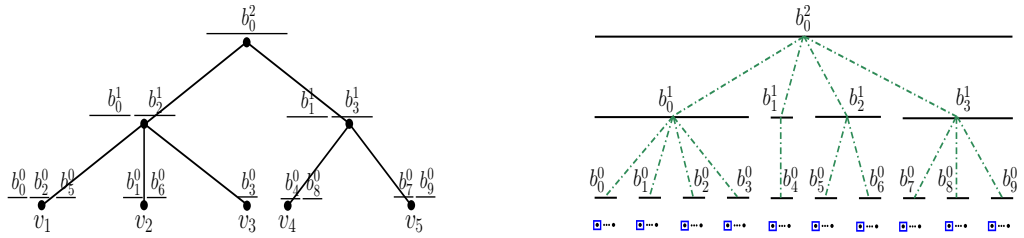
We hierarchically partition the requests R according to the Arrow queueing order and the hierarchical structure of the HST T. On each level ℓ of T, we partition the requests into blocks, where a block of requests is a maximal set of requests that are ordered consecutively by Arrow inside some level-ℓ subtree of T. In the following, for non-negative integers s and t, we use the abbreviations $[s] := \{0, \ldots, s-1\}$ and $[s, t] := \{s, \ldots, t\}$. Formally, instead of partitioning the set of requests R directly, we partition the set of indexes $[|R|]$. Recall that the requests in R are indexed consecutively according to the queueing order of Arrow.

▶ **Definition 8** (Hierarchical Block Partition)**.** For each level $\ell \in [0, h]$, we partition $[|R|]$ into $n(\ell)$ blocks $\left\{b_0^\ell, b_1^\ell, \cdots, b_{n(\ell)-1}^\ell\right\}$ such that

1. each block is a consecutive set of integers (i.e., a consecutively ordered set of requests),
2. for every block b_i^ℓ, all requests r_p for $p \in b_i^\ell$ are in the same level-ℓ subtree of T, and
3. for all $i, j \in [n(\ell)]$ and all $p \in b_i^\ell$ and $q \in b_j^\ell$, $i < j \implies p < q$.

For each block b, we define the first request of b to be the one with min. index in b.

Note that for each level ℓ and for the first block of this level, the first request of the block has index 0. The block partition defined in Definition 8 is illustrated in Figure 1. Figure 1a shows the blocks within the HST structure, whereas Figure 1b shows the hierarchical partition induced by the blocks. To simplify the presentation of our analysis, we also define a level -1 block b_i^{-1} for each individual request r_i. Note that we have $n(-1) = |R|$. The following definition allows us to navigate through the block hierarchy.



(a) Blocks within the same subtree. **(b)** Tree induced by the block hierarchy.

Figure 1 The partition of R. (a) An HST with height 2 and 5 leaves. The leaves issue requests at different times. The issued requests by nodes v_1, v_2, and v_3 are partitioned into the blocks b_0^1 and b_2^1 on level 1. These two blocks are called neighbor blocks at a subtree rooted at height 1. (b) The corresponding 4 level-wise partition based on Arrow's order that forms a parent-child relation between the blocks on different levels. Blue boxes include the requests that are ordered first by Arrow among all requests in blocks b_i^0 for all $i \in [0, 9]$.

▶ **Definition 9** (Children Blocks)**.** The set of children blocks of a block b_i^ℓ on a level $\ell \in [0, h]$ is defined as $child(b_i^\ell) := \left\{b_j^{\ell-1} : b_j^{\ell-1} \subseteq b_i^\ell\right\}$. Block b_i^ℓ is called the parent block of each of the blocks in $child(b_i^\ell)$.

In Figure 1b, block b_2^1 is the parent block of its children blocks b_5^0 and b_6^0. Block b_1^1 has only one child block b_4^0 and thus $b_1^1 = b_4^0$.

The blocks $\left\{b_0^\ell, b_1^\ell, \cdots, b_{n(\ell)-1}^\ell\right\}$ of level ℓ belong to the subtrees rooted at height ℓ of the HST T. Note that by the definition of the block partition, no two consecutive blocks at the same level ℓ belong to the same level-ℓ subtree of T. The next definition specifies notation to argue about blocks of the same subtree of T.

▶ **Definition 10** (Blocks of Same Subtree)**.** If two blocks b_i^ℓ and b_j^ℓ belong to the same level-ℓ subtree of T, this is denoted by $\widehat{b_i^\ell b_j^\ell}$. Moreover, $|\widehat{b_i^\ell b_j^\ell}| := \left|\left\{w : i < w < j \ \wedge\ \widehat{b_i^\ell b_w^\ell} \text{ holds}\right\}\right|$. Two blocks b_i^ℓ and b_j^ℓ are called *neighbor blocks* if $\widehat{b_i^\ell b_j^\ell}$ and $|\widehat{b_i^\ell b_j^\ell}| = 0$.

In Figure 1a, blocks b_0^0, b_2^0, and b_5^0 are within the same subtree rooted at node v_1. Blocks b_0^0 and b_5^0 are not neighbor blocks, however blocks b_0^0 and b_2^0, as well as blocks b_2^0 and b_5^0 are neighbor blocks. The next lemma lists a number of simple properties of the block partition.

▶ **Lemma 11.** *The hierarchical block partition of Def. 8 satisfies the following properties:*

1. *For every block b_i^ℓ and for all $p, q \in b_i^\ell$, we have $d_T(v_p, v_q) \leq \delta(\ell)$.*
2. *For each level ℓ and all level-ℓ blocks b_i^ℓ and b_j^ℓ, if $\widehat{b_i^\ell b_j^\ell}$ holds, for any $p \in b_i^\ell$ and $q \in b_j^\ell$, we have $d_T(v_p, v_q) \leq \delta(\ell)$.*
3. *For each level ℓ and all level-ℓ blocks b_i^ℓ and b_j^ℓ, if $\widehat{b_i^\ell b_j^\ell}$ does* not *hold, for all $p \in b_i^\ell$ and $q \in b_j^\ell$, we have $d_T(v_p, v_q) \geq \delta(\ell + 1)$.*
4. *Assume $\ell < h$ and consider two blocks b_i^ℓ and b_j^ℓ that have a common parent block $b_w^{\ell+1}$, but for which $\widehat{b_i^\ell b_j^\ell}$ does not hold. Then, for all $p \in b_i^\ell$ and $q \in b_j^\ell$, we have $d_T(v_p, v_q) = \delta(\ell+1)$.*

We have seen that in a synchronous Arrow execution, the latency cost for ordering request r_{i+1} as the successor of r_i is exactly the distance $d_T(v_i, v_{i+1})$ between the nodes of the two requests. The total cost of Arrow therefore directly follows from the structure of the block partition.

▶ **Lemma 12.** *The total cost of a synchronous* Arrow *execution on the HST T with corresponding hierarchical block partition is given by*

$$cost_{\mathcal{A}}(\pi_{\mathcal{A}}) = \sum_{\ell=0}^{h-1} \big(n(\ell) - n(\ell+1)\big) \cdot \delta(\ell+1).$$

3.2 HST Conversion

We next provide a recursive (top-down) splitting procedure that converts the original HST into a new HST with better properties. The conversion does not change the total cost of ordering the requests by Arrow (in fact, it does not change the block partition). Further, the total Manhattan cost of optimal offline algorithm's order asymptotically remains unchanged as well. We describe how the splitting procedure works and we then argue its properties.

Splitting Procedure. We describe the splitting procedure as it is applied to a subtree T' that is rooted at a given level $\ell \in \{0, \dots, h\}$ of T. If $\ell = 0$, the tree T' is returned unchanged. Otherwise ($\ell \geq 1$), we go through all level-$(\ell-1)$ subtrees T'' of T'. As long as the tree T'' has two neighbor blocks $b_i^{\ell-1}$ and $b_j^{\ell-1}$ (for $i < j$) for which the following condition (6) is true, the subtree T'' is split into two separate subtrees T_1'' and T_2'' of T'.

$$t_{\min}(b_j^{\ell-1}) - t_{\max}(b_i^{\ell-1}) \geq \delta(\ell). \tag{6}$$

The splitting of T'' into T_1'' and T_2'' works as follows. The topology of T_1'' and T_2'' is identical to the topology of T''. Each request $r = (v, t)$ that is issued at some node v of T'' is either placed on the isomorphic copy of v in T_1'' or in T_2''. All requests r in blocks $b_x^{\ell-1}$ of T'' for $x \leq i$ are placed in tree T_1'' and all request in blocks $b_y^{\ell-1}$ of T'' for $y \geq j$ are placed in tree T_2''. We perform such splittings for trees T' of level ℓ as long as there are subtrees of T' on level $\ell - 1$ with neighbor blocks that satisfy Condition (6). As soon as no such neighbor blocks exist, the procedure is applied recursively to all trees T'' at level $\ell - 1$ (incl. the new subtrees). The conversion is started by applying the procedure to the complete HST T.

▶ **Lemma 13.** *The above splitting procedure does not change the hierarchical block partition and it thus also preserves* Arrow*'s queueing order $\pi_{\mathcal{A}}$ and its total cost $cost_{\mathcal{A}}(\pi_{\mathcal{A}})$.*

The next lemma shows that if a tree T'' is split into two trees T_1'' and T_2'' such that all requests in T_1'' are ordered before all requests in T_2'', there is a significant time of occurrence gap between the requests ending up in subtrees T_1'' and T_2''.

▶ **Lemma 14.** *Assume that we are performing a single splitting. Further, assume that we are working on a tree T' on level ℓ and that we are splitting a subtree T'' of T' into T_1'' and T_2'' such that T_1'' obtains the blocks that are scheduled first by* Arrow*. If R_1 and R_2 are the request sets of T_1'' and T_2'', respectively, we have $t_{\min}(R_2) - t_{\max}(R_1) \geq \delta(\ell) - \delta(\ell-1)$.*

It remains to show that the splitting also does not affect the optimal offline cost in a significant way. The following lemma shows that the Manhattan cost $c_{\mathcal{M}}(r, r')$ for any two requests r and r' can increase by at most a factor 3. Hence, also the total Manhattan cost of an optimal ordering cannot increase by more than a factor 3.

▶ **Lemma 15.** *For any two requests r and r', the splitting procedure does not increase the Manhattan cost $c_{\mathcal{M}}(r, r')$ by more than a factor 3.*

For the remainder of the analysis, we assume that the HST T is an HST that is obtained after applying the splitting procedure recursively. We therefore assume that for every level ℓ and every subtree T' at level ℓ, there is no level-$(\ell-1)$ subtree T'' of T' that contains two neighbor blocks that satisfy Condition (6).

3.3 Lower Bounding The Optimal Manhattan Cost

In this section, we construct a tree S^* that spans all requests in R. The tree S^* has a nice hierarchical structure: For each subtree T' of T, the set of edges of S^* induced by the request set of the subtree T' forms a spanning tree of the request set of T'. Apart from this useful structural property, we will show that the total Manhattan cost of the spanning tree S^* is within a constant factor of minimum spanning tree (MST) of the request set R w.r.t. the Manhattan cost. We have seen that on condensed request sets, the optimal TSP path of the request set w.r.t. the Manhattan cost is within a constant factor of the optimal offline queueing cost. Note that because any TSP path is also a spanning tree, this implies that the total Manhattan cost of the MST and thus also the total Manhattan cost of the tree S^* are lower bounding the optimal offline queueing cost within a constant multiplicative factor.

For convenience, we add one more level to the HST T. Instead of placing the requests at the leaves on level 0, we assume that each level 0 node v has a child node on level -1 for each of the requests issued at node v. Hence, the new leaf nodes are on level -1 and each leaf node receives exactly one request.[4] The distance between a level -1 node and its parent on level 0 is set to be 0.

Spanning Tree Construction. The spanning tree S^* is constructed greedily in a bottom-up fashion. For each subtree T' of T, we recursively define a tree $S^*(T')$ as follows. For the leaf nodes on level -1, the tree consists of the single request placed at the node. For a tree T' rooted at a node v on level $\ell \geq 0$, the tree $S^*(T')$ consists of the recursively constructed trees $S^*(T_1''), S^*(T_2''), \ldots$ of the subtrees $T_1'', T_2'', \ldots$ of T'' and of edges connecting the trees $S^*(T_1''), S^*(T_2''), \ldots$ to a spanning tree of the set of requests issued at leaves of tree T'. The edges for connecting the trees $S^*(T_1''), S^*(T_2''), \ldots$ are chosen so that they have minimum total Manhattan cost. That is, to connect the trees $S^*(T_1''), S^*(T_2''), \ldots,$ we compute an MST of the graph we get if each of the trees $S^*(T_i'')$ is contracted to a single node. We can therefore for example choose the edges to connect the trees $S^*(T_1''), S^*(T_2''), \ldots$ in e greedy way: Always add the lightest (w.r.t. Manhattan cost) edge that does not close a cycle with the already existing edges, including the edges of the trees $S^*(T_1''), S^*(T_2''), \ldots.$

MST Approximation. In the following, it is shown that the total Manhattan cost of the tree $S^* = S^*(T)$ is within a constant factor of the cost of an MST w.r.t. the Manhattan cost. Where convenient, we identify a tree τ with its set of edges, i.e., we also use S^* to denote the set of edges of the tree S^*. Further, the cost of an edge $e = \{r, r'\}$ is the Manhattan cost $c_{\mathcal{M}}(r, r')$. We also slightly abuse notation and use $c_{\mathcal{M}}(e)$ to denote this cost. The proof applies a general MST approximation result that appears as Theorem A.1 in the full version [11]. Together with the following lemma, Theorem A.1 of [11] directly implies that the total Manhattan cost of S^* is within a factor 4 of the MST Manhattan cost. For a subtree T' of T, we use $R(T')$ to denote the subset of the requests in R that are issued at nodes of T'.

[4] Note that subtrees of T that do not have any queueing requests can be ignored and therefore, we can w.l.o.g. assume that every leaf node issues some queueing request.

▶ **Lemma 16.** *Consider the constructed spanning tree S^* and consider an arbitrary edge e of S^*. Let S_1^* and S_2^* the two subtrees that result when removing edge e from S^*. Further, assume e^* be an edge that connects the two subtrees S_1^* and S_2^* and that has minimum Manhattan cost among all such edges. We then have $c_{\mathcal{M}}(e) \leq 4 \cdot c_{\mathcal{M}}(e^*)$.*

▶ **Corollary 17.** *The total Manhattan cost of the spanning tree S^* is at most 4 times the total Manhattan cost of an MST spanning all the requests.*

4 Analysis of the Online Queueing Cost

In this section, we give a general framework to compare the queueing cost of an online queueing algorithm on HST T with the bound of the offline queueing cost as established in Section 3. At the end of the section, we apply the method to analyze synchronous Arrow executions on T. As in Section 3.3, for convenience, we add one more level to the HST T so that each level 0 node v has a child node on level -1 for each of the requests issued at node v. The new leaf nodes are on level -1 and each leaf node receives exactly one request.

We first state two basic locality properties of Arrow. We will then show that those properties are sufficient to prove a constant competitive ratio compared to the optimal offline queueing cost on T. We define the notion of a *distance-respecting queueing order* and the notion of *distance-respecting latency cost* of a queueing algorithm.

▶ **Definition 18** (Distance-Respecting Order)**.** Let R be a set of requests $r_i = (v_i, t_i)$ issued at the nodes of a tree T and let π be permutation on $[0, |R| - 1]$. The ordering $r_{\pi(0)}, r_{\pi(1)}, \ldots, r_{\pi(|R|-1)}$ induced by π is called *distance-respecting* if whenever $\pi(i) < \pi(j)$, we have $t_i - t_j \leq d_T(v_i, v_j)$.

▶ **Definition 19** (Distance-Respecting Latency Cost)**.** An online distributed queueing algorithm ALG is said to have *distance-respecting latency cost* if for any request set R and any possible queueing order π_{ALG} of ALG, for all $1 \leq i < j < |R|$, it holds that

$$t_{\pi_{\mathsf{ALG}}(i)} + L_{\mathsf{ALG}}(r_{\pi_{\mathsf{ALG}}(i)}, \pi_{\mathsf{ALG}}(i-1)) \leq t_{\pi_{\mathsf{ALG}}(j)} + d_T(v_{\pi_{\mathsf{ALG}}(j)}, v_{\pi_{\mathsf{ALG}}(i-1)}).$$

4.1 Constructing a Spanning Tree

As the first part of the online queueing cost analysis, we construct a new tree $\mathbb{S}$ that spans all requests in R. It will be shown that the total Manhattan cost of $\mathbb{S}$ asymptotically equals the total Manhattan cost of the tree S^* constructed in the previous section.

We construct a new tree $\mathbb{S}$ on R based on an ordering π of the set of requests. We assume that the ordering of the requests given by π is $r_{\pi(0)}, r_{\pi(1)}, \ldots, r_{\pi(|R|-1)}$. For each index i with $i \in [0, |R| - 2]$, we define the *local successor* as

$$next(i) := \min \left\{ j \in [i+1, |R|-1] \,:\, d_T(v_{\pi(i)}, v_{\pi(j)}) = \min_{k \in [i+1, |R|-1]} d_T(v_{\pi(i)}, v_{\pi(k)}) \right\}. \quad (7)$$

Hence, among the requests ordered after $r_{\pi(i)}$ by order π, $next(i)$ is the position of a request in the order π with minimum tree distance to $v_{\pi(i)}$ and among those, of the first one ordered by π. Note that this means that for all requests $r_{\pi(k)}$ for which $i < k < next(i)$, we have $d_T(v_{\pi(i)}, v_{\pi(k)}) > d_T(v_{\pi(i)}, v_{\pi(next(i))})$ and for all requests $r_{\pi(k)}$ for which $k \geq next(i)$, we have $d_T(v_{\pi(i)}, v_{\pi(k)}) \geq d_T(v_{\pi(i)}, v_{\pi(next(i))})$.

The spanning tree $\mathbb{S}$ is constructed as follows. For every request $r_{\pi(i)}$ for all $i \in [0, |R| - 2]$, we add the edge $\{r_{\pi(i)}, r_{\pi(next(i))}\}$ to the tree $\mathbb{S}$. Note that $\mathbb{S}$ is indeed a spanning tree: If

directing each edge from $r_{\pi(i)}$ to $r_{\pi(next(i))}$, each node has out-degree 1 and we cannot have cycles because $next(i) > i$. The following observation shows that in addition, $\mathbb{S}$ has the same useful hierarchical structure as the tree S^* constructed in Section 3.3.

▶ **Observation 20.** *As the tree S^*, also the tree $\mathbb{S}$ has the property that for any subtree T' of T, the subgraph of $\mathbb{S}$ induced by only the requests at nodes in T' is a connected subtree of $\mathbb{S}$. This follows directly from the definition of the local successor $r_{\pi(next(i))}$. Except for the last ordered request inside T', the local successor of any other request of T' is inside T' (because the local successor is a request with minimum tree distance).* ◀

In light of Observation 20, for any subtree T' of T, we use $\mathbb{S}(T')$ to denote the subtree of $\mathbb{S}$ induced by the requests issued at nodes in T'.

4.2 Bounding the Manhattan Cost of the Spanning Tree

The following lemma shows that if the spanning tree $\mathbb{S}$ is constructed by using a distance-respecting ordering π, the total Manhattan cost of the spanning tree $\mathbb{S}$ is asymptotically equal the total Manhattan cost of S^*.

▶ **Lemma 21.** *Let $C_{\mathcal{M}}(\mathbb{S})$ and $C_{\mathcal{M}}(S^*)$ be the total Manhattan costs of $\mathbb{S}$ and of S^*. If the tree $\mathbb{S}$ is constructed using a distance-respecting ordering π, we have $C_{\mathcal{M}}(\mathbb{S}) \leq 3 \cdot C_{\mathcal{M}}(S^*)$.*

4.3 Bounding the Total Latency Cost

It remains to prove the main claim and show that the total online queueing cost on the HST T is within a constant factor of the optimal offline cost on T. The following theorem states that this is generally true for algorithms with distance-respecting latency cost (Def. 19) and which produce distance-respecting queueing orders (Def. 18), as long as the request set R is condensed (Def. 4).

▶ **Theorem 22.** *Assume that we are given an HST T and a condensed set of requests issued at the leaves of R. Further, assume that we are given a distributed queueing algorithm* ALG *that has distance-respecting latency cost and that always produces a distance-respecting queueing order π. Then, the total latency cost of* ALG *is within a constant factor of the optimal offline cost on T.*

▶ **Corollary 23.** *The total latency cost of a synchronous execution of* Arrow *on an HST T is within a constant factor of the optimal offline queueing cost on T.*

▶ Remark. The above corollary proves Theorem 1 (cf. Section 1) for synchronous executions on the HST T. The full statement of Theorem 1 for general asynchronous executions is proven in the full version of the paper [11]. There, it is shown that also for asynchronous executions, Arrow has distance-respecting latency cost and produces distance-respecting queueing orders. In addition, we also show that we can still restrict attention to condensed request sets. The claim of Theorem 1 for the asynchronous case then follows from Theorem 22 in the same way as in the above corollary.

References

1 I. Abraham, D. Dolev, and D. Malkhi. LLS: a locality aware location service for mobile ad hoc networks. In *D-POMC co-located 10th C. Mobicom*, pages 75–84, 2004.

2 H. Attiya, V. Gramoli, and A. Milani. A provably starvation-free distributed directory protocol. In *Proc. of the 12th Symp. on Self-Stabilizing Syst. (SSS)*, pages 405–419, 2010.

3 B. Awerbuch and D. Peleg. Sparse partitions. In *Proc. 31st Symp. Foundations of Computer Science (FOCS)*, pages 503–513, 1990.

4 Baruch Awerbuch and David Peleg. Online tracking of mobile users. *Journal of the ACM (JACM)*, 42(5):1021–1058, 1995.

5 Y. Bartal. Probabilistic approximations of metric spaces and its algorithmic applications. In *Proc. 37th Symp. on Foundations of Computer Science (FOCS)*, pages 184–193, 1996.

6 D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal. Directory-based cache coherence in large-scale multiprocessors. *Computer*, 23(6):49–58, 1990.

7 M. Demirbas et al. A hierarchy-based fault-local stabilizing algorithm for tracking in sensor networks. In *P. 8th I. C. on Princ. of Dist. Syst. (OPODIS)*, pages 299–315, 2004.

8 Michael J Demmer and Maurice P Herlihy. The arrow distributed directory protocol. In *Proc. of the 12th Symp. on Dist. Comp. (DISC)*, pages 119–133, 1998.

9 J. Fakcharoenphol, S. Rao, and K. Talwar. A tight bound on approximating arbitrary metrics by tree metrics. In *Proc. 35th S. on Th. of Comp. (STOC)*, pages 448–455, 2003.

10 Mohsen Ghaffari and Christoph Lenzen. Near-optimal distributed tree embedding. In *International Symposium on Dist. Comp.*, pages 197–211, 2014.

11 Abdolhamid Ghodselahi and Fabian Kuhn. Dynamic analysis of the arrow distributed directory protocol in general networks. *arXiv preprint arXiv:1705.07327*, 2017.

12 David Ginat, Daniel D Sleator, and Robert E Tarjan. A tight amortized bound for path reversal. *Information Processing Letters*, 31(1):3–5, 1989.

13 A. Gupta. Steiner poi,nts in tree metrics don't (really) help. In *Proc. 12th Symp. on Discrete Algorithms (SODA)*, pages 220–227, 2001.

14 M. Herlihy, F. Kuhn, S. Tirthapura, and R. Wattenhofer. Dynamic analysis of the arrow distributed protocol. *Theory of Comp. Syst. (TCS)*, 39(6):875–901, 2006.

15 M. Herlihy, S. Tirthapura, and R. Wattenhofer. Competitive concurrent distributed queuing. In *Proc. of the 20th Symp. on Princ. of Dist. Comp. (PODC)*, pages 127–133, 2001.

16 Maurice Herlihy. The Aleph toolkit: Support for scalable distributed shared objects. In *Proc. 3rd Workshop on Comm. Arch. and Appl. for Network-Based Parallel Comp. (CANPC)*, pages 137–149, 1999.

17 Maurice Herlihy and Ye Sun. Distributed transactional memory for metric-space networks. *Distributed Computing*, 20(3):195–208, 2007.

18 Maurice Herlihy, Srikanta Tirthapura, and Rogert Wattenhofer. Ordered multicast and distributed swap. *ACM SIGOPS Operating Syst. Rev. (OSR)*, 35(1):85–96, 2001.

19 Maurice Herlihy and Michael P Warres. A tale of two directories: implementing distributed shared objects in java. In *Proc. of the ACM Conf. on Java Grande*, pages 99–108, 1999.

20 Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. on Computer Syst. (TOCS)*, 7(4):321–359, 1989.

21 Mohamed Naimi and Michel Trehel. An improvement of the $\log n$ distributed algorithm for mutual exclusion. In *Proc. 7th Conf. on Distr. Comp. Sys. (ICDCS)*, pages 371–377, 1987.

22 D. Peleg and E. Reshef. A variant of the arrow distributed directory with low average complexity. In *Proc. of the 26th Int. Coll. on A. L. and P. (ICALP)*, pages 615–624, 1999.

23 Y. Rabinovich and R. Raz. Lower bounds on the distortion of embedding finite metric spaces in graphs. *Discrete and Computational Geometry*, (19), 1998.

24 Kerry Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Trans. on Computer Syst. (TOCS)*, 7(1):61–77, 1989.

25 R. Rosenkrantz, R. Stearns, and P. Lewis. An analysis of several heuristics for the traveling salesman problem. *SIAM J. on Computing*, 6(3):563–581, 1977.

26 Gokarna Sharma and Costas Busch. Distributed transactional memory for general networks. *Distributed Computing*, 27(5):329–362, 2014.

27 Gokarna Sharma and Costas Busch. An analysis framework for distributed hierarchical directories. *Algorithmica*, 71(2):377–408, 2015.

28 Srikanta Tirthapura and Maurice Herlihy. Self-stabilizing distributed queuing. *IEEE Trans. on Parallel and Dist. Syst. (PDS)*, 17(7):646–655, 2006.

29 Jan L. A. van de Snepscheut. Fair mutual exclusion on a graph of processes. *Distributed Computing*, 2(2):113–115, 1987.

30 B. Zhang and B. Ravindran. Dynamic analysis of the relay cache-coherence protocol for distributed transactional memory. In *Proc. of the 24th IPDPS*, pages 1–11, 2010.

Consistency Models with Global Operation Sequencing and their Composition

Alexey Gotsman[*1] **and Sebastian Burckhardt**[2]

1 **IMDEA Software Institute, Madrid, Spain**
2 **Microsoft Research, Redmond, USA**

Abstract

Modern distributed systems often achieve availability and scalability by providing consistency guarantees about the data they manage weaker than linearizability. We consider a class of such consistency models that, despite this weakening, guarantee that clients eventually agree on a global sequence of operations, while seeing a subsequence of this final sequence at any given point of time. Examples of such models include the classical Total Store Order (TSO) and recently proposed dual TSO, Global Sequence Protocol (GSP) and Ordered Sequential Consistency.

We define a unified model, called *Global Sequence Consistency (GSC)*, that has the above models as its special cases, and investigate its key properties. First, we propose a condition under which multiple objects each satisfying GSC can be composed so that the whole set of objects satisfies GSC. Second, we prove an interesting relationship between special cases of GSC–GSP, TSO and dual TSO: we show that clients that do not communicate out-of-band cannot tell the difference between these models. To obtain these results, we propose a novel axiomatic specification of GSC and prove its equivalence to the operational definition of the model.

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases Consistency conditions, Weak memory models, Compositionality

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.23

1 Introduction

Modern distributed systems often achieve availability and scalability by providing consistency guarantees about the data they manage weaker than the gold standard of linearizability [16]. In this paper we consider a class of such consistency models that, despite this weakening, guarantee *global operation sequencing*: clients eventually agree on a global sequence of operations, while seeing a subsequence of this final sequence at any given point of time. An implementation of a service providing such a model may consist of a single *server* and multiple *clients*, each maintaining a replica of the data managed by the service. Clients accept operations from end-users, evaluate them on their local (possibly stale) data replica and forward the operations to the server. The server arranges all received operations into a totally ordered log and forwards them to clients in the order determined by the log. The server log thus establishes the desired global sequence of operations.

Such consistency models arise in different domains. For instance, clients may correspond to mobile devices, cloud servers or processor cores; the role of the server may be played by an elected leader, a replicated state machine [26], a reliable total-order broadcast [11] or the memory subsystem in a multiprocessor architecture [28]. Various models differ in whether

* Alexey Gotsman was supported by an ERC Starting Grant RACCOON.


© Alexey Gotsman and Sebastian Burckhardt;
licensed under Creative Commons License CC-BY

Table 1 Specialising GSC.

Implicit fences	pull	push
GSP [10]	no	no
TSO [24, 23]	yes	no
dual TSO [2]	no	yes
OSC [22]	updates	yes
linearizability [16]	yes	yes

the propagation of operations from clients to the server and vice versa is asynchronous or synchronous. Thus, in the *Global Sequence Protocol (GSP)* model [10], the propagation is asynchronous in both directions, which allows clients to execute operations even if they get partitioned from the server [14]. This model is implemented in Microsoft's TouchDevelop system for mobile app programming, to support offline access [1], and in the Orleans actor framework [6], to support geo-replication [5]. In the *Total Store Order (TSO)* model [24, 23], implemented by SPARC and x86 multiprocessors, operation propagation from clients to the server is asynchronous, but the one from the server to clients is synchronous: clients *pull* all new operations from the server before evaluating each operation. Conversely, in the *dual TSO* model [2] operation propagation from the server to clients is asynchronous, but the one from the clients to the server is synchronous: clients *push* operations to the server immediately after they are executed. If we strengthen dual TSO by requiring that all update operations are propagated synchronously in both directions, we obtain *Ordered Sequential Consistency (OSC)* [22], which captures the semantics of coordination services such as ZooKeeper [18]. Finally, we obtain linearizability [16] when operation propagation is synchronous in both directions.

In this paper we study key properties of the consistency models from the above class. To this end, we consider a flexible model, called *Global Sequence Consistency (GSC)*, that has the above models as its special cases and obtain novel results about this model: a condition for safely composing multiple GSC services and a certain interesting relationship between the model's special cases. The GSC model is defined by the above client-server protocol where operation propagation is by default asynchronous, but operations may include two kinds of *fences*. The fences respectively force a client to pull all new operations from the server or push all outstanding local operations to the server (§3). Then we obtain various existing consistency models by systematically associating fences with operations as shown in Table 1.

Like sequential consistency [20], GSC is not *composable* (aka *local*) [16]: objects satisfying GSC may fail to provide this consistency guarantee when combined. This is a problem because application programmers often want to distribute objects among multiple services, e.g., to place them in geographical locations where they are most likely to be updated and thereby minimise latency [21]. Non-composability does not allow programmers to easily predict the behavior of such a system. This is a particular issue in the Orleans implementation of geo-replication [5], which guarantees GSP only for each individual object.

To address this problem, we propose a condition under which multiple objects each satisfying GSC can be composed so that the whole set of objects satisfies GSC (§5). Informally, the condition requires using fences according to the following discipline: when switching between different objects, a client has to push the operations done on the old object and pull operations on the new object. Our result ensures that in this case clients interacting with multiple GSC services implementing different objects will behave as though they are interacting with a single GSC service. This result holds even when clients can communicate

out-of-band, without using the GSC services. As its special cases, we obtain novel conditions for composing TSO and dual TSO objects, as well as a recently proposed condition for OSC [22, 21].

We also prove an interesting relationship between special cases of GSC– GSP, TSO and dual TSO (§4): we show that clients that do not communicate out-of-band cannot tell the difference between them. In particular, this result implies that a program without out-of-band communication written assuming TSO operates correctly under much weaker, fully asynchronous GSP. This equivalence has been previously conjectured without proof [10]; the present paper confirms this conjecture. Assuming the absence of out-of-band communication is common for memory models, where clients are processors that do not communicate directly. However, this assumption is often not appropriate for distributed interactive applications, where clients can have external means of communication. In this setting, the above special cases of GSC are observably different.

Proving the above results about compositionality and equivalence is nontrivial due to the complexity of reasoning about the distributed protocol implementing GSC. Our main tool in tackling this complexity is an *axiomatic* specification of GSC, given in the style often used for consistency models in shared-memory [19] and distributed storage systems [9, 8] (§6). The specification represents service executions using several relations, declaratively describing how operations are processed by the GSC protocol; the consistency model is then defined by a set of axioms, constraining these relations. We prove that our axiomatic specification is equivalent to the operational one. A particular subtlety in formulating the axiomatic specification and proving this equivalence is the need for the specification to track the *real-time order* between operations, determining when one operation finishes before another one starts. This makes results established using the axiomatic specification applicable in the case when clients can communicate out-of-band [12, 3].

The axiomatic specification of GSC is instrumental in obtaining our results. A recurring challenge is to prove the existence of an execution that satisfies some conditions, e.g., is a composition of single-object executions in the proof of the compositionality criterion (§8). Constructing the desired execution is difficult to do directly on the operational model. Because of the wide-ranging effect of fences, such an execution cannot be obtained simply by local reordering of independent steps, as with simpler operational models. But via the axiomatic specification of GSC, we can solve this problem indirectly by formulating constraints on precedence of events in the execution as relations and then using algebraic techniques to prove that their union is acyclic, which guarantees that there exists an execution satisfying them. We hope that, in the future, the GSC model, with its two equivalent definitions, and our proof techniques will provide a solid foundation for obtaining further results about consistency models with global operation sequencing.

2 Preliminaries

We consider a distributed service managing a collection of *objects* $\mathsf{Obj} = \{x, y, \ldots\}$. A finite number of clients interact with the service by performing *operations* on the objects, which are ranged over by *op* and come from a set Op. Parameters of operations, if any, are part of the operation name. For uniformity, we assume that all objects admit the same set of operations and that each operation returns one value from a set Val; we can use a special member of Val to model operations that return no value. The sequential semantics of operations is defined by a function $\mathsf{eval} : \mathsf{Op}^* \times \mathsf{Op} \to \mathsf{Val}$ that determines the return value of an operation on an object given the sequence of operations previously executed on this object.

The consistency model provided by the service defines the set of all possible interactions between the service and its clients. We now introduce a structure that records such interactions in a single computation, called a *history*. In it we denote client-service interactions using *events*, which are ranged over by e, f, g and come from an infinite countable set Event. Events have unique identifiers from a set Id. An event is of the form $e = (\iota, x, op, a, fen)$, where $\iota \in \mathsf{Id}$ is the event identifier, $x \in \mathsf{Obj}$ is the object on which the event occurs, $op \in \mathsf{Op}$ is the operation done, $a \in \mathsf{Val}$ is its return value, and $fen \subseteq \{\mathsf{push}, \mathsf{pull}\}$ gives the *fences* requested by the client. We use $\mathsf{obj}(e)$, $\mathsf{oper}(e)$, $\mathsf{rval}(e)$, $\mathsf{fences}(e)$ to select event components.

We use the following kinds of relations. A relation is a *strict partial order* if it is transitive and irreflexive. It is a *total order* if it additionally relates every two distinct elements one way or another. A relation is *prefix-finite* if each element is reachable along directed paths from at most finitely many others. A strict partial order R is an *interval order* if

$$\forall e_1, e_2, f_1, f_2.\, (e_1 \xrightarrow{R} e_2 \wedge f_1 \xrightarrow{R} f_2) \implies (e_1 \xrightarrow{R} f_2 \vee f_1 \xrightarrow{R} e_2).$$

Intuitively, an interval order R is consistent with an interpretation of events as segments of time during which the corresponding operations executed, with R ordering e before f if e finishes before f starts [13]. For example, the real-time order considered in linearizability [16] is an interval order.

A *history* is a triple $\mathcal{H} = (E, \mathsf{so}, \mathsf{rt})$, where: $E \subseteq \mathsf{Event}$; *session order* $\mathsf{so} \subseteq E \times E$ is a union of prefix-finite total orders over a finite number of disjoint subsets of E (each corresponding to operations by the same client); and *real-time order* $\mathsf{rt} \subseteq E \times E$ is a prefix-finite interval order such that $\mathsf{so} \subseteq \mathsf{rt}$ and $\forall e \in E.\, |\{f \in E \mid \neg(e \xrightarrow{\mathsf{rt}} f)\}| < \infty$.

The set E defines all operations invoked by clients in a single computation and can be infinite. The session order arranges operations by the same client in the order in which they were executed. The real-time order $e \xrightarrow{\mathsf{rt}} f$ tells us that the operation of e finished before the one of f started (the last restriction on rt ensures that every operation finishes). Tracking this relationship is important because it allows the client who executed the operation of e to communicate its return value to the client executing f out-of-band, without using the service; the return value of e can then influence the operation executed by f [12, 3]. We denote components of histories and similar structures as in $E_{\mathcal{H}}$ and $\mathsf{so}_{\mathcal{H}}$. A consistency model is defined by a set of histories.

3 Operational Specification

We define Global Sequence Consistency using the idealised protocol in Figure 1, which is a generalisation of the Global Sequence Protocol (GSP) [10]. It assumes a single *server* and a finite number of *clients*. The server state is represented by a log *server_log* of operations received from clients, tagged with unique identifiers from Id. The state of each client c includes three logs: $known_c$ is the prefix of *server_log* that c knows about; $pending_c$ is the log of operations by c that have not yet been pushed to the server; and $unacked_c$ is the log of operations by c that have been pushed to the server, but $known_c$ has not yet advanced enough to incorporate them.

The communication between the server and each client c is modeled by transitions $\texttt{push}(c)$ and $\texttt{pull}(c)$ that can fire nondeterministically at any time when the client is not executing an operation and atomically modify the client and the server state (implementations may refine this using asynchronous communication channels as in [10]). The $\texttt{push}(c)$ function models how the server processes the next operation by client c: it appends the oldest record in $pending_c$ to *server_log* and moves it to the end of $unacked_c$. The $\texttt{pull}(c)$ function models

State for each client c:
  $known_c \in (\mathsf{Id} \times \mathsf{Op})^*$
  $unacked_c \in (\mathsf{Id} \times \mathsf{Op})^*$
  $pending_c \in (\mathsf{Id} \times \mathsf{Op})^*$

`exec`(c, op, fen):
  if $(\mathsf{pull} \in fen)$
    while $(known_c \neq server\_log)$ `pull`(c)
  $result :=$
    $\mathsf{eval}(\mathsf{stripIds}(known_c \cdot unacked_c \cdot pending_c), op)$
  $pending_c := pending_c \cdot (\mathsf{uniqueId}(), op)$
  if $(\mathsf{push} \in fen)$
    while $(pending_c \neq [])$ `push`(c)
  return $result$

Server state:
  $server\_log \in (\mathsf{Id} \times \mathsf{Op})^*$

`push`(c):
  if $(pending_c = (id, op) \cdot remaining_c)$
    $server\_log := server\_log \cdot (id, op)$
    $unacked_c := unacked_c \cdot (id, op)$
    $pending_c := remaining_c$

`pull`(c):
  if $(server\_log = known_c \cdot (id, op) \cdot \_)$
    $known_c := known_c \cdot (id, op)$
    if $(unacked_c = (id, op) \cdot remaining_c)$
      $unacked_c := remaining_c$

Figure 1 The pseudocode of the protocol defining the GSC model. We denote sequence concatenation by $\cdot$, an empty sequence by $[]$ and an irrelevant expression by $\_$.

how the client c learns about the next entry in the server log: it appends to $known_c$ the next operation in $server\_log$ that is not yet part of $known_c$. If this operation is an echo of an operation previously executed by the same client c, we remove it from the $unacked_c$ log; the protocol ensures that in this case the operation is the first (oldest) one in $unacked_c$.

We model a client c executing an operation op with fences $fen \subseteq \{\mathsf{push}, \mathsf{pull}\}$ by `exec`(c, op, fen). The body of `exec`() is executed atomically, and only a single invocation of it can be in progress per client. At the beginning of `exec`(), we handle pull fences by repeatedly calling `pull`(c) until the local $known_c$ matches $server\_log$. At the end of `exec`(), we handle push fences by repeatedly calling `push`(c) until all $pending_c$ operations have been processed by the server. At the core of `exec`(), we first compute the result of the operation by conjoining the logs $known_c$, $unacked_c$ and $pending_c$, stripping identifiers using stripIds and applying the sequential semantics of operations defined by eval (§2). We then append the operation to the $pending_c$ with a unique identifier generated by uniqueId. Since op is evaluated on a log that includes $unacked_c$ and $pending_c$, the client is always guaranteed to observe its own operations, even before they are acknowledged by the server (the "read-your-writes" property [29]). Note that when fen is empty, `exec`(c, op, fen) returns immediately without communicating, so that in this case the protocol is partition-tolerant [14].

We only consider computations of the protocol that adhere to certain *fairness constraints*: every operation by a client eventually gets pushed to the server, every operation received by the server eventually gets pulled by any client and every invocation of `exec`() terminates.

The set of histories $(E, \mathsf{so}, \mathsf{rt})$ allowed by GSC is defined by considering all possible computations of the above protocol. The invocations of `exec`() define the set of events E, the order in which they are invoked on clients defines so, and two events are related by rt if the `exec`() function of the former finishes before the `exec`() function the latter starts. We denote the set of histories defined in this way HistGSC.

By systematically associating fences with operations in GSC we get various existing models as its special cases (Table 1). If operations are executed without any fences, the GSC protocol exactly matches the one used to define GSP [10]. If every operation includes a pull fence, then the GSC protocol is isomorphic to one defining the *Total Store Order (TSO)* consistency model [24, 23]. In this case, operations are always evaluated based on an up-to-date state on the server, but are propagated to the server asynchronously. If every operation includes a push fence, then the GSC protocol is isomorphic to one defining a recently proposed *dual TSO* model [2]. In this case, all operations are pushed to the server

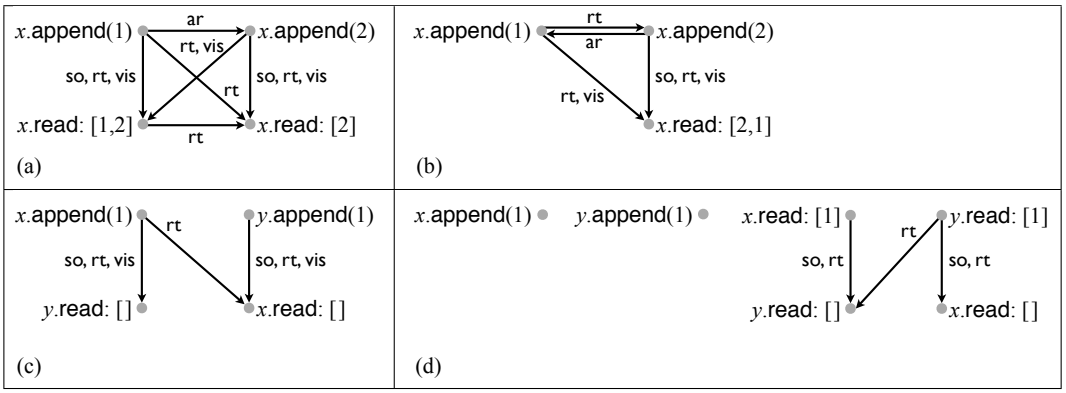


Figure 2 Examples of histories and abstract executions. Events do not include fences unless explicitly noted. Events by the same client are related by the session order so and laid out vertically. Thus, there are two clients in (a-c) and four in (d).

immediately, but are evaluated on a client-local possibly stale state. If every operation includes both a pull and a push fence, then the GSC protocol produces exactly those histories that are linearizable [16] (we prove this in [15, §C]). Informally, in this case the total order in which the operations go into *server_log* defines a linearization of the execution, which preserves the real-time order between the operations.

As a subcase of dual TSO, we also obtain a recently proposed *Ordered Sequential Consistency (OSC)* [22], which captures the semantics of coordination services such as ZooKeeper [18]. OSC assumes a partitioning of all operations into read-only and update operations: $\mathsf{Op} = \mathsf{OpReadOnly} \uplus \mathsf{OpUpdate}$. Read-only operations do not change the state of an object: for any operation op and a sequence of operations ξ, we have $\mathsf{eval}(\xi, op) = \mathsf{eval}(\xi|_{\mathsf{OpUpdate}}, op)$, where $\xi|_{\mathsf{OpUpdate}}$ is the projection of ξ onto OpUpdate. In our setting, OSC is defined by requiring that every operation include a push fence (like in dual TSO) and all updates additionally include a pull fence. Thus, update operations are evaluated on an up-to-date state, whereas read-only operations can be evaluated on a stale state. We prove the correspondence to the original OSC definition in [15, §C].

With unrestricted fence placements, GSC is weaker than linearizability, as we illustrate by the example histories in Figures 2(a-c) (for now ignore the extra relations vis and ar). They use sequence objects x and y for which $\mathsf{eval}(\xi, \mathsf{read})$ returns the sequence of values in the append operations in ξ. The histories in Figures 2(a-c) can be produced by the GSC protocol, but are not linearizable: there does not exist a linearization of the events consistent with the real-time order and the sequential semantics of objects. In the following, we briefly describe how the GSC protocol produces these histories; the reader may wish to consult [15, §A], where we describe the corresponding protocol computations in detail.

In history (a) the read by the second client does not see 1, even though it happens after the read by the first client that does see 1. In the GSC protocol this can happen if the second client does not pull append(1) from the server before executing the read. This history is disallowed if the read by the second client is executed with a pull fence: since the read by the first client returns $[1, 2]$, at the time the read is executed, 1 must be in *known* and, hence, on the server; then the pull fence ensures that the later read by the second client sees 1.

In history (b) the return value of the read is $[2, 1]$ even though append(1) finishes before append(2) starts. This can happen if the latter operation is pushed to the server before the

former. This outcome is disallowed if $\mathsf{append}(1)$ is executed with a push fence, so that it is pushed to the server before the operation finishes.

In history (c) each read does not see the append by the other client; this is a variant of the *store buffering* anomaly, characteristic of TSO [24]. It can be produced by the GSC protocol if the appends are pushed to the server only after the reads execute. The history is disallowed if the appends include push fences and the reads pull fences.

Finally, history (d) is a variant of the *independent reads of independent writes* anomaly [7] and cannot be produced by the GSC protocol. There two clients concurrently append 1 to different sequence objects x and y. A third client sees the append to x, but not to y, and a fourth client sees the append to y, but not to x. Thus, from the perspective the latter two clients the updates to x and y happen in different orders. This outcome cannot happen in a GSC protocol computation, because there is a single order in which the append operations will be incorporated into the server log. If $x.\mathsf{append}(1)$ precedes $y.\mathsf{append}(1)$ in the log, then the read from x in the fourth client cannot return $[\,]$; otherwise, the read from y in the third client cannot return $[\,]$.

4 Equivalence between GSP, TSO and Dual TSO

We now establish a certain relationship between special cases of the GSC model: TSO [24] (all operations pull), dual TSO [2] (all operations push) and GSP [10] (operations neither pull nor push). We prove that the sets of histories allowed by these three models are the same modulo the real-time order, which means that the models are observationally equivalent to clients that cannot communicate out-of-band [12, 3].

Formally, for an event $e = (\iota, x, op, a, fen)$ let $\mathsf{mkPull}(e) = (\iota, x, op, a, \{\mathsf{pull}\})$ and $\mathsf{mkPush}(e) = (\iota, x, op, a, \{\mathsf{push}\})$. We lift mkPull and mkPush to sets of events and relations in the expected way. Let $\mathsf{EPush} = \{e \mid \mathsf{push} \in \mathsf{fences}(e)\}$ and $\mathsf{EPull} = \{e \mid \mathsf{pull} \in \mathsf{fences}(e)\}$.

▶ **Theorem 1.**

$$\begin{aligned}\forall E.\, \forall \mathsf{so}.\, E \cap (\mathsf{EPush} \cup \mathsf{EPull}) = \emptyset \implies ((\exists \mathsf{rt}.\, (E, \mathsf{so}, \mathsf{rt}) \in \mathsf{HistGSC}) \iff \\ (\exists \mathsf{rt}'.\, (\mathsf{mkPush}(E), \mathsf{mkPush}(\mathsf{so}), \mathsf{rt}') \in \mathsf{HistGSC}) \iff \\ (\exists \mathsf{rt}''.\, (\mathsf{mkPull}(E), \mathsf{mkPull}(\mathsf{so}), \mathsf{rt}'') \in \mathsf{HistGSC})).\end{aligned}$$

We prove Theorem 1 in §7 and [15, §C]. According to it, any GSP computation of the protocol, where operations are propagated asynchronously both from clients to the server and from the server to clients, can be transformed into an equivalent-modulo-rt computation where operations can be propagated asynchronously in only one direction. While the equivalence between TSO and dual TSO has been established before [2], the result about GSP was only conjectured [10], and its proof is a contribution of the present paper. Like proofs of other results of ours, this one exploits the axiomatic specification of GSC that we present in §6.

If we take the real-time order into account and, hence, allow clients to communicate out-of-band, then GSP is strictly weaker than TSO and dual TSO, and the latter two are incomparable. In particular, the above theorem does not hold if we additionally require $\mathsf{rt}' = \mathsf{rt}$ or $\mathsf{rt}'' = \mathsf{rt}$. Indeed, as we noted in §3, the history in Figure 2(a) is allowed by GSP, but is disallowed if the operations pull; hence, it is disallowed by TSO. However, the history is allowed if all operations push and, hence, is allowed by dual TSO. The history in Figure 2(b) is similarly allowed by GSP, but is disallowed if all operations push; hence, it is disallowed by dual TSO. On the other hand, it is allowed if all operations pull and, hence, is allowed by TSO. Finally, even modulo real-time order, GSP, TSO and dual TSO are strictly weaker

than linearizability [16]: the history in Figure 2(c) is allowed by these models, but is not linearizable no matter how we change the real-time order.

5 Composing GSC Objects

GSC is not a *composable* (aka *local*) property [16]: objects satisfying GSC may fail to provide this consistency guarantee when combined. Indeed, consider the history in Figure 2(d). It is easy to see that the projections of the history to events on objects x or y yield GSC histories: e.g., the projection to x can be produced by the GSC protocol if the rightmost client is slow to pull updates from the server. However, as we explained in §3, the overall history is not GSC. We now give a condition under which multiple objects each satisfying GSC behave such that the whole set of objects satisfies GSC. The condition requires using fences according to a certain discipline, formalised as follows. A history $\mathcal{H} = (E, \mathsf{so}, \mathsf{rt})$ is *well-fenced* if

$$\forall e, f \in E.\, e \xrightarrow{\mathsf{so}} f \wedge \mathsf{obj}(e) \neq \mathsf{obj}(f) \implies \exists e' \in \mathsf{EPush}.\, \exists f' \in \mathsf{EPull}.\\ \mathsf{obj}(e') = \mathsf{obj}(e) \wedge \mathsf{obj}(f') = \mathsf{obj}(f) \wedge e \xrightarrow{\mathsf{so}?} e' \xrightarrow{\mathsf{so}} f' \xrightarrow{\mathsf{so}?} f,$$

where $R?$ is the reflexive closure of R. The above condition requires that, when switching between different objects, a client pushes to the server the operations done on the old object and pulls from the server operations on the new object. Let us denote by $\mathcal{H}|_x$ the projection of $\mathcal{H}$ to events on an object x. The following theorem is our main result (proved in §8).

▶ **Theorem 2.** *For a well-fenced history $\mathcal{H}$, we have* $(\forall x.\, \mathcal{H}|_x \in \mathsf{HistGSC}) \implies \mathcal{H} \in \mathsf{HistGSC}$.

The theorem ensures that well-fenced clients interacting with multiple GSC services, implementing different objects, behave as though they are interacting with a single GSC service. Since our histories track the real-time order between events, this result holds even when clients can communicate out-of-band, without using GSC services. Programmers can thus ensure consistency when accessing multiple GSC services by placing fences according to the proposed discipline. Even though fences are expensive (in particular, not partition-tolerant), clients only incur this overhead when switching between different services. A client accessing the same service incurs no overhead.

For example, assume we make the upper reads in Figure 2(d) push and the lower reads pull. Then the projection of the history to y is no longer GSC: since the lower read from y happens after the upper read from y and pulls operations from the server, it has to also observe 1. Hence, in this case the outcome shown in Figure 2(d) cannot happen when clients interact with multiple GSC services. (Actually, making the upper reads push is not required to ensure this, since they are read-only operations. Our results could be strengthened to incorporate such optimisations, but for simplicity we decided to treat all operations uniformly.)

As special cases of Theorem 2, we obtain novel criteria for composing TSO and dual TSO objects. Since in TSO all operations pull, we only need to require that a client pushes operations on an object before accessing a new one. Since in dual TSO all operations push, a client need only pull operations on the new object. As a subcase of dual TSO, we obtain the recently proposed criterion for composing OSC objects [22]. Recall that in OSC all operations push and update operations pull. Hence, in this case we require that a client start accessing a new object with an update operation. This can be ensured by adding dummy updates – a policy implemented by the ZooNet system [21] for composing ZooKeeper services [18]. Thus, our results generalise the compositionality criterion for OSC.

RetVal. $\forall e \in E.\, \mathsf{rval}(e) = \mathsf{eval}(\mathsf{ctxt}_{\mathcal{A}}(e), \mathsf{oper}(e))$.
RYW. $\mathsf{so} \subseteq \mathsf{vis}$.
MonotonicView. $\mathsf{vis}\,;\,\mathsf{so} \subseteq \mathsf{vis}$.
ObservedVis. $\mathsf{ar}?\,;\,(\mathsf{vis} \setminus \mathsf{so})\,;\,(\mathsf{rt} \cap (\mathsf{Event} \times \mathsf{EPull}))? \subseteq \mathsf{vis}$.
PushedVis. $\mathsf{ar}?\,;\,(\mathsf{rt}? \cap (\mathsf{EPush} \times \mathsf{EPull})) \subseteq \mathsf{vis}?$.
ObservedAr. $(\mathsf{vis} \setminus \mathsf{so})\,;\,\mathsf{rt} \subseteq \mathsf{ar}$.
PushedAr. $\mathsf{rt} \cap (\mathsf{EPush} \times \mathsf{Event}) \subseteq \mathsf{ar}$.
Eventual. $\forall e \in E.\, |\{f \in E \mid \neg(e \xrightarrow{\mathsf{vis}} f)\}| < \infty$.

Figure 3 Axioms of the GSC model, constraining an execution $\mathcal{A} = ((E, \mathsf{so}, \mathsf{rt}), \mathsf{vis}, \mathsf{ar})$.

6 Axiomatic Specification

We now present the main technical tool we use to prove Theorems 1 and 2 – an *axiomatic* specification of GSC, given in the style often used for consistency models in shared-memory [19] and distributed storage systems [9, 8]. It is based on the following notion. An *abstract execution* is a triple $\mathcal{A} = ((E, \mathsf{so}, \mathsf{rt}), \mathsf{vis}, \mathsf{ar})$, where: $(E, \mathsf{so}, \mathsf{rt})$ is a history; *visibility* $\mathsf{vis} \subseteq E \times E$ is a prefix-finite acyclic relation; and *arbitration* $\mathsf{ar} \subseteq E \times E$ is a prefix-finite total order such that $\mathsf{vis} \subseteq \mathsf{ar}$. Visibility and arbitration declaratively describe how the GSC protocol processes the operations in E. Given a computation of the protocol, we have $e \xrightarrow{\mathsf{vis}} f$ if, when a client executed the operation of f, the operation of e was in one of its three local logs. We have $e \xrightarrow{\mathsf{ar}} f$ if the operation of e preceded the one of f in the server log. Figures 2(a-c) give examples of abstract executions (we omit some edges irrelevant for the following explanations).

To define the set of histories allowed by GSC, our specification constrains abstract executions using the *consistency axioms* in Figure 3, which declaratively describe guarantees the GSC protocol provides about operation processing and are explained in the following. In the axioms $R_1; R_2$ denotes the sequential composition of relations R_1 and R_2; we define $\mathsf{ctxt}_{\mathcal{A}}$ below. The axiomatic specification admits those histories that can be extended to an abstract execution satisfying the axioms. Denoting the latter set of executions ExecGSC, the corresponding set of histories is

$$\mathsf{HistGSC}_{\mathrm{ax}} = \{\mathcal{H} \mid \exists \mathsf{vis}, \mathsf{ar}.\, (\mathcal{H}, \mathsf{vis}, \mathsf{ar}) \in \mathsf{ExecGSC}\}.$$

As the following shows, the axiomatic specification is equivalent to the operational one.

▶ **Theorem 3.** $\mathsf{HistGSC} = \mathsf{HistGSC}_{\mathrm{ax}}$.

We now explain the axioms in Figure 3 and, on the way, give the key ideas for the proof of the "$\subseteq$" direction of the theorem, showing the *soundness* of the axiomatic specification. Consider a computation of the GSC protocol producing a history $\mathcal{H} = (E, \mathsf{so}, \mathsf{rt})$. To prove the soundness result, we extract vis and ar from the computation as described above and show that the resulting abstract execution satisfies all the axioms in Figure 3. RetVal says that the result of an operation e is computed by applying its sequential semantics to the sequence of operations given by $\mathsf{ctxt}_{\mathcal{A}}(e)$, which is obtained by arranging the operations invoked by the events in the set $\{f \mid f \xrightarrow{\mathsf{vis}} e \wedge \mathsf{obj}(e) = \mathsf{obj}(f)\}$ according to ar. For example, the execution in Figure 2(b) satisfies RetVal: the read returns $[2, 1]$ because both appends are visible to it and $x.\mathsf{append}(2) \xrightarrow{\mathsf{ar}} x.\mathsf{append}(1)$. RYW formalises the "read-your-writes" guarantee from §3: a client observes all operations it has executed before. MonotonicView similarly ensures that a client observes all operations it has observed before.

The axioms ObservedVis to PushedAr are more subtle, and we thus give detailed justifications for their soundness. They constrain vis or ar based on the fact that, by a certain moment, a particular operation was guaranteed to have been pushed to the server. In ObservedVis and ObservedAr this is the case because the operation was observed by a client other the one that that executed it (expressed in the axioms using $\mathsf{vis} \setminus \mathsf{so}$); in PushedVis and PushedAr this is the case because the operation included a push fence (expressed using EPush). In more detail, these axioms are justified as follows:

- ObservedVis. Assume $e_1 \xrightarrow{\mathsf{ar}^?} e_2 \xrightarrow{\mathsf{vis}\setminus\mathsf{so}} e_3 \xrightarrow{\mathsf{rt}\cap(\mathsf{Event}\times\mathsf{EPull})^?} e_4$. Since $e_2 \xrightarrow{\mathsf{vis}\setminus\mathsf{so}} e_3$, when a client executed e_3, it was aware of the event e_2 by a different client. The client could only find out about e_2 from the server, so by the time e_3 finished, e_2 was on the server. Since $e_1 \xrightarrow{\mathsf{ar}^?} e_2$, so was e_1. If $e_3 = e_4$, then the client executing this event was also aware of e_1, since clients pull operations in the order of the server log. Hence, $e_1 \xrightarrow{\mathsf{vis}} e_4$. If $e_3 \xrightarrow{\mathsf{rt}\cap(\mathsf{Event}\times\mathsf{EPull})} e_4$, then after e_3 finished, the client executing e_4 pulled all updates from the server, which must have included e_1. Hence, $e_1 \xrightarrow{\mathsf{vis}} e_4$ again.
- PushedVis. Assume $e_1 \xrightarrow{\mathsf{ar}^?} e_2 \xrightarrow{\mathsf{rt}^?} e_3$, $e_2 \in \mathsf{EPush}$ and $e_3 \in \mathsf{EPull}$. Since $e_2 \in \mathsf{EPush}$, e_2 was on the server after its operation finished. Since $e_1 \xrightarrow{\mathsf{ar}^?} e_2$, so was e_1. If $e_1 = e_3$, we trivially have $e_1 \xrightarrow{\mathsf{vis}^?} e_3$. Otherwise, since $e_2 \xrightarrow{\mathsf{rt}^?} e_3$, e_1 was also on the server before e_3 started. Since $e_3 \in \mathsf{EPull}$, e_3 pulled all operations from the server, including e_1. Hence, $e_1 \xrightarrow{\mathsf{vis}} e_3$.
- ObservedAr. Assume $e_1 \xrightarrow{\mathsf{vis}\setminus\mathsf{so}} e_2 \xrightarrow{\mathsf{rt}} e_3$. Since $e_1 \xrightarrow{\mathsf{vis}\setminus\mathsf{so}} e_2$, e_1 must have been on the server by the time e_2 finished. Since $e_2 \xrightarrow{\mathsf{rt}} e_3$, e_3 started after e_2 finished and thus must follow e_1 in the server log. Hence, $e_1 \xrightarrow{\mathsf{ar}} e_3$.
- PushedAr. Assume $e_1 \xrightarrow{\mathsf{rt}} e_2$ and $e_1 \in \mathsf{EPush}$. Then e_1 was pushed to the server before e_2 started. Hence, e_2 was pushed onto the server after e_1, so that $e_1 \xrightarrow{\mathsf{ar}} e_2$.

Finally, the Eventual axiom guarantees that an event e can be invisible to at most finitely many other events f. Its soundness is ensured by the fairness constraints in the GSC protocol (§3). The axioms imply more properties of the relations in an execution.

▶ **Proposition 4.** *If $\mathcal{A}$ satisfies* MonotonicView *and* ObservedVis*, then $\mathsf{vis}_{\mathcal{A}}$ is transitive. If $\mathcal{A}$ satisfies* ObservedAr*, then $\mathsf{vis}_{\mathcal{A}} \cup \mathsf{rt}_{\mathcal{A}}$ is acyclic.*

The executions in Figures 2(a-c) satisfy all the axioms. On the other hand, the history in Figure 2(d) cannot be extended to an execution satisfying the axioms. Indeed, for the return values of the upper reads to be consistent with RetVal, we must have $x.\mathsf{append}(1) \xrightarrow{\mathsf{vis}} x.\mathsf{read} : [1]$ and $y.\mathsf{append}(1) \xrightarrow{\mathsf{vis}} y.\mathsf{read} : [1]$. Arbitration has to order the two appends one way or another. If, for example, we have $x.\mathsf{append}(1) \xrightarrow{\mathsf{ar}} y.\mathsf{append}(2)$, then by ObservedVis we must also have $x.\mathsf{append}(1) \xrightarrow{\mathsf{vis}} x.\mathsf{read} : [\,]$, contradicting RetVal.

Recall from §3 that GSC disallows the history in Figure 2(a) if the read in the second client is a pull. Accordingly, there is no abstract execution that extends the resulting history and satisfies the axioms: by ObservedVis, in such an execution we would have $x.\mathsf{append}(1) \xrightarrow{\mathsf{vis}} x.\mathsf{read} : [2]$, contradicting RetVal. Similarly, there is no execution that extends the history in Figure 2(b) assuming $x.\mathsf{append}(1)$ is a push. This is because by PushedAr in such an execution we must have $x.\mathsf{append}(1) \xrightarrow{\mathsf{ar}} x.\mathsf{append}(2)$, so that by RetVal the read must return $[1, 2]$. Finally, there is no execution for the history in Figure 2(c) assuming the appends push and the reads pull: by PushedVis we must have $x.\mathsf{append}(1) \xrightarrow{\mathsf{vis}} x.\mathsf{read} : [\,]$, contradicting RetVal.

As follows from the "$\supseteq$" direction of Theorem 3, the axioms in Figure 3 are also *complete*: given an abstract execution $(\mathcal{H}, \mathsf{vis}, \mathsf{ar})$, we can construct a computation of the GSC protocol producing the history $\mathcal{H}$. Due to space constraints, we defer the detailed proof of Theorem 3 to [15, §B]. The completeness part of the proof is nontrivial, but uses similar techniques to the proof of the compositionality criterion that we present in §8.

7 Proof of Model Equivalence

As a simple illustration of the use of the axiomatic specification of GSC, we prove the first "$\Longleftrightarrow$" in Theorem 1, showing that GSP and dual TSO are equivalent modulo real-time order (the rest of the proof is given in [15, §C]). Consider E and so such that $E \cap (\mathsf{EPush} \cup \mathsf{EPull}) = \emptyset$.

The "$\Longleftarrow$" direction. It is easy to see that

$$\forall \mathsf{rt}.\, (\mathsf{mkPush}(E), \mathsf{mkPush}(\mathsf{so}), \mathsf{mkPush}(\mathsf{rt})) \in \mathsf{HistGSC} \implies (E, \mathsf{so}, \mathsf{rt}) \in \mathsf{HistGSC},$$

since erasing fences from events does not invalidate any axioms.

The "$\Longrightarrow$" direction. Assume rt such that $(E, \mathsf{so}, \mathsf{rt}) \in \mathsf{HistGSC}$. Then for some vis and ar we have $\mathcal{A} \triangleq ((E, \mathsf{so}, \mathsf{rt}), \mathsf{vis}, \mathsf{ar}) \in \mathsf{ExecGSC}$. Let $\mathsf{rt}' = \mathsf{mkPush}(\mathsf{ar})$. Then

$$\mathcal{A}' \triangleq ((\mathsf{mkPush}(E), \mathsf{mkPush}(\mathsf{so}), \mathsf{rt}'), \mathsf{mkPush}(\mathsf{vis}), \mathsf{mkPush}(\mathsf{ar}))$$

is an abstract execution. Further, since $\mathcal{A}$ satisfies all GSC axioms, so does $\mathcal{A}'$. In particular, $\mathcal{A}'$ satisfies ObservedVis and PushedVis because $\mathsf{mkPush}(E) \cap \mathsf{EPull} = \emptyset$, and ObservedAr and PushedAr by the choice of rt'. This completes the proof.

Thus, our axiomatic specification allows easily proving the above model equivalence by picking a witness for the real-time order and checking axiom validity. Such a proof would be much more challenging with the operational specification, as it would require devising a nontrivial transformation of one execution of the GSC protocol into another.

8 Proof of the Compositionality Criterion

We next show how to use our axiomatic specification of the GSC model to prove Theorem 2. Here we give only the key ideas and defer the complete proof to [15, §D]. Consider a well-fenced history $\mathcal{H} = (E, \mathsf{so}, \mathsf{rt})$ such that $\forall x.\, \mathcal{H}|_x \in \mathsf{HistGSC}$. Then for any x there is an execution $\mathcal{A}_x = (\mathcal{H}|_x, \mathsf{vis}_x, \mathsf{ar}_x) \in \mathsf{ExecGSC}$. We need to show $\mathcal{H} \in \mathsf{HistGSC}$, to which end we construct an execution $\mathcal{A} = (\mathcal{H}, \mathsf{vis}, \mathsf{ar}) \in \mathsf{ExecGSC}$.

Let $\mathsf{so}_0 = \bigcup_{x \in \mathsf{Obj}} \mathsf{so}_{\mathcal{H}|_x}$, $\mathsf{vis}_0 = \bigcup_{x \in \mathsf{Obj}} \mathsf{vis}_x$ and $\mathsf{ar}_0 = \bigcup_{x \in \mathsf{Obj}} \mathsf{ar}_x$. It is reasonable to expect vis and ar to extend the corresponding per-object orders in $\mathcal{A}_x$, so we should have $\mathsf{vis}_0 \subseteq \mathsf{vis}$ and $\mathsf{ar}_0 \subseteq \mathsf{ar}$. The most difficult part is to construct ar; once this is done, we construct vis as the smallest relation containing vis_0 that is a solution to the system of inequalities given by the axioms RYW-PushedVis in Figure 3. The following lemma gives a closed form for this solution. Let $\mathsf{Id} = \{(e, e) \mid e \in E\}$.

▶ **Lemma 5.** *Given any arbitration order* $\mathsf{ar} \supseteq \mathsf{ar}_0$*, the relation*

$$\mathsf{vis} = \mathsf{so} \cup (\mathsf{ar}?\,;(\mathsf{vis}_0 \setminus \mathsf{so})\,;(\mathsf{rt} \cap (\mathsf{Event} \times \mathsf{EPull}))?\,;\mathsf{so}?) \cup ((\mathsf{ar}?\,;(\mathsf{rt}? \cap (\mathsf{EPush} \times \mathsf{EPull}))\,;\mathsf{so}?) \setminus \mathsf{Id})$$

is the smallest one such that $\mathsf{vis}_0 \subseteq \mathsf{vis}$ *and* $(\mathcal{H}, \mathsf{vis}, \mathsf{ar})$ *satisfies* RYW-PushedVis.

The first component of vis is meant to validate RYW, the second ObservedVis and the third PushedVis. Appending $\mathsf{so}?$ at the end of the last two components validates MonotonicView.

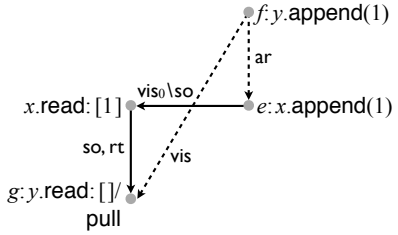


Figure 4 Motivation for $\prec$.

We now describe the construction of ar. This order needs to include several relations. Since $\mathsf{vis}_0 \subseteq \mathsf{vis}$ and $\mathcal{A}$ should satisfy ObservedAr, we must have $(\mathsf{vis}_0 \setminus \mathsf{so})\ ;\ \mathsf{rt} \subseteq \mathsf{ar}$. Since $\mathcal{A}$ should satisfy PushedAr we must have $\overline{\mathsf{rt}} \triangleq \mathsf{rt} \cap (\mathsf{EPush} \times \mathsf{Event}) \subseteq \mathsf{ar}$. Since $\mathcal{A}$ should satisfy RYW and $\mathsf{vis} \subseteq \mathsf{ar}$, we must have $\mathsf{so} \subseteq \mathsf{ar}_0$. Finally, for $\mathcal{A}$ to satisfy RetVal, ar should include one more relation that is more subtle. We illustrate the need for it using the example in Figure 4. Assume that we have the solid edges in the figure. If we arbitrate between the two appends as shown by the dashed edge $f \xrightarrow{\mathsf{ar}} e$, then according to the construction in Lemma 5 we will also have the dashed edge $f \xrightarrow{\mathsf{vis}} g$ (needed for $\mathcal{A}$ to satisfy ObservedVis). But then the resulting $\mathcal{A}$ will violate RetVal. We therefore include the following relation into ar, which ensures that such situations do not happen:

$$\begin{aligned} e \prec f \iff\ & \exists g.\, \mathsf{obj}(f) = \mathsf{obj}(g) \wedge (f,g) \notin \mathsf{vis}_0 \wedge \\ & (e,g) \in (\mathsf{vis}_0 \setminus \mathsf{so})\ ;\ (\mathsf{rt} \cap (\mathsf{Event} \times \mathsf{EPull}))\ ;\ \mathsf{so}_0? \cup (\mathsf{rt} \cap (\mathsf{EPush} \times \mathsf{EPull}))\ ;\ \mathsf{so}_0?. \end{aligned}$$

If $e \prec f$, then adding an edge $f \xrightarrow{\mathsf{ar}} e$ would create a visibility edge $f \xrightarrow{\mathsf{vis}} g$ between events on the same object that is not in vis_0. Note that the expression covering (e,g) above is more specific than the one in Lemma 5: we have so_0 instead of so, and rt must be used. This is crucial for the proof (specifically, Lemma 6 below) and, as we show, is still sufficient to validate RetVal because the history $\mathcal{H}$ is well-fenced.

Thus, we need to construct an ar that includes $R \triangleq \overline{\mathsf{rt}} \cup \mathsf{so} \cup \mathsf{ar}_0 \cup ((\mathsf{vis}_0 \setminus \mathsf{so})\ ;\ \mathsf{rt}) \cup \prec$. For this to be possible, R has to be acyclic.

▶ **Lemma 6.** $\overline{\mathsf{rt}} \cup \mathsf{so} \cup \mathsf{ar}_0 \cup ((\mathsf{vis}_0 \setminus \mathsf{so})\ ;\ \mathsf{rt}) \cup \prec$ *is acyclic.*

Establishing this lemma is the most subtle part of the proof. To do this, we construct a closed-form expression covering the transitive closure of R.

▶ **Lemma 7.**

$$\begin{aligned} & (\overline{\mathsf{rt}} \cup \mathsf{so} \cup \mathsf{ar}_0 \cup ((\mathsf{vis}_0 \setminus \mathsf{so})\ ;\ \mathsf{rt}) \cup \prec)^+ \\ = {} & (\overline{\mathsf{rt}} \cup \mathsf{so} \cup \mathsf{ar}_0 \cup ((\mathsf{vis}_0 \setminus \mathsf{so})\ ;\ \mathsf{rt}))^+ \cup (\prec \cup \mathsf{ar}_0\ ;\ \prec)\ ;\ (\overline{\mathsf{rt}} \cup \mathsf{so} \cup \mathsf{ar}_0 \cup ((\mathsf{vis}_0 \setminus \mathsf{so})\ ;\ \mathsf{rt}))^* \quad \textit{and} \\ & (\overline{\mathsf{rt}} \cup \mathsf{so} \cup \mathsf{ar}_0 \cup ((\mathsf{vis}_0 \setminus \mathsf{so})\ ;\ \mathsf{rt}))^+ \\ \subseteq {} & \overline{\mathsf{rt}} \cup \mathsf{ar}_0 \cup (\mathsf{ar}_0\ ;\ \overline{\mathsf{rt}}) \cup (\overline{\mathsf{rt}}\ ;\ \mathsf{ar}_0) \cup (\mathsf{ar}_0\ ;\ \overline{\mathsf{rt}}\ ;\ \mathsf{ar}_0) \cup ((\mathsf{vis}_0 \setminus \mathsf{so})\ ;\ \mathsf{rt}) \cup \\ & (\mathsf{ar}_0\ ;\ ((\mathsf{vis}_0 \setminus \mathsf{so})\ ;\ \mathsf{rt})) \cup (((\mathsf{vis}_0 \setminus \mathsf{so})\ ;\ \mathsf{rt})\ ;\ \mathsf{ar}_0) \cup (\mathsf{ar}_0\ ;\ ((\mathsf{vis}_0 \setminus \mathsf{so})\ ;\ \mathsf{rt})\ ;\ \mathsf{ar}_0). \end{aligned}$$

The proof Lemma 7 relies on establishing that components of R satisfy various algebraic properties, some of which exploit the fact that the history $\mathcal{H}$ is well-fenced. For example, we prove that $\prec$ is a strict partial order, i.e., transitive and irreflexive.

To prove Lemma 6, it is thus sufficient to prove that the relation covering R^+ in Lemma 7 is irreflexive. This relation describes only particular paths in R of length at most 5. Its irreflexivity is then established by a case analysis on these paths.

Using Lemma 6, we can extend R to a prefix-finite total order, which we take as ar; then vis is defined by Lemma 5. We can then show that vis defined in this way is prefix-finite, acyclic and $\mathsf{vis} \subseteq \mathsf{ar}$, so that $\mathcal{A} = (\mathcal{H}, \mathsf{vis}, \mathsf{ar})$ is an abstract execution. By Lemma 5, $\mathcal{A}$ satisfies RYW-PushedVis. It satisfies PushedAr because $\overline{\mathsf{rt}} \subseteq \mathsf{ar}$, and it is also easy to check that it satisfies ObservedAr.

We next argue that $\mathcal{A}$ satisfies RetVal, which exploits the particular way in which we constructed ar. To this end, we show that for any object x we have $\mathsf{vis}|_x = \mathsf{vis}_x$, where $\mathsf{vis}|_x$ is the projection of vis to events on x. Then since for any x we have $\mathsf{ar}_x \subseteq \mathsf{ar}$ and $\mathcal{A}_x$ satisfies RetVal, so does $\mathcal{A}$. Since $\mathsf{vis}_x \subseteq \mathsf{vis}$ by construction, we only need to show $\mathsf{vis}|_x \subseteq \mathsf{vis}_x$. Consider arbitrary $f, g \in E$ such that $\mathsf{obj}(f) = \mathsf{obj}(g) = x$ and $f \xrightarrow{\mathsf{vis}} g$. To show $f \xrightarrow{\mathsf{vis}_x} g$ our proof considers several cases corresponding to which of the components of the union defining vis in Lemma 5 the edge (f, g) belongs to. For illustration, here we only consider a single case when (f, g) comes from the following instance of the second component of the union, which uses an rt edge: $(f, g) \in \mathsf{ar}? \,;\, (\mathsf{vis}_0 \setminus \mathsf{so}) \,;\, (\mathsf{rt} \cap (\mathsf{Event} \times \mathsf{EPull})) \,;\, \mathsf{so}?$. Then for some g' we have

$$f \xrightarrow{\mathsf{ar}?;(\mathsf{vis}_0 \setminus \mathsf{so});(\mathsf{rt} \cap (\mathsf{Event} \times \mathsf{EPull}))} g' \xrightarrow{\mathsf{so}?} g.$$

Figure 4 illustrates the case when $g' = g$. If $\mathsf{obj}(g') \neq \mathsf{obj}(g)$, then since the history $\mathcal{H}$ is well-fenced, for some $g'' \in \mathsf{EPull}$ we have $g' \xrightarrow{\mathsf{so}} g'' \xrightarrow{\mathsf{so}_0?} g$. Since $\mathsf{so} \subseteq \mathsf{rt}$, this implies $g' \xrightarrow{\mathsf{rt} \cap (\mathsf{Event} \times \mathsf{EPull})} g'' \xrightarrow{\mathsf{so}_0?} g$. Hence,

$$f \xrightarrow{\mathsf{ar}?;(\mathsf{vis}_0 \setminus \mathsf{so});(\mathsf{rt} \cap (\mathsf{Event} \times \mathsf{EPull}))} g'' \xrightarrow{\mathsf{so}_0?} g. \tag{1}$$

If $\mathsf{obj}(g') = \mathsf{obj}(g)$, then $g' \xrightarrow{\mathsf{so}_0?} g$ and we again have (1) for $g'' = g'$. Thus, in all cases (1) holds for some g''. Then for some e we have

$$f \xrightarrow{\mathsf{ar}?} e \xrightarrow{(\mathsf{vis}_0 \setminus \mathsf{so});(\mathsf{rt} \cap (\mathsf{Event} \times \mathsf{EPull}))} g'' \xrightarrow{\mathsf{so}_0?} g.$$

Now if $\neg(f \xrightarrow{\mathsf{vis}_x} g)$, then $e \prec f$, contradicting $\prec \subseteq \mathsf{ar}$. Hence, $f \xrightarrow{\mathsf{vis}_x} g$, as required.

Thus, $\mathcal{A}$ satisfies all GSC axioms except for possibly Eventual. Since $\forall x.\, \mathsf{vis}|_x = \mathsf{vis}_x$ and $\mathcal{A}_x$ satisfies Eventual, we have

$$\forall e \in E.\, |\{f \in E \mid \mathsf{obj}(e) = \mathsf{obj}(f) \wedge \neg(e \xrightarrow{\mathsf{vis}} f)\}| < \infty, \tag{2}$$

i.e., an event e cannot be invisible to infinitely many events f on the same object. Then, as the following lemma shows, we can extend vis so as to validate Eventual without invalidating any of the other axioms.

▶ **Lemma 8.** *Let $\mathcal{H} = (E, \mathsf{so}, \mathsf{rt})$ and $\mathcal{A} = (\mathcal{H}, \mathsf{vis}, \mathsf{ar})$ be an execution that satisfies all GSC axioms except for possibly* Eventual. *Assume (2) holds. Then there exists $\mathsf{vis}' \supseteq \mathsf{vis}$ such that $(\mathcal{H}, \mathsf{vis}', \mathsf{ar}) \in \mathsf{ExecGSC}$.*

We thus construct an execution $(\mathcal{H}, \mathsf{vis}', \mathsf{ar}) \in \mathsf{ExecGSC}$, which shows that $\mathcal{H} \in \mathsf{HistGSC}$ and thereby establishes Theorem 2.

The axiomatic specification of GSC plays an important role in the above proof. It allows us to concisely state constraints that the global order on operations represented by ar needs to satisfy for the global execution to be GSC. We can then show that the desired global order exists by proving algebraic properties over relations, as exemplified by Lemma 7.

9 Related Work and Discussion

Lev-Ari et al. [22] have proposed a criterion for composing objects providing Ordered Sequential Consistency (OSC), which is a special case of our results (§5). In comparison to them, we handle a more complex consistency model, which requires a different proof approach: specifying the consistency model axiomatically and reasoning about it using algebraic techniques. Lev-Ari et al. have also implemented their criterion in a library for composing ZooKeeper instances and showed that it has a competitive performance [21]. We hope that our results will enable similar practical implementations for systems providing other consistency models from the family we considered. In particular, the implementation of GSP in Orleans [5] provides only per-object consistency guarantees, and our results should allow its clients to use multiple objects while preserving the consistency model.

There are other widely used consistency models that are in general non-composable, such as sequential consistency [20]. Perrin et al. [25] proposed conditions on the use of sequentially consistent concurrent objects under which a composition of multiple objects stays sequentially consistent. Our compositionality result is similar in spirit, but handles a family of more complex consistency models implemented in modern systems [10, 23, 18]. Vitenberg and Friedman [30] showed that combining sequential consistency with any composable property yields a non-composable property. Our compositionality criterion does not contradict this result, since well-fencedness of histories is not a composable property.

Our operational specification of the GSC model generalizes the GSP protocol [10], with significant differences. First, GSP allows only pure read and update operations, while GSC permits mixed operations that both modify the state and return a value to the caller. Second, GSP does not support push and pull fences that are attached to operations. Rather, its original proposal [10] investigated stronger synchronization primitives, such as standalone fences and transactions, which cannot be used to define TSO, dual TSO and OSC as special cases. Therefore, GSP is unsuitable to serve as a unifying model that clarifies the relationship between these instances.

Axiomatic specifications have been previously proposed for consistency models in shared-memory [23, 19] and distributed storage systems [9, 8]. Our GSC specification uses the same framework as for the latter. Researchers have proposed axiomatic specifications for TSO-like models and proved their equivalence to operational ones [23, 17]. However, our specifications are the first to formalise the role of the real-time order in distinguishing between these models. Including real-time order into axiomatic models [8] is important in a distributed setting because of the possibility of out-of-band communication between clients; without this one cannot safely substitute implementations for specifications [12, 3].

We have exploited the axiomatic specification of GSC to establish a compositionality criterion and an equivalence between GSP and TSO/dual TSO. However, axiomatic specifications of consistency models have been shown useful to obtain other kinds of results, such as criteria for *robustness* – checking when an application running on a weak consistency model behaves as if it runs on a strong one [27, 4]. We hence hope that our specifications will allow obtaining such results for consistency models with global operation sequencing.

Acknowledgements. We thank Idit Keidar, Kfir Lev-Ari and Matthieu Perrin for helpful comments.

References

1 Microsoft TouchDevelop. https://www.touchdevelop.com/.

2 Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. The benefits of duality in verifying concurrent programs under TSO. In *CONCUR: International Conference on Concurrency Theory*, 2016.

3 Hagit Attiya, Alexey Gotsman, Sandeep Hans, and Noam Rinetzky. A programming language perspective on transactional memory consistency. In *PODC: Symposium on Principles of Distributed Computing*, 2013.

4 Giovanni Bernardi and Alexey Gotsman. Robustness against consistency models with atomic visibility. In *CONCUR: International Conference on Concurrency Theory*, 2016.

5 Phil Bernstein, Sebastian Burckhardt, Sergey Bykov, Natacha Crooks, Jose Faleiro, Gabriel Kliot, Alok Kumbhare, Muntasir Raihan Rahman, Vivek Shah, Adriana Szekeres, and Jorgen Thelin. Geo-distribution of actor-based services. Technical Report MSR-TR-2017-3, Microsoft Research, 2017.

6 Philip A. Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. Orleans: Distributed virtual actors for programmability and scalability. Technical Report MSR-TR-2014-41, Microsoft Research, 2014.

7 Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In *PLDI: Conference on Programming Language Design and Implementation*, 2008.

8 Sebastian Burckhardt. *Principles of Eventual Consistency.* now publishers, 2014.

9 Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: specification, verification, optimality. In *POPL: Symposium on Principles of Programming Languages*, 2014.

10 Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. Global sequence protocol: A robust abstraction for replicated shared state. In *ECOOP: European Conference on Object-Oriented Programming*, 2015.

11 Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4), 2004.

12 Ivana Filipovic, Peter O'Hearn, Noam Rinetzky, and Hongseok Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 411(51-52), 2010.

13 Peter C. Fishburn. Intransitive indifference with unequal indifference intervals. *Journal of Mathematical Psychology*, 7, 1970.

14 Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2), 2002.

15 Alexey Gotsman and Sebastian Burckhardt. Consistency models with global operation sequencing and their composition (extended version). *arXiv CoRR*, 1707.09242, 2017. URL: `http://arxiv.org/abs/1701.05463`.

16 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3), 1990.

17 Lisa Higham and Jalal Kawash. Memory consistency and process coordination for sparc multiprocessors. In *HiPC: International Conference on High Performance Computing*, 2000.

18 Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC: USENIX Annual Technical Conference*, 2010.

19 ISO/IEC Standard. Programming languages – C++, 14882:2011, 2011.

20 L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9), 1979.

21 Kfir Lev-Ari, Edward Bortnikov, Idit Keidar, and Alexander Shraer. Modular composition of coordination services. In *USENIX ATC: USENIX Annual Technical Conference*, 2016.

22 Kfir Lev-Ari, Edward Bortnikov, Idit Keidar, and Alexander Shraer. Composing ordered sequential consistency. *Information Processing Letters*, 123, 2017.

23 Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In *TPHOLs: International Conference on Theorem Proving in Higher Order Logics*, 2009.

24 S. Park and D. L. Dill. An executable specification, analyzer and verifier for RMO. In *SPAA'95: Symposium on Parallel Algorithms and Architectures*, 1995.

25 Matthieu Perrin, Matoula Petrolia, Achour Mostéfaoui, and Claude Jard. On composition and implementation of sequential consistency. In *DISC: International Symposium on Distributed Computing*, 2016.

26 Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22, 1990.

27 Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2), 1988.

28 Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 1st edition, 2011.

29 Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent W. Welch. Session guarantees for weakly consistent replicated data. In *PDIS: Conference on Parallel and Distributed Information Systems*, 1994.

30 Roman Vitenberg and Roy Friedman. On the locality of consistency conditions. In *DISC: International Symposium on Distributed Computing*, 2003.

Improved Deterministic Distributed Construction of Spanners*

Ofer Grossman[1] and Merav Parter[2]

1 **MIT, CSAIL, Cambridge, USA**
ofer.grossman@gmail.com
2 **Weizmann Institute of Science, Rehovot, Israel**
merav.parter@weizmann.ac.il

Abstract

Graph spanners are fundamental graph structures with a wide range of applications in distributed networks. We consider a standard synchronous message passing model where in each round $O(\log n)$ bits can be transmitted over every edge (the CONGEST model).

The state of the art of deterministic distributed spanner constructions suffers from large messages. The only exception is the work of Derbel et al. [9], which computes an optimal-sized $(2k-1)$-spanner but uses $O(n^{1-1/k})$ rounds.

In this paper, we significantly improve this bound. We present a deterministic distributed algorithm that given an unweighted n-vertex graph $G = (V, E)$ and a parameter $k > 2$, constructs a $(2k-1)$-spanner with $O(k \cdot n^{1+1/k})$ edges within $O(2^k \cdot n^{1/2-1/k})$ rounds for every even k. For odd k, the number of rounds is $O(2^k \cdot n^{1/2-1/(2k)})$. For the weighted case, we provide the first deterministic construction of a 3-spanner with $O(n^{3/2})$ edges that uses $O(\log n)$-size messages and $\widetilde{O}(1)$ rounds. If the vertices have IDs in $[1, \Theta(n)]$, the spanner is computed in only 2 rounds!

1998 ACM Subject Classification G.2.2 Graph Algorithms

Keywords and phrases spanners, clustering, deterministic algorithms

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.24

1 Introduction & Related Work

Graph spanners are fundamental graph structures that are used as a key building block in various communication applications, e.g., routing, synchronizers, broadcasting, distance oracles, and shortest path computations. For this reason, the distributed construction of sparse spanners has been studied extensively [2, 5, 6, 7, 8, 9, 15]. The standard setting is a synchronous message passing model where per round each node can send one message to each of its neighbors. Of special interest is the case where the message size is limited to $O(\log n)$ bits, a.k.a. the CONGEST model.

The common objective in distributed computation of spanners is to achieve the best-known existential size-stretch trade-off as fast as possible: It is folklore that for every graph $G = (V, E)$, there exists a $(2k-1)$-spanner $H \subseteq G$ with $O(n^{1+1/k})$ edges. Moreover, this size-stretch tradeoff is believed to be optimal, following the girth conjecture of Erdős.

Designing deterministic algorithms for local problems has been receiving a lot of attention since the foundation of the area in 1980's. Towards the end of this section, we elaborate more on the motivation for studying deterministic algorithms in the distributed setting.

* A full version of the paper is available at https://arxiv.org/abs/1708.01011.

© Ofer Grossman and Merav Parter;
licensed under Creative Commons License CC-BY

State of the art for deterministic distributed constructions of spanners. Whereas there are efficient randomized constructions for spanners, as the reader will soon notice, the state of the art for distributed deterministic spanner constructions suffers from large message sizes: Derbel and Gavoille [5] construct constant stretch spanners with $o(n^2)$ edges and $O(n^\epsilon)$ rounds for any constant ϵ, using messages of size $O(n)$. Derbel, Gavoille and Peleg improved this result and presented in [6] a construction of an $O(k)$-spanner with $O(kn^{1+1/k})$ edges in $O(\log^{k-1} n)$ rounds. This was further improved in the seminal work of Derbel, Gavoille, Peleg, and Viennot [7], which provides a deterministic k-round algorithm for constructing $(2k-1)$-spanners with optimal size. However, again the algorithm uses messages of size $O(n)$. Using large messages is indeed inherent to all known efficient deterministic techniques, which are mostly based on network decomposition and graph partitioning. In the conventional approaches of network decomposition, the deterministic algorithms for spanners usually require a vertex to learn the graph topology induced by its $O(1)$-neighborhood. This cannot be done efficiently with small messages.

As Pettie [15] explicitly noted, *all* these constructions have the disadvantage of using large messages. Derbel et al. [8] also pointed out that constructing sparse spanners deterministically with small message sizes remains open.

The state of the art when using small messages. There are only two exceptions for this story. Barenboim et al. [1] showed a construction of $O(\log^{k-1} n)$ spanner with $O(n^{1+1/k})$ edges in $O(\log^{k-1} n)$ rounds. Hence, whereas the runtime is polylogarithmic, the stretch-size tradeoff of the output spanner is quite far from the optimal one.

We are then left with only *one* previous work that fits our setting, due to Derbel, Mosbah and Zemmari [9]. They provide a deterministic construction of an optimal-size $(2k-1)$-spanner but using $O(n^{1-1/k})$ rounds.

The state of the art in other distributed settings. Turning to *randomized* constructions, perhaps one of the most well known approaches to construct a spanner is given by Baswana and Sen [2], which we review soon. Recently, [3] showed that the Baswana-Sen algorithm can be derandomized in the congested clique model of communication in which every pair of nodes (even non-neighbors in the input graph) can exchange $O(\log n)$ bits per round. Note that this model is much stronger than the standard model in which only *neighboring* vertices can communicate. Indeed the algorithm of [3] requires a global evaluation of the random seed, thus implementing this algorithm in the standard CONGEST model requires $\Omega(\text{diam}(G) + n^{1-1/k})$ rounds where $\text{diam}(G)$ is the diameter of the graph. Hence, deterministic construction of spanners in the CONGEST model calls for new ideas!

Before we proceed with introducing our main contribution, we make a short pause to further motivate the study of deterministic distributed algorithms.

A note on deterministic distributed algorithms. Much effort has been invested in designing deterministic distributed algorithms for local problems. Examples include MIS (maximal independent set), vertex coloring, edge coloring and and matching. Until recently, a deterministic poly-log n round algorithm was known only for the maximal matching problem (see [13] and a recent improvement by [10]). In a recent breakthrough [11], a polylogarithmic solution was provided also for the $(2\Delta-1)$ edge coloring[1]. Aside from the general theoretical question,

[1] Where Δ is the maximum degree in the graph.

the distributed setting adds additional motivation for studying deterministic algorithms (as nicely noted in [10]). First, in the centralized setting, if the randomized algorithm ends with an error, we can just repeat. In the distributed setting, detecting a global failure requires communicating to a leader, which blows up the runtime by a factor of network diameter. Second, for problems as MIS, [4] showed that improving the randomized complexity requires an improvement in the deterministic complexity.

Wheres most results for deterministic local problems are for the LOCAL model, which allows unbounded messages; the size of messages that are sent throughout the computation is a second major attribute of distributed algorithms. It is therefore crucial to study the complexity of local problems under bandwidth restrictions. Surprisingly, most of the algorithms for local problems already use only small messages. The problem of spanners is distinguished from these problems, and in fact, spanners is the only setting we are aware of, in which all existing deterministic algorithms use *large* messages. Hence, the main challenge here is in the *combination* of **deterministic** algorithms with **congestion** constraints.

1.1 Our Contribution

Our main result is:

▶ **Theorem 1.** *For every n-vertex unweighted graph $G = (V, E)$ and even k, there exists a deterministic distributed algorithm that constructs a $(2k-1)$-spanner with $O(k \cdot n^{1+1/k})$ edges in $O(2^k \cdot n^{1/2-1/k})$ rounds using $O(\log n)$-size messages*[2]*.*

A key element in our algorithm is the construction of sparser spanners for unbalanced bipartite graph. This construction might become useful in other spanner constructions.

▶ **Lemma 2** (Bipartite Spanners)**.** *Let $G = (A \cup B, E)$ be an unweighted bipartite graph, with $|A| \leq |B|$. For even $k \geq 4$, one can construct (in the* CONGEST *model) a $(2k-1)$ spanner H with $|E(H)| = O(k|A|^{1+2/k} + |B|)$ edges within $O(|A|^{1-2/k})$ rounds*[3]*.*

Turning to weighted graphs, much less in known about the deterministic construction of spanners in the distributed setting. The existing deterministic constructions of optimal-sized $(2k-1)$-spanners (even in the LOCAL model) are restricted to *unweighted* graphs, already for $k = 2$. If the edge weights are bounded by some number W, there is a simple reduction[4] to the unweighted setting, at the cost of increasing the stretch by a factor of $(1+\epsilon)$ and the size of the spanner by a factor of $\log_{1+\epsilon} W$. Hence, already in the LOCAL model and $k = 2$, we only have a $(3+\epsilon)$ spanner with $\widetilde{O}(n^{3/2})$ edges. Whereas our general approach does not support the weighted case directly, our algorithm for 3-spanners does extend for weighted graphs. Hence, we give here the first deterministic construction with nearly tight tradeoff between the size, stretch and *runtime*.

▶ **Theorem 3** (3-Spanners for Weighted Graphs)**.** *For every n-vertex weighted graph $G = (V, E)$, there exists a deterministic distributed algorithm that constructs a 3-spanner with $O(n^{3/2})$ edges in $O(\log n)$ rounds using $O(\log n)$-size messages. If vertices have IDs in the range of $[1, O(n)]$, it can be done in two rounds.*

[2] For odd k, we obtain a similar theorem, but with $O(2^k \cdot n^{1/2-1/(2k)})$ rounds.
[3] Hence, yielding an improved edge bound, for $|A| \leq n^{(k+1)/(k+2)}$.
[4] Apply the algorithm for unweighted graphs separately for every weight scale $((1+\epsilon)^i, (1+\epsilon)^{i+1}]$.

1.2 Our Approach and Key Ideas in a Nutshell

For the sake of discussion, let $k = O(1)$ throughout this section.

A brief description of the randomized construction by Baswana-Sen. A clustering $\mathcal{C} = \{C_1, \ldots, C_\ell\}$ is a collection of vertex disjoint sets which we call clusters. Every cluster has some a special vertex which we call the *cluster center*. In the high level, the Baswana-Sen algorithm computes k levels of clustering $\mathcal{C}_0, \ldots, \mathcal{C}_{k-1}$ where each clustering $\mathcal{C}_i$ is obtained by sampling the cluster center of each cluster in $\mathcal{C}_{i-1}$ with probability $n^{-1/k}$. Each cluster $C \in \mathcal{C}_i$ has in the spanner H, a BFS tree of depth i rooted at the cluster center spanning[5] all the nodes of C. The vertices that are not incident to the sampled clusters become unclustered. For each unclustered vertex v, the algorithm adds one edge to each of the clusters incident to v in $\mathcal{C}_{i-1}$. This randomized construction is shown to yield a spanner with $O(kn^{1+1/k})$ edges in expectation and it can be implemented in $O(k^2)$ rounds[6]. Note that the only randomized step in Baswana-Sen is in picking the cluster centers of the i^{th} clustering. That is, given the $n^{1-(i-1)/k}$ cluster centers Z_{i-1} of the clusters in $\mathcal{C}_{i-1}$, it is required to pick $n^{-1/k}$ fraction of it, to be centers of the clusters in $\mathcal{C}_{i-1}$.

The brute-force deterministic solution in $O(n)$ rounds. A brute-force approach to pick the new cluster centers of $\mathcal{C}_i$ is to iterate over the clusters in $\mathcal{C}_{i-1}$ one by one, checking if they satisfy some expansion criteria. Informally, the expansion is measured by the number of vertices in the i^{th}-neighborhood of the cluster center (i.e., number of vertices that can be covered[7] by the cluster center in case it proceeds to the i^{th} level). If the expansion is large enough, the current cluster "expands" (i.e., covers vertices up to distance i), and joins the i^{th}-level of the clustering $\mathcal{C}_i$. Since in the first level there are $O(n)$ clusters (each vertex forms a singleton cluster), this approach gives an $O(n)$-round algorithm. With some adaptations, this approach can yield an improved $O(n^{1-1/k})$ round algorithm (as in [9]).

Our $O(n^{1/2-1/k})$-round deterministic solution. Inspired by the randomized construction of Baswana-Sen and the work of Derbel, Gavoille, and Peleg [6], we present a new approach for constructing spanners, based on two novel components which we discuss next.

1.2.1 Key Idea (I): Grouping Baswana-Sen Clusters into Superclusters

Our approach is based on adding an additional level of clustering on top of Baswana-Sen clustering. We introduce the novel notion of a supercluster – a subset of Baswana-Sen clusters that are *close* to each other in G. In every level $i \leq k/2$, we group the $O(n^{1-i/k})$ clusters of $\mathcal{C}_i$ into $O(\sqrt{n})$ *superclusters*, each containing $O(n^{1/2-i/k})$ clusters which are also close to each other in G. Specifically, the superclusters have the following useful structure: cluster-centers of the same supercluster are connected in G by a *constant depth tree* (i.e., the weak diameter of the superclusters is $O(1)$), and the trees of different superclusters are *edge-disjoint*.

Unlike the brute-force $O(n)$-round algorithm mentioned above, our algorithm iterates over *superclusters* rather than clusters. We define the neighborhood of the supercluster to be all vertices that belong to – or have a neighbor in– one of the clusters of that supercluster. The expansion of the supercluster is simply the size of this neighborhood. The importance

[5] The vertices of the tree are precisely the vertices of the cluster.
[6] With some care, we believe the algorithm can also be implemented in $O(k)$ rounds
[7] We say the a vertex is covered by a cluster-center if it gets into its cluster.

of having this specific structure in each supercluster is that it allows the superclusters to compute their expansion in $O(1)$ rounds[8]. The faith of the superclusters (i.e., whether they continue on to the next level of clustering), in our algorithm, is determined by their expansion. If the expansion of a supercluster is sufficiently high, *all* the cluster centers of that supercluster join the next level i of the clustering. Otherwise, all these clusters are discarded from the clustering. As we will show in depth, the algorithm makes sure that at most $O(n^{1/2-1/k})$ superclusters pass this "expansion test" and the remaining superclusters with low-expansion are handled using our second key tool as explained next.

1.2.2 Key Idea (II): Better Spanners for Unbalanced Bipartite Graphs

In our spanner construction, each supercluster with low-expansion has additional useful properties: it has $|A| = O(\sqrt{n})$ vertices and only $|B| = O(n^{1/2+1/k})$ many "actual" neighbors[9]. We then apply Lemma 2 on these superclusters by computing the $(2k-1)$-spanner for each of these bipartite graphs obtained taking the vertices of the supercluster to be on one side of the bipartition, A, and their "actual" neighbors on the other side B. Since there are $O(n^{1/2})$ superclusters, this adds $O(n^{1/2+1/k} \cdot n^{1/2}) = O(n^{1+1/k})$ edges to the spanner.

Finally to provide a good stretch in the spanner for all the edges in G between vertices of the same supercluster[10], we simply recurse inside each supercluster– this can be done efficiently since the superclusters are vertex disjoint (as they contain sets of vertex disjoint clusters), and each of supercluster has $O(\sqrt{n})$ vertices.

Roadmap. The structure of the paper is as follows. We start by considering in Section 3 the simplified case of 3-spanners (hence $k = 2$) and present a deterministic construction with $O(\log n)$ rounds. Section 4 considers the general case of $k > 2$. In Sec. 4.1, we first describe an $O(n^{1-1/k})$-round algorithm that already contains some of the ideas of the final algorithm. Then, before presenting the algorithm, we describe the two key tools that it uses. For didactic reasons, in Sec. 4.2, we first describe the construction of sparser spanners for unbalanced bipartite graphs. Only later 4.3, we present the new notion of superclusters. Finally, in Sec. 4.4, we show how these tools can be used to construct $(2k-1)$-spanners for graphs of low diameter. The extension for general graphs is deferred to the full version [12].

2 Preliminaries, Notation and Model

Notations and Definitions. We consider an undirected unweighted n-vertex graph $G = (V, E)$ where V represents the set of processors and E is the set of links between them. Let $\mathrm{diam}(H)$ be the diameter of the subgraph $H \subseteq G$. We denote the diameter of G by $D = \mathrm{diam}(G)$. For $u, v \in V(G)$ and a subgraph H, let $\texttt{dist}(u, v, H)$ denote the $u-v$ distance in the subgraph $H \subseteq G$. When $H = G$, we omit it and write $\texttt{dist}(u, v)$. Let $\Gamma(u) = \{v \mid (u, v) \in E\}$ be the set of u's neighbors in G and $\Gamma^+(u) = \Gamma(u) \cup \{u\}$. For a subset of the vertices $V' \in V$, let $\Gamma(V', G) = \bigcup_{u \in V'} \Gamma(u)$ and $\Gamma^+(V', G) = \Gamma(V', G) \cup V'$. Let $\Gamma_i(v) = \{u \mid \texttt{dist}(u, v) \leq i\}$, for a subset V', $\Gamma_i(V')$ is defined accordingly.

For a subgraph $H \subseteq G$, let $E(v, H) = \{(u, v) \in E(H)\}$ be the set of edges incident to v in the subgraph H and let $\deg(v, H) = |E(v, H)|$ denote the degree of node v in G. For a set

[8] using the collection of edge disjoint $O(1)$-depth trees that connect their cluster centers.
[9] This term is informal, the actual neighbors are the vertices that are no longer clustered by the current clustering.
[10] Vertices whose clusters belong to the same supercluster.

of vertices C, let $G(C)$ be the induced graph of G on C. We say the a tuple $(a, b) > (c, d)$ if $a > c$ or $a = c$ but $b > d$.

Spanners and Clustering. A subgraph $H \subseteq G$ is a $(2k-1)$-*spanner* if $\texttt{dist}(u, v, H) \leq (2k-1)\texttt{dist}(u, v, G)$ for every $u, v \in V$. Given a graph G, a subgraph H, and an edge $e = (u, v)$ in G, we define the *stretch* of e in H to be the length of the shortest path from u to v in H. If no such path exists, we say that the stretch is infinite. We say that an edge $e = (u, v)$ is *taken care of* in H if $\texttt{dist}(u, v, H) \leq (2k-1)$.

A *cluster* is a connected set C of vertices of the original graph. Often, a cluster will have one of its vertices $s \in C$ be the *cluster center*. The ID of the cluster is the ID of its center. Two clusters C_1 and C_2 are *neighbors* if $\Gamma(C) \cap C' \neq \emptyset$. For a subset of vertices $S \subseteq V$, the diameter of the subset is simply the diameter of the induced graph G on S.

Ruling Sets. An (α, β)-*ruling set* with respect to G and $V' \subseteq V$ is a subset $U \subseteq V'$ satisfying the following: (I) All pairs $u, v \in U$ satisfy $\texttt{dist}(u, v) \geq \alpha$, (II) For all $v \in V'$, there exists a $u \in U$ such that $\texttt{dist}(u, v) \leq \beta$.

The Communication Model. We use a standard message passing model, the CONGEST model [14], where the execution proceeds in synchronous rounds and in each round, each node can send a message of size $O(\log n)$ to each of its neighbors. In this model, local computation is done for free at each node and the primary complexity measure is the number of communication rounds. Each node holds a processor with a unique and arbitrary ID of $O(\log n)$ bits. Throughout, we assume that the nodes know a constant approximation on the number of nodes n, same holds also for the randomized algorithm of Baswana-Sen[11].

3 3-Spanners in $\widetilde{O}(1)$ Rounds

The key building block of the algorithm is the construction of a linear sized 3-spanner for a $\sqrt{n} \times n$ bipartite graph. A similar idea already appeared in [6], but using $O(n)$-bit messages.

The core construction: 3-spanners for unbalanced bipartite graphs

▶ **Lemma 4.** *Let $G = (A, B, E)$ be a (possibly weighted) bipartite graph where $|A| = O(\sqrt{n})$, $|B| = O(n)$, and each vertex knows whether it is in A or in B. Then one can construct (in the* CONGEST *model) a 3-spanner H with $O(n)$ edges within* two *rounds.*

Algorithm Bipartite3Spanner first forms $|A|$ vertex-disjoint star clusters (clusters of radius 1), each centered at a vertex of A. To do that, every vertex $v_b \in B$ picks one of its neighbors $v_a \in \Gamma(v_b) \cap A$ to be its cluster center and sends the ID of its chosen neighbor v_a to all of its neighbors. We write $c(v_b) = v_a$ to denote that the cluster center of v_b is v_a.

All edges $(v_b, c(v_b))$ are added to the spanner H. At this point, the graph contains $O(\sqrt{n})$ clusters centered at the vertices in A. We say that two stars S_1 and S_2 are *neighbors* if the *center* of S_1 has a neighbor in S_2, or vice-versa. Note that because the graph is bipartite, this is the only possible connection between clusters. Then, for each vertex u_a in A, and each neighboring star-cluster u'_a, the vertex u_a adds to the spanner H one edge to one of its neighbors in the cluster of u'_a. For a complete description of the algorithm see Alg. 1.

[11] In Baswana-Sen, each center samples itself with probability $n^{-1/k}$, which requires knowing n.

Algorithm 1 Bipartite3Spanner($G = (A \cup B, E)$) for $|A| = O(\sqrt{n})$ and $|B| = O(n)$.

1: $H \leftarrow \emptyset$
2: Each vertex $v_b \in B$ selects an arbitrary neighboring vertex $v_a \in A$, assigns $c(v_b) = v_a$ and send $c(v_b)$ to all its neighbors. It adds the edge $(v_b, c(v_b))$ to H.
3: Each vertex $u_a \in A$ does the following (in parallel):
4: **for** each ID v_a received **do**
5: Pick a single neighbor v_b satisfying $c(v_b) = v_a$. Add the edge (u_a, v_b) to H.

To adapt the algorithm for the weighted case, we simply let each $v_b \in B$ pick its closest neighbor in A. In addition, each vertex u_a connects to its closest neighbor in each star of u'_a. It is easy to see that the algorithm takes 2 rounds. In the full version [12], we show:

▶ **Lemma 5.** *The output H of Alg.* Bipartite3Spanner *is a 3-spanner with $O(n)$ edges.*

Constructing 3-spanners for general graphs in $O(\log n)$ rounds. Let $V_h = \{v \in V \mid \deg(v, G) \geq \sqrt{n}\}$ be the set of *high* degree vertices in G and let $V_\ell = V \setminus V_h$ be the remaining *low*-degree vertices. First, the algorithm adds to the spanner H, all the edges of the low-degree vertices V_ℓ. Then, it proceeds by partitioning (in a way that will be described later) the *high-degree* vertices V_h into $t = O(\sqrt{n})$ balanced sets $V_1, \ldots, V_t$. This partition gives rise to t bipartite $\sqrt{n} \times n$ graphs B_i obtained by taking V_i to be on one side of the partition and $V \setminus V_i$ on the other side. We describe the partitioning procedure in Lemma 6. We can then apply Algorithm Bipartite3Spanner to construct 3-spanners for all these subgraphs in parallel. Finally, we simply add to H, all the internal edges $V_i \times V_i$ for every i, again adding total of $t \cdot O(n)$ edges.

Algorithm Improved3Spanner

- **(S0) Handling Low-Degree Vertices:** Add to H all edges in $(V_\ell \times V) \cap E(G)$.
- **(SI) Balanced Partitioning of High-Degree Vertices V_h:** Partition the high-degree vertices of V into $\Theta(\sqrt{n})$ sets $V_1, \ldots, V_t$ each with $O(\sqrt{n})$ vertices.
- **(SII) Taking care of edges $V_i \times (V \setminus V_i)$, for every** $i \in \{1, \ldots, t\}$:
  - Define $B_i = (V_i, V \setminus V_i)$ for every $i \in \{1, \ldots, t\}$.
  - Construct a 3-spanner $H_i \subseteq B_i$ by applying Algorithm Bipartite3Spanner on each of the B_i graphs in parallel, for every i.
- **(SIII) Taking care of edges $V_i \times V_i$,** $i \in \{1, \ldots, t\}$: Add to H all edges in $V_i \times V_i$ for every $i \in \{1, \ldots, t\}$.

Note that eventhough the bipartite graphs B_i are not vertex disjoint, each edge belongs to at most two such graphs, and hence we can construct the 3-spanners for all B_i in parallel. It is also easy to see that the final spanner has $O(n^{3/2})$ edges.

The only missing piece at that point concerns the computation of partitioning V_h.

Balanced partitioning of V_h in $O(\log n)$ rounds. The partition procedure starts by computing $(4, O(\log n))$-ruling set $R \subseteq V$ for the high-degree vertices V_h. We will use the following lemma that uses standard technique for constructing $(t, t \log n)$-ruling sets.

▶ **Lemma 6.** *Given a graph $G = (V, E)$ and a subset $V_h \in V$ of the vertices, one can compute in $O(\log n)$ rounds in the* CONGEST *model, a $(4, O(\log n))$-ruling set $U \subseteq V_h$ with respect to G and the high-degree vertices V_h.*

We now view each of the vertices $r \in R$ as a center of a cluster of diameter $O(\log n)$: let each high-degree vertex join the cluster of the vertex closest to it in R, breaking ties based on IDs. Since every vertex in V_h is at distance $O(\log n)$ from R, all the vertices V_h will be clustered within $O(\log n)$ rounds. Each vertex r in R can then partition the vertices of its cluster into subsets of size $\lfloor\sqrt{n}\rfloor$, and an additional leftover subset of size at most $\sqrt{n}$ (this can be done using balanced partitioning lemma, Lemma 9). We now claim that this partition is balanced. Clearly, all sets are of size $O(\sqrt{n})$, so we just show that there $O(\sqrt{n})$ subsets. Since every $r \in R$ is high-degree and since every two vertices in R are at distance at least 4, we have that $|R| = O(\sqrt{n})$. For each $r \in R$, there is at most one subset of size less than $\lfloor\sqrt{n}\rfloor$. Therefore, there are $O(\sqrt{n})$ subsets of size less than $\lfloor\sqrt{n}\rfloor$. All other subsets are of size $\lfloor\sqrt{n}\rfloor$. However, there can be at most $O(\sqrt{n})$ disjoint subsets of size $\lfloor\sqrt{n}\rfloor$, hence there are $O(\sqrt{n})$ subsets in total, as desired.

We conclude by showing:

▶ **Lemma 7** (3-Spanner Given Partition). *Given a (possibly weighted) n-vertex graph $G = (V, E)$ with a vertex-partition $V_1, V_2, \ldots, V_t$ such that $|V_i| = O(\sqrt{n})$ and $t = O(\sqrt{n})$, one can construct a 3-spanner H of size $O(n^{3/2})$ in 2 rounds in the* CONGEST *model.*

Finally, if the vertices IDs are bounded, two rounds are sufficient to construct the spanner.

▶ **Theorem 8** (Small IDs). *Given a graph $G = (V, E)$ where the IDs of the vertices have $\log(n) + O(1)$ bits, one can construct a 3-spanner H of G with $|H| = O(n^{3/2})$ edges in two rounds in the* CONGEST *model.*

4 $(2k-1)$ Spanners

The structure of Baswana-Sen clustering. At the heart of the algorithm is a construction of $(k-1)$-levels of clustering $\mathcal{C}_0, \ldots, \mathcal{C}_{k-1}$. The initial clustering $\mathcal{C}_0 = \{\{v\}, v \in V\}$ simply contains n singleton clusters. For every i, each cluster $C \in \mathcal{C}_i$ has a cluster center z and we denote by Z_i the collection of cluster centers. We define $V_i = \bigcup_{z \in Z_i} \Gamma_i(z)$. A vertex v is *i-clustered* if $v \in V_i$, otherwise it is i-unclustered. Hence V_i is the set of clustered vertices appearing in the clusters of $\mathcal{C}_i$. The algorithm consists of $k-1$ steps where at the end of step $i \in \{1, \ldots, k-1\}$, we have an i^{th}-level clustering $\mathcal{C}_i = \{C_1, \ldots, C_\ell\}$ and a partial spanner H_i that satisfies the following: (P1) The clustering $\mathcal{C}_i$ contains $\ell = O(n^{1-i/k})$ clusters. (P2) For each cluster $C_j \in \mathcal{C}_i$ with a center z_j, the subgraph H_i contains a BFS tree $T_i(C)$ of depth at most i that spans all the vertices of C (i.e., the vertices of $T_i(C)$ are precisely C) and (P3) For every $u \in V_{i-1} \setminus V_i$, and every $v \in \Gamma(u)$, $\texttt{dist}(u, v, H_i) \leq 2k-1$.

High-Level Description of Phase i in Baswana-Sen Algorithm

- **(SI) Selecting $O(n^{1-i/k})$ cluster centers** $Z_i \subseteq Z_{i-1}$. In the randomized algorithm, this is done by sampling each center in Z_{i-1} independently with probability $n^{-1/k}$. The i-clustered vertices are $V_i = \Gamma_i^+(Z_i)$.
- **(SII) Taking care of unclustered vertices** $V_{i-1} \setminus V_i$. That is, taking care of the vertices that stopped being clustered at that point.
- **(SIII) Forming the clusters of $\mathcal{C}_i$ around** Z_i. This is done by letting each $u \in V_i$ join the cluster of its *closest* center in Z_i breaking tie based on ID's. The latter can be implemented in $O(i)$ rounds of constructing BFS trees of depth i from all centers Z_i while breaking ties appropriately.

At the final phase of Baswana-Sen, there are $O(n^{1/k})$ clusters in $\mathcal{C}_{k-1}$ and at that point, each vertex $v \in V$ adds one edge to each of its neighboring clusters in $\mathcal{C}_{k-1}$.

Note that the only step that uses randomness in this algorithm is sub-step (SI), and the other two sub-steps (SII-SIII) and the final phase are completely deterministic. Our challenge is to implement sub-step (SI) deterministically in a way that in sub-step (SII) we do not add too many edges to the spanner. The algorithms presented from now on, will simulate the i^{th} phase of Baswana-Sen only without using randomness. Sub-step (SIII) and the final phase will be implemented exactly as in Baswana-Sen.

4.1 Take (I): $O(n^{1-1/k})$-Round Algorithm NaiveSpanner

It is easy to see that 0^{th}-level clustering containing n singleton clusters satisfies properties (P1-P3). To simulate the i^{th} phase of Baswana-Sen algorithm, we employ $O(i \cdot n^{1-i/k})$ deterministic rounds: Initially, we unmark all the vertices and over time, some of the vertices will get marked (i.e., indicating that they are i-clustered). The procedure consists of $O(n^{1-i/k})$ steps where at each step, we look at the remaining set Z'_{i-1} of cluster centers in Z_{i-1} that have not yet been added to Z_i. Let U be the current set of unmarked vertices and let $\mathcal{C}'_{i-1} \subseteq \mathcal{C}_{i-1}$ be the corresponding clusters of Z'_{i-1}. For each cluster $C \in \mathcal{C}_{i-1}$, define its unmarked neighborhood by $\Gamma^U(C) = \bigcup_{u \in C} \Gamma(u) \cap U$ and its current unmarked-degree by $\deg^U(C) = |\Gamma^U(C)|$. We say that cluster C is a *local-maxima* in its unmarked neighborhood if it has the maximum tuple (lexicographically) $(\deg^U(C), ID(C))$ among all other clusters C' that have mutual unmarked neighbors (i.e., $\Gamma^U(C) \cap \Gamma^U(C') \neq \emptyset$).

Phase i of Algorithm NaiveSpanner

(SI) Defining the centers Z_i.
Set $Z'_{i-1} \leftarrow Z_{i-1}$, $U = V$ and for $O(n^{1-i/k})$ steps do the following:
- Every center $z \in Z'_{i-1}$ of cluster C computes $\deg^U(C)$.
- Every center $z \in Z'_{i-1}$ whose cluster C has the maximum tuple $(\deg^U(C), ID(C))$ in its unmarked neighborhood, $\deg^U(C)$, joins Z_i only if $\deg^U(C) \geq n^{i/k}$.
- Remove from Z'_i the centers $z \in C$ that join Z_i and mark $\Gamma^U(C)$.

(SII) Taking care of unclustered vertices.
- Let $\mathcal{C}'_{i-1}$ be the clusters whose centers did *not* join Z_i.
- For every unmarked vertex u, add one edge per neighboring cluster in $\mathcal{C}'_{i-1}$.

(SIII) Forming the $\mathcal{C}_i$ clusters centered at Z_i. As in Baswana-Sen.

Sketch of the Analysis. The key part to notice is that by picking the local-maxima clusters, we have that for any two cluster-centers $z_1 \in C_1, z_2 \in C_2$ that join Z_i, their unmarked neighborhoods $\Gamma^U(C_1), \Gamma^U(C_2)$ are vertex disjoint, hence Z_i contains $O(n^{1-i/k})$ centers; in addition, after $O(n^{1-i/k})$ steps, the clusters of all remaining centers have $O(n^{i/k})$ unmarked neighbors. Hence, at step (SII), total of $O(n^{1-(i-1)/k}) \cdot O(n^{i/k}) = O(n^{1+1/k})$ edges are added to the spanner. Turning to runtime, we claim that each of the $O(n^{1-i/k})$ steps can be implemented in $O(i)$ rounds. Since each cluster $C \in \mathcal{C}_{i-1}$ is connected i G by a depth-i tree, and since trees of different clusters are vertex-disjoint, computing the unmarked degree $\deg^U(C)$ of each cluster C can be done in $O(i)$ rounds; To avoid the double of counting of unmarked vertices that have many neighbors at the same cluster, each unmarked vertex respond to only one its neighbors in each cluster. Similarly, also selecting the local maxima clusters can be done in $O(i)$ rounds. We note that the time complexity of the algorithm is $O(n^{1-1/k})$, as opposed to $O(n)$, since after $O(n^{1-1/k})$ iterations of finding clusters of locally maximal unmarked degree, all remaining clusters will have low unmarked degree, and can

be dealt with in parallel in $O(1)$ rounds, by adding an edge to every unmarked neighbor. Towards speeding up this algorithm, we now introduce our key technical tools.

A remark regarding step (SII). Let $V'_i = V_{i-1} \setminus V_i$ be the set of newly unclustered vertices. In Baswana-Sen algorithm, step (SII) takes care of all the edges in $V'_i \times V$. That is, the edges added to the spanner H at that stage provide that $\texttt{dist}(u, v, H) \leq 2i - 1$ for every $(u, v) \in (V'_i \in V) \cap E$. Most of the algorithms we present in this paper, have a weaker but sufficient guarantee when implementing step (SII). In particular, we only add edges between the remaining unmarked vertices and the remaining clusters whose centers did not join Z_i. We now show why it is sufficient. Consider an edge $(u, v) \in E$. Let i_u be the largest level of the clustering such that u is i_u-clustered, define the same for v. Without loss of generality, assume that $i_v \leq i_u$.

Case (1): $i_u = k - 1$: Let C be the cluster of u in $\mathcal{C}_{k-1}$. Since in the last step, v adds one edge to $\Gamma(u) \cap C$, the claim holds.

Case (2): $i_u \leq k - 2$: Consider phase $(i_u + 1)$ where the clustering $\mathcal{C}_{i_u+1}$ is constructed given $\mathcal{C}_{i_u}$.

By definition, in step (SII) of phase $(i_u + 1)$ we have that the vertex v is unmarked and the vertex u belongs to a remaining cluster $C \in \mathcal{C}_{i_u}$. Since every unmarked vertex adds one edge to each remaining cluster, we have that v added one edge to $C \cap \Gamma(v)$. The claim follows.

4.2 Key Tool (I): Sparser Spanner for Unbalanced Bipartite Graphs

In this section, we consider Lemma 2. Similarly to the construction of 3-spanners in Section 3, a key ingredient in our algorithm is the construction of *sparser* spanners for unbalanced $A \times B$ bipartite graphs for $|A| \leq |B|$. The algorithm of [6] constructs a $(2k - 1)$ spanners for these bipartite graphs with $O(|A||B|^{2/k})$ edges in the LOCAL model, using large messages. Our algorithm is slower than that of [6], but has the benefit of obtaining a sparser $(2k-1)$-spanner with only $O(k|A|^{1+2/k} + |B|)$ edges and while using $O(\log n)$-bit messages.

The high-level strategy of Alg. SparserBipartiteSpanner is to first compute $|A|$ star clusters (clusters of radius 1) by letting each vertex of B join an arbitrary neighbor in A. Hence, after one step of clustering, we have $|A|$ clusters rather than $O(n^{1-1/k})$ clusters is in Baswana-Sen. We then consider *star graph* G_S obtained contracting each star into a vertex, and essentially apply Alg. NaiveSpanner on the star-graph G_S to construct a $(k-1)$-spanner $H_S \subseteq G_S$ with $O(|A|^{1+2/k})$ edges within $O(k|A|^{1-2/k})$ rounds. To get a $(2k - 1)$ spanner $H \subseteq G$ from H_S, for every star-edge $(S_i, S_j) \in H_S$, add a single edge in $(S_1 \times S_2) \cap E$ to H. Finally, adding the star edges to the spanner, gives a total of $O(|A|^{1+2/k} + |B|)$ edges. Simulating Alg. NaiveSpanner on the star-graph in the CONGEST model requires some effort. The description of Alg. SparserBipartiteSpanner and its analysis is in the full version [12].

4.3 Key Tool (II): Superclustering – Grouping Baswana-Sen Clusters

Why Superclusters? In this section, we describe the main tool that allows us to speed up Alg. NaiveSpanner by a factor of $\sqrt{n}$. The idea is to group the $n^{1-i/k}$ clusters in the i^{th}-clustering $\mathcal{C}_i$ into $\sqrt{n}$ superclusters, each containing $O(n^{1/2-i/k})$ clusters. Then, instead of iterating over clusters one by one (as in Alg. NaiveSpanner), we iterate over the superclusters. Each time, either *all* the cluster centers of a given supercluster join the next level of clustering, or none of them join. As will be shown later, in order to construct the i^{th}-clustering $\mathcal{C}_i$, it will be sufficient for our algorithm to consider $n^{1/2-1/k}$ superclusters (and not all $\sqrt{n}$ superclusters), hence yielding the round complexity of $O(n^{1/2-1/k})$ (for fixed k). For a supercluster to compute the number of its (unmarked) neighbors, all cluster centers in a given

supercluster should be able to communicate efficiently. For that purpose, we make sure that the cluster centers in each supercluster are connected by an $O(2^k)$-depth tree[12], and that the trees of different superclusters are edge-disjoint. These trees will be used for communication purposes, and will allow us to aggregate information to leaders of all superclusters in parallel.

Defining the Superclusters. Let $\mathcal{C}_i$ be a collection of $O(n^{1-i/k})$ i-clusters. A *supercluster* $SC_{i,j} = \{C_{j_1}, \ldots, C_{j_\ell}\}$ is a collection of clusters from $\mathcal{C}_i$. A *Superclustering* $\mathcal{SC}_i = \{SC_{i,1}, \ldots, SC_{i,p}\}$ is a covering partition of all clusters from $\mathcal{C}_i$. That is, $\bigcup_{j=1}^{p} SC_{i,j} = \mathcal{C}_i$, and the superclusters are cluster-disjoint (every cluster in $\mathcal{C}_i$ belongs to exactly one supercluster). To select the cluster centers of level i, the algorithm constructs in each phase $i \in \{1, \ldots, k/2\}$ a superclustering $\mathcal{SC}_i$ which satisfies some helpful properties. We call a superclustering satisfying these properties a *nice superclustering*. Before defining the properties of a nice supercluster, we introduce some notation. For a supercluster $SC_{i,j} = \{C_{j_1}, \ldots, C_{j_p}\}$, let $V(SC_{i,j}) = \bigcup_{C \in SC_{i,j}} C$ be the set of all vertices in its clusters and $N_V(SC_{i,j}) = |V(SC_{i,j})|$ be the number of vertices in the supercluster $SC_{i,j}$. Also, let $N_C(SC_{i,j})$ denote the number of clusters that the supercluster $SC_{i,j}$ contains. A supercluster $SC_{i,j}$ with only one cluster (i.e., $N_C(SC_{i,j}) = 1$) is called a *singleton*. In addition, a singleton supercluster is called a *small-singleton* if $N_V(SC_{i,j}) \leq \sqrt{n}$ (otherwise, if $N_V(SC_{i,j}) > \sqrt{n}$, it is a *large-singleton*). Our $(2k-1)$-spanner construction is based upon the construction of superclusters with some *nice* useful properties, as defined next.

Nice Superclustering. A superclustering $\mathcal{SC}_i = \{SC_{i,1}, \ldots, SC_{i,\ell}\}$ is *nice* if it contains $\ell = O(\sqrt{n})$ superclusters, and each of these superclusters $SC_{i,j} \in \mathcal{SC}_i$ satisfies the following:

(N0) Singleton: If $N_V(SC_{i,j}) = \Omega(\sqrt{n})$, then $N_C(SC_{i,j}) = 1$.

Every non-singleton supercluster $SC_{i,j}$ (i.e., every supercluster containing at least two clusters) satisfies:

(N1) Cluster Balance: $N_C(SC_{i,j}) = O(n^{1/2-i/k})$, and

(N2) Vertex Balance: $N_V(SC_{i,j}) = O(\sqrt{n})$.

(N3) Connectivity: In the graph G, each $SC_{i,j} \in \mathcal{SC}_i$ has a tree $T(SC_{i,j})$ of depth[13] $O(2^k)$. In addition, the trees $T(SC_{i,1}), \ldots, T(SC_{i,\ell})$ are edge-disjoint.

Intuitive discussion of these properties. Property (N0) implies that if a supercluster has many vertices (more than $\sqrt{n}$), then it is a singleton supercluster. Property (N1) implies that non-singleton superclusters with at least two clusters are balanced with respect to the number of *clusters* from $\mathcal{C}_i$ that they contain. Since there are $O(n^{1-i/k})$ clusters in the i^{th} clustering, dividing it "fairly" between $\sqrt{n}$ superclusters yields this bound. Property (N2) also implies a balance among non-singleton superclusters, but this time with respect to the number of vertices. Finally, Property (N3) provides the existence of a $O(2^k)$-depth tree that connects the cluster centers of that supercluster. This "weird" looking depth of $O(2^k)$ shows up when computing the 0^{th}-level superclustering for general graphs (for graphs of constant diameter a much simpler construction exists). In particular, it shows up in Step (SI) of Alg. ConsZeroSuperclustering [12]. Finally, (N4) requires these trees to be edge-disjoint to allow communication within different superclusters, *in parallel* without congestion. As will

[12] This bound arises in the algorithm for graphs with general diameter, in the full version, and will be discussed later on.

[13] When the diameter of the original graph G is $O(1)$, the diameter of $T(SC_{i,j}) = O(1)$. The term $O(2^k)$ appears when dealing with graphs of large diameter, as described in the full version.

be shown in the next subsection, to satisfy Properties (N1) and (N2), the construction of the i^{th}-level of superclustering requires to partition both the vertices and the clusters into *balanced* the $\sqrt{n}$ superclusters. The key tool to achieve it is the following:

The Balanced Partitioning Lemma. The input to the partitioning lemma is a vertex-weighted tree T, where every vertex v in T has a non-negative weight $w(v)$ and in addition, we are given a bound B on the allowed total weight of each tree. The goal is to partition the tree into edge-disjoint subtrees, such that, all but one of the subtrees have a weight in $[B, 2B]$. The lemma achieves this but with some subtle specification. It partitions the vertices of the tree T into p disjoint sets: $\widehat{V}(T_0), \widehat{V}(T_1), \ldots, \widehat{V}(T_p)$. The total weight of each set $\widehat{V}(T_i)$, except for at most one, $\widehat{V}(T_0)$, is bounded by $[B, 2B]$. Hence, the partition respects the weight bound. Next, each set $\widehat{V}(T_i)$ is connected by a subtree $T_i \subseteq T$. The important feature of these trees T_i is that they might contain an additional vertex $v \in V(T) \setminus \widehat{V}(T_i)$. This additional vertex v, if exists, is the root of T_i and it is essential to connect the vertices in $\widehat{V}(T_i)$. Intuitively, this additional vertex helps us to communicate between the vertices of $\widehat{V}(T_i)$. Even though the trees T_i are not vertex disjoint, they are shown to be edge disjoint, which is sufficient for our applications.

▶ **Lemma 9** (Balanced Partitioning Lemma). *In $O(\mathrm{diam}(T))$ rounds, one can construct subtrees $T_0, T_1, \ldots, T_p \subseteq T$, with roots $r(T_0), r(T_1), \ldots r(T_p)$ and corresponding* disjoint *vertex sets $\widehat{V}(T_0), \widehat{V}(T_1), \ldots, \widehat{V}(T_p)$ such that*

(D1) *The $\widehat{V}(T_i)$ sets are vertex disjoint and $\bigcup_{i=1}^{p} \widehat{V}(T_i) = V(T)$.*

(D2) *$W(T_i) \in [B, 2B]$ for every $i \geq 1$, and $W(T_0) \leq 2B$ where $W(T_i) = \sum_{u \in \widehat{V}(T_i)} w(u)$.*

(D3) *$V(T_i) = \widehat{V}(T_i) \cup r(T_i)$.*

(D4) *All $T_0, \ldots, T_p$ are edge-disjoint and with diameter at most* $\mathrm{diam}(T)$.

Intuitively, the important vertex set of the tree T_i is the set of vertices $\widehat{V}(T_i)$ and hence the weight of the tree in Property (D2) is defined by summing over all these vertices (instead of summing over all vertices in the tree). Property (D3) implies that the tree T_i might contain, in addition to $\widehat{V}(T_i)$, also an additional vertex – its root – that allows the connectivity of the set $\widehat{V}(T_i)$ in T_i. The full proof of Lemma 9 appears in [12]. In the common application of this lemma, the tree T is a tree that connects the cluster-centers of a given supercluster, these cluster-centers are given a weight (e.g., the size of their cluster) and the remaining vertices in T are given a zero weight. The bound corresponds to the maximum allowed number of clusters (or vertices) in the supercluster (as in Section 4.3, (N2,N3)).

4.4 Take (II): $(2k-1)$-Spanners in $O(2^k \cdot n^{1/2-1/k})$ Rounds

We first consider the construction for graphs with constant diameter. At the end of the section, we discuss the extension for general graphs with arbitrary diameter. Recall that for $i \leq k/2$, $\mathcal{C}_i$ is a clustering that contains $O(n^{1-i/k})$ vertex-disjoint clusters centered at the vertices Z_i. The set of i-clustered vertices V_i are in $\Gamma_i(Z_i)$.

The first part of the algorithm contains $k/2$ phases. In each phase $i \in \{1, \ldots, k/2\}$, we are given a $(i-1)^{th}$ nice superclustering $\mathcal{SC}_{i-1}$ (whose superclusters contain the clusters of $\mathcal{C}_{i-1}$) and the current spanner H. We then construct the i^{th} nice superclustering $\mathcal{SC}_i$ and add edges to H in order to take care of the newly unclustered vertices in $V_{i-1} \setminus V_i$. At the end of the first part, we have a $(k/2)^{th}$ superclustering $\mathcal{SC}_{k/2}$ with $O(\sqrt{n})$ clusters. At that point, the number of clusters is small enough, and so Alg. NaiveSpanner can be applied.

Constructing the 0^{th}-level superclustering $\mathcal{SC}_0$ in $O(\mathrm{diam}(G))$ rounds. To compute $\mathcal{SC}_0$, we apply the Partitioning Lemma 9 on a BFS tree T rooted at some arbitrary vertex (e.g., of maximum ID) using weights of $w(v) = 1$ for each $v \in V$ and bound $B = O(\sqrt{n})$. This partitions the vertices into $\Theta(\sqrt{n})$ subsets S_i, each of size $O(\sqrt{n})$. Each such subset $S_i = \{v_{i,0}, \ldots, v_{i,\ell}\}$ defines a supercluster $SC_{0,i} = \{\{v_{i_0}\}, \ldots, \{v_{i,\ell}\}\}$ containing the singleton clusters of S_i's vertices. By that, we get $O(\sqrt{n})$ superclusters $\mathcal{SC}_0 = \{SC_{0,1}, \ldots, SC_{0,\sqrt{n}}\}$. By the partitioning lemma, we also have a tree T_i for each $SC_{0,i}$, satisfying Prop. (N3).

The i^{th} phase of Algorithm ImprovedSpanner for $i \in \{1, \ldots, k/2\}$. At the beginning of the phase, we are given the $(i-1)^{th}$-clustering $\mathcal{C}_{i-1}$ grouped into the nice superclustering $\mathcal{SC}_{i-1}$. Our first goal is to use the superclustering $\mathcal{SC}_{i-1}$ to define the set of new $O(n^{1-i/k})$ cluster centers Z_i. The high-level idea here is to implement Alg. NaiveSpanner on each supercluster rather than on each cluster. Given a set U of unmarked vertices and a supercluster $SC \in \mathcal{SC}_{i-1}$, define its *unmarked neighborhood* and *unmarked degree* by

$$\Gamma^U(SC) = \bigcup_{v \in V(SC)} (\Gamma^+(v) \cap U) \quad \text{and} \quad \deg^U(SC) = |\Gamma^U(SC)| \ . \tag{1}$$

Similarly to before, we say that a supercluster SC is a *local-maxima* in its unmarked neighborhood, if for every other SC' such that $\Gamma^U(SC) \cap \Gamma^U(SC') \neq \emptyset$, it holds that

$$(\deg^U(SC), ID(SC)) > (\deg^U(SC'), ID(SC')).$$

We say that supercluster SC has *low-expansion* if $\deg^U(SC) \leq n^{1/2+1/k}$. Otherwise, it has *high-expansion*. Note that unlike the previous algorithms presented before, here the expansion threshold $n^{1/2+1/k}$ is independent[14] of the level i.

Step (S1) of phase i: Selecting the centers Z_i. Selecting the $O(n^{1-i/k})$ cluster centers of Z_i is done in $O(n^{1/2-1/k})$ iterations. We start by unmarking all vertices. At each iteration, we have a set U of remaining unmarked vertices and a subset of remaining superclusters $\mathcal{SC}'_{i-1}$ of superclusters whose cluster centers have not yet been added to Z_i. All superclusters $SC \in \mathcal{SC}'_{i-1}$ compute their unmarked degree $\deg^U(SC)$ in parallel. (This can be done in $O(i \cdot 2^k)$ rounds thanks to Prop. (N3) in Section 4.3).

▶ **Definition 10** (Successful Supercluster)**.** A supercluster SC that is local-maxima in its unmarked neighborhood and has high-expansion, that is $\deg^U(SC) \geq n^{1/2+1/k}$, is called a *successful* supercluster.

It is easy to see that the leader (vertex v of maximum ID in $V(SC)$) of every supercluster SC can verify in $O(i \cdot 2^k)$ rounds whether it is a local-maxima in its unmarked neighborhood.

In the algorithm, each successful supercluster SC adds all its cluster centers to Z_i, and mark all the vertices in $\Gamma^U(SC)$. This continues for $O(n^{1/2-1/k})$ iterations.

As we will show in the analysis section, since the unmarked neighborhoods of successful superclusters are disjoint and large, there are at most $O(n^{1/2-1/k})$ such superclusters. In addition, by Prop. (N2), each supercluster has $O(n^{1/2-(i-1)/k})$ clusters, overall $|Z_i| = O(n^{1/2-(i-1)/k} \cdot n^{1/2-1/k}) = O(n^{1-i/k})$ as desired.

[14] The intuition is that in each level i, the superclusters have at most $\sqrt{n}$ vertices, and we say that it has high expansion if the size of its neighborhood size is factor $n^{1/k}$ larger.

Step (S2) of phase i: Taking care of unclustered vertices. After $O(n^{1/2-1/k})$ steps of computing successful superclusters, all remaining superclusters SC have low-expansion with respect to the remaining unmarked vertices U'. That is, $\deg^{U}(SC) \leq n^{1/2-1/k}$. First, we take care of the singleton superclusters.

(S2.1): Singleton supercluster SC with low-expansion. Each unmarked vertex $u \in U'$ add to H an edge to one of its neighbor in $\Gamma(u) \cap V(SC)$. Since there are $O(\sqrt{n})$ superclusters, each with $\deg^{U'}(SC) \leq n^{1/2+1/k}$, overall we add $O(n^{1+1/k})$ such edges.

(S2.2): Non-singleton superclusters SC with low-expansion. Here, the construction of sparser spanners for bipartite graphs comes into play (see Sec. 4.2). Recall that by Prop. (N2), $N_V(SC) = O(\sqrt{n})$ vertices. Let $\Gamma^{U',-}(SC) = \Gamma^{U'}(SC) \setminus V(SC)$ be the unmarked neighbors of SC excluding the vertices of the supercluster SC. Since SC has low-expansion, it also holds that $|\Gamma^{U',-}(SC)| = O(n^{1/2-1/k})$. For every such supercluster SC, we consider the bipartite graph $B(SC) = (V(SC), \Gamma^{U',-}(SC))$, and apply Alg. SparserBipartiteSpanner to compute for it a $(2k-1)$-spanner $H(SC) \subseteq B(SC)$ with $O(n^{1/2+1/k})$ edges (see Lemma 2). This is done for all the graphs $B(SC)$ in parallel.

Note that the graphs $B(SC)$ are not necessarily vertex disjoint since an unmarked vertex can appear in several such graphs. The key observation that allows the parallel computation of all these spanners, is that every edge (u, v) can belong to at most *two* bipartite graphs, say, $B(SC)$ and $B(SC')$, where SC, SC' is the supercluster of u, v respectively[15]. Overall, since there are $O(\sqrt{n})$ superclusters, this adds $O(n^{1/2} \cdot n^{1/2+1/k}) = O(n^{1+1/k})$ edges.

Finally, it remains to take care of all edges between vertices belonging to the *same* supercluster. Note that in Alg. NaiveSpanner, there was no need for such a step since all vertices belonging to the same cluster are connected in H by an i-depth BFS tree rooted at the cluster center. However, in our setting, vertices that belong to *different* clusters of the *same* superclusters might still have large stretch (as cluster centers of the same supercluster might be at distance $O(2^k)$ in G). At that point, we use the fact that all superclusters are vertex disjoint and each contains $O(\sqrt{n})$ vertices. We then recursively apply the algorithm ImprovedSpanner on each of these superclusters in parallel. That is, we apply ImprovedSpanner on the induced subgraph on $V(SC)$ for every such supercluster SC.

Note that since in each phase we unmark all the vertices, unclustered vertices can become clustered again and in particular, edges between newly unclustered vertices and clustered vertices will be taken care of later on. This completes the description of the second step.

Steps (SIII) and (SIV): Defining i^{th}-clustering and the i^{th}-superclustering. The clusters $\mathcal{C}_i$ centered at the cluster centers Z_i computed at step (SI) are computed exactly as in Baswana-Sen Algorithm. The depth i-trees of these clusters are added to the spanner. The main challenge here is to *re-group* the new $O(n^{1-i/k})$ clusters into $O(\sqrt{n})$ superclusters, in a way that satisfies all the properties of the nice superclustering mentioned in Section 4.3.

Our starting point is as follows: we have a collection of $O(n^{1/2-1/k})$ successful superclusters $SC \in \mathcal{SC}_{i-1}$ whose cluster centers joined Z_i. Since $\mathcal{SC}_{i-1}$ is nice, by Prop. (N3), each such supercluster $\mathcal{SC}$ has a tree $T(\mathcal{SC})$ of depth $O(2^k)$ that spans all its cluster centers.

First, we let each cluster $C \in \mathcal{C}_i$ with $\Omega(\sqrt{n})$ vertices, to define its own singleton superclusters. Since clusters are vertex disjoint, there are $O(\sqrt{n})$ such superclusters. It now remains to re-group the remaining clusters of $\mathcal{C}_i$ into $O(\sqrt{n})$ superclusters.

[15] Recall that the superclusters share no vertex in common.

For each successful supercluster SC, we now consider only its centers of clusters with $O(\sqrt{n})$ vertices. First, we consider Property (N1) and use Lemma 9 with the tree $T(SC)$, weights $w(z) = 1$ for every cluster-center z of SC (only those that have $O(\sqrt{n})$ vertices in their cluster) and bound $B = O(n^{1/2-i/k})$. All other vertices v' in $T(SC)$ have $w(v') = 0$ (in particular, the centers z of clusters in SC which have been turned into singleton superclusters, we set $w(z) = 0$). By Prop. (N2) for $\mathcal{SC}_{i-1}$, we know that $SC \in \mathcal{SC}_{i-1}$ has $O(n^{1/2-(i-1)/k})$ cluster centers. Hence, the partition procedure will partition each of these superclusters into $O(n^{1/k})$ superclusters $SC_1, \ldots, SC_\ell$. In addition, by Lemma 9(D5), all these resulting superclusters SC_j are equipped with edge-disjoint trees $T(SC_j)$ of diameter $O(2^k)$. Since there are $O(n^{1/2-1/k})$ successful superclusters, overall after this partition there are $O(n^{1/2-1/k}) \cdot O(n^{1/k})$ superclusters.

We then turn to property (N3), and farther partition the superclusters to obtain a balance partition of the *vertices* into superclusters. For that purpose, for each supercluster SC' (obtained from the step above), we again apply the Partitioning Lemma on $T(SC')$. This time we use $B = \sqrt{n}$ and the weight $w(z)$ of each cluster center z in SC' is the number of vertices in its cluster C, that is $w(z) = |C|$ (for clusters v' which have turned into singleton superclusters, or any other non-center vertex in $T(SC')$, we set $w(v') = 0$). Since the vertices of superclusters are disjoint, this step increase the number of superclusters only by an additive $O(\sqrt{n})$ term. Hence, overall the number of superclusters is kept bounded by $O(\sqrt{n})$. This completes the description of the i^{th} phase of Alg. ImprovedSpanner.

The terminating step $k/2$. At the $(k/2)^{th}$ step we have $O(\sqrt{n})$ superclusters, each containing $O(1)$ clusters, hence overall we have $O(\sqrt{n})$ clusters. Now we can afford using Algorithm NaiveSpanner (described near the beginning of Section 4), which iterates over the *clusters* one by one. This completes the description of the algorithm for graph with $\mathrm{diam}(G) = O(1)$. The analysis of stretch, size and round complexity is in the full version [12].

Extension for general graphs of diameter $\mathrm{diam}(G)$. The only step that requires adaptation is that of constructing the 0^{th}-level superclustering $\mathcal{SC}_0$.

▶ **Lemma 11.** *[12] One can construct in $O(2^k \cdot n^{1/2-1/k})$ rounds, nice superclustering $\mathcal{SC}_0 = \{SC_{0,1}, \ldots, SC_{0,p}\}$ along with a subgraph H' with $O(kn^{1+1/k})$ edges such that for every vertex u not participating in the clusters of these superclusters $SC_{0,j} \in \mathcal{SC}_0$, it holds that $\mathtt{dist}(u, v, H') \leq 2k - 1$ for every $v \in \Gamma(u)$.*

References

1. Leonid Barenboim, Michael Elkin, and Cyril Gavoille. A fast network-decomposition algorithm and its applications to constant-time distributed computation. *TCS*, 2016.
2. Surender Baswana and Sandeep Sen. A simple and linear time randomized algorithm for computing sparse spanners in weighted graphs. *Random Structures & Algorithms*, 30(4):532–563, 2007.
3. Keren Censor-Hillel, Merav Parter, and Gregory Schwartzman. Derandomizing local distributed algorithms under bandwidth restrictions. *CoRR*, abs/1608.01689, 2016.
4. Yi-Jun Chang, Tsvi Kopelowitz, and Seth Pettie. An exponential separation between randomized and deterministic complexity in the local model. In *FOCS*, 2016.
5. Bilel Derbel and Cyril Gavoille. Fast deterministic distributed algorithms for sparse spanners. *Theoretical Computer Science*, 2008.
6. Bilel Derbel, Cyril Gavoille, and David Peleg. Deterministic distributed construction of linear stretch spanners in polylogarithmic time. In *DISC*, pages 179–192. Springer, 2007.

7 Bilel Derbel, Cyril Gavoille, David Peleg, and Laurent Viennot. On the locality of distributed sparse spanner construction. In *PODC*, pages 273–282, 2008.

8 Bilel Derbel, Cyril Gavoille, David Peleg, and Laurent Viennot. Local computation of nearly additive spanners. In *DISC*, 2009.

9 Bilel Derbel, Mohamed Mosbah, and Akka Zemmari. Sublinear fully distributed partition with applications. *Theory of Computing Systems*, 47(2):368–404, 2010.

10 Manuela Fischer and Mohsen Ghaffari. Deterministic distributed matching: Simpler, faster, better. *arXiv preprint arXiv:1703.00900*, 2017.

11 Manuela Fischer, Mohsen Ghaffari, and Khun Fabian. Deterministic distributed edge-coloring via hypergraph maximal matching. *arXiv preprint arXiv:1704.02767*, 2017.

12 Ofer Grossman and Merav Pater. Improved deterministic distributed construction of spanners. *arXiv preprint arXiv:1708.01011*, 2017.

13 Michal Hanckowiak, Michal Karonski, and Alessandro Panconesi. On the distributed complexity of computing maximal matchings. *SIDMA*, 15(1):41–57, 2001.

14 David Peleg. *Distributed Computing: A Locality-sensitive Approach*. SIAM, 2000.

15 Seth Pettie. Distributed algorithms for ultrasparse spanners and linear size skeletons. *Distributed Computing*, 22(3):147–166, 2010.

An Efficient Communication Abstraction for Dense Wireless Networks

Magnús M. Halldórsson[*1], Fabian Kuhn[†2], Nancy Lynch[‡3], and Calvin Newport[§4]

1 **Reykjavik University, Iceland**
mmh@ru.is
2 **University of Freiburg, Germany**
kuhn@cs.uni-freiburg.de
3 **MIT, Cambridge, USA**
lynch@csail.mit.edu
4 **Georgetown University, Washington, DC, USA**
cnewport@cs.georgetown.edu

Abstract

In this paper we study the problem of developing efficient distributed algorithms for dense wireless networks. For many problems in this setting, fast solutions must leverage the reality that radio signals fade with distance, which can be exploited to enable concurrent communication among multiple sender/receiver pairs. To simplify the development of these algorithms we describe a new communication abstraction called FadingMAC which exposes the benefits of this concurrent communication, but also hides the details of the underlying low-level radio signal behavior. This approach splits efforts between those who develop useful algorithms that run on the abstraction, and those who implement the abstraction in concrete low-level wireless models, or on real hardware.

After defining FadingMAC, we describe and analyze an efficient implementation of the abstraction in a standard low-level SINR-style network model. We then describe solutions to the following problems that run on the abstraction: max, min, sum, and mean computed over input values; process renaming; consensus and leader election; and optimal packet scheduling. Combining our abstraction implementation with these applications that run on the abstraction, we obtain near-optimal solutions to these problems in our low-level SINR model – significantly advancing the known results for distributed algorithms in this setting. Of equal importance to these concrete bounds, however, is the general idea advanced by this paper: as wireless networks become more dense, both theoreticians and practitioners must explore new communication abstractions that can help tame this density.

1998 ACM Subject Classification C.2.1 Wireless Communication, G.2.2 Graph Theory – network problems

Keywords and phrases wireless networks, abstractions, SINR, signal fading

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.25

[*] MMH is supported by Icelandic Research Fund grants 152679-05 and 174484-05.
[†] FK is supported by ERC grant no. 336495 (ACDC).
[‡] NL is supported by AFOSR contract FA9550-13-1-0042 and NSF grants 0939370-CCF, CCF-1217506, and CCF-AF-1461559.
[§] CN is supported by NSF grant CCF-1649484.

© Magnús Halldórsson, Fabian Kuhn, Nancy Lynch, and Calvin Newport;
licensed under Creative Commons License CC-BY
31st International Symposium on Distributed Computing (DISC 2017).
Editor: Andréa W. Richa; Article No. 25; pp. 25:1–25:16

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

1 Introduction

In this paper we present and study a communication abstraction designed to facilitate efficient computation in dense wireless networks. In addition to defining the abstraction, we describe and analyze an implementation on a low-level SINR-style network model, and provide multiple applications that leverage it to efficiently solve standard network problems.

Problem: Contention Management in Dense Wireless Networks. In wireless networks devices share channels. A common approach to avoiding contention among multiple transmitters on the same channel is to have devices compete to gain exclusive use of the channel. This strategy, for example, is deployed by both the 802.11 and Zigbee standards to prevent message loss due to collision. It is also implicitly enforced in the graph-based models often used by theoreticians to study distributed algorithms for wireless networks (e.g., the radio network model [6, 1] or dual graph model [16, 4]), as these models assume any contention leads to message loss at the receiver.

This *exclusive-use* strategy is effective for many existing networking scenarios. It is not well suited, however, for *dense wireless networks* in which hundreds, thousands, or even tens of thousands of wireless devices might be located in a small area. Dense networks have become increasingly relevant due to emerging trends such as the Internet-of-Things, in which networking capability is embedded into a wide variety of objects. The density of these networks render exclusive-use contention management strategies too inefficient for many problems, as the time required for each device to use the channel in isolation is prohibitive.

Solution: Abstractions that Leverage Spatial Reuse. One way to enable efficient computation in dense wireless networks is to take advantage of the reality that radio signals fade with distance. This property makes it possible to support a large amount of concurrent communication among closely packed devices – a strategy often called *spatial reuse*.[1] Leveraging spatial reuse can allow network applications to solve certain problems much faster than relying on an exclusive-use strategy (c.f., the summary of our results below).

Developing algorithms that leverage spatial reuse requires low-level wireless network models that directly model radio signal fading and determine receive behavior using a *signal to interference and noise ratio* (SINR) equation. These models have received extensive attention from specialists in the algorithms community (e.g., [7, 14, 17, 13, 12, 11, 3, 19, 20, 18]), but they are also complex, and arguably demand too high a barrier of entry for those interested in dense networks, but not necessarily willing to master the intricacies of affectance formulas and interference-bounded annuli. These difficulties impede efforts to develop useful and efficient applications that are well suited for dense networks.

We argue that a solution to these problem is to develop *communication abstractions* that present the network application developer with an intuitive interface that exposes the concurrent communication benefits of spatial reuse, while hiding the low-level details of the underlying radio signal behavior. This approach splits efforts between those developing network applications that run on the abstraction, and those who are implementing the abstraction in low-level models, and, eventually, on real hardware. By doing so it simplifies

[1] The basic mechanism behind spatial reuse is the following: If a pair of wireless devices u and v are close together compared to other nearby transmitters in the network, then the strength of a radio signal sent from u and received at v might be strong enough compared to interference from other slightly more distant transmitters to allow v to successfully receive u's message.

the study of efficient distributed algorithms in SINR models, and provides feedback to practitioners about how best to innovate network stacks to accommodate the continued growth of wireless device density.

The FadingMAC Abstraction. In this paper we propose one such communication abstraction, which we call FadingMAC. This abstraction organizes the wireless devices (called *processes* in the following) into a labelled tree structure. It then moves through the levels of the resulting tree: at each level, processes assigned to tree nodes at that level can use the abstraction to communicate with the processes assigned to their parent and/or child nodes in the tree. This abstraction allows processes to aggregate and disseminate information in time proportional to the tree height. Therefore, if the tree has a small height compared to the network size, it enables certain computations to complete faster than what is possible with exclusive-use strategies.

Our Results. We begin by describing and analyzing an implementation of the FadingMAC abstraction using a standard low-level SINR wireless network model. In the following, let n be the number of processes in the system and R be the ratio between the longest and shortest link in the low-level model (in nearly all realistic scenarios R is at most polynomial in n). Our FadingMAC implementation guarantees the following with high probability in n: its setup mode requires $O(\log R \cdot \log n \cdot \log^* n)$ rounds; the height of the resulting tree is in $O(\log R)$; and it requires only $O(1)$ rounds to handle communication at each tree level.

We then turn our attention to applications that run on the FadingMAC abstraction. We begin by adapting a classical parallel prefix scan strategy to our setting, and then use this scan as a key subroutine to implement solutions to the following problems, selected due to their relevance to dense wireless networking scenarios:

- Max, min, sum, and mean calculated over input values;
- Renaming processes with unique labels from 1 to n (or renaming any subset of k processes with unique labels from 1 to k);
- Leader election and consensus among arbitrary subsets of participating processes; and
- Optimal packet scheduling (each process has a request for a certain number of rounds it needs to send information, and the goal is to agree on a schedule that assigns each process exactly their requested number of rounds).

Our solutions require visits to at most $O(h)$ tree levels by the abstraction, where h is the tree height. Combined with our abstraction implementation, and the assumption that R is polynomial in n, we obtain new distributed algorithms for these problems in a low-level SINR model that require a setup cost of $O(\log^2 n \cdot \log^* n)$ rounds and only $O(\log n)$ rounds per instance. Given that $\Omega(\log n)$ rounds is a fundamental speed limit for solving non-trivial problems with high probability in the SINR setting (see the discussion of [8] in Section 2), these solutions are near optimal, and exponentially faster than the $\Omega(n)$ rounds required for solving these types of problems with exclusive-use strategies.

To summarize, our results provide three strong contributions to the study of dense wireless networks. First, our applications combined with our abstraction implementation provide new (and nearly optimal) solutions to multiple useful distributed algorithm problems in the SINR model. Second, our abstraction simplifies the future development of efficient distributed algorithms for the complicated SINR setting. Third, we validate the general argument that in both theory and practice, the right abstractions can help tame the increasingly important setting of dense networks.

2 Related Work

In a 2006 paper [19], Moscibroda and Wattenhofer described and analyzed a centralized algorithm that can schedule a connected structure of links in an SINR model in $O(\log^4 n)$ rounds. They note that in many standard models (such as the exclusive-use models cited above as motivation) this problem requires $\Omega(n)$ rounds – underscoring the importance of spatial reuse for efficient computation in dense networks. A subsequent series of papers [20, 18, 12] presented refined centralized scheduling algorithms, culminating in a $O(\log n)$-round solution due to Halldórsson and Mitra [12]. These same authors subsequently published a distributed version of their algorithm [11]. By comparison, our FadingMAC implementation solves this same scheduling problem in $O(\log R \cdot \log n \cdot \log^* n)$ rounds. These results, however, are not directly comparable as [20, 18, 12, 11] all assume that devices can adjust their transmission power, whereas we assume all devices use a fixed uniform power.

More recent results [14, 17] study efficient data aggregation in the SINR model with distributed algorithms and fixed uniform power. Hobbs et al. [14] describe a deterministic algorithm that aggregates data in a multihop network[2] in $O(D + \Delta \cdot \log n)$ rounds, where D is the diameter of the connectivity graph and Δ is its maximum degree (see the below footnote about connectivity graphs). Notice, in our single hop setting $D = n$. In addition, this algorithm, unlike ours, requires carrier sensing. Perhaps closer is the work of Li et al. [17], which shows how to aggregate data in a single hop setting in $O(\log R)$ rounds. Their solution, however, requires that devices are provided their position and neighbor count in advance – enabling efficient strategies built on fixed geographic regions. Our FadingMAC solution, by contrast, does not assume this information is known.

Fineman et al. [8] prove a $\Omega(\log n)$ lower bound on symmetry breaking with high probability in the same SINR model we study. A closer inspection of this bound reveals it holds for basic partition detection, and therefore applies to the applications we study in Section 7 – establishing their time complexities as optimal or near optimal (depending on assumptions about R and whether you include the one-time abstraction setup cost).

Our implementation of the FadingMAC abstraction adapts the reliable subgraph simulation strategy introduced to solve distributed one-to-all broadcast in [7], while our applications leverage a distributed prefix scan subroutine that adapts strategies from classical parallel solutions to this problem (see [2] and our discussion in Section 7). We previously explored the idea of mediating between low-level wireless models and high-level applications with our earlier work on the Abstract MAC Layer [15], including an implementation in the SINR model [10]. The Abstract MAC Layer is intended to abstract away the details of contention management (primarily) in graph-based models.

3 System Model

We assume a synchronous network with $n > 1$ computational processes. Let V be the set of processes. We assume each process is distinguished by a comparable and unique ID. Time proceeds in synchronous rounds that we label $1, 2, 3, ...$, and all processes start in round 1.

We assume the processes are connected by some underlying wireless network over which they can attempt to broadcast and receive messages. We assume this network is *single hop*, which we define in a general way here to mean that if a process $i \in V$ transmits alone in a given round, all nodes receive i's message during this round. We assume message sizes are bounded to contain at most $O(\text{polylog}(n))$ unique IDs and/or bits.

[2] In the SINR setting, *single hop* and *multihop* refer to the *connectivity graph* that results from connecting with an edge any process pair that are sufficiently close to communicate in the absence of interference.

In this paper, we present and study a *communication abstraction* called FadingMAC (defined below), that mediates between high-level network algorithms and a low-level wireless network model. Accordingly, we assume each computational process implements two distinct components:

(1) a FadingMAC abstraction implementation that interacts with other processes through the low-level wireless network model; and

(2) a high-level algorithm that interacts with other processes only through the interface provided by its local copy of the abstraction.

We sometimes say the high-level algorithm *runs on* the abstraction.

4 The FadingMAC Abstraction

The FadingMAC abstraction organizes the processes into a rooted tree structure. Each process is potentially assigned to multiple nodes in the tree. The abstraction is used in phases, with each phase dedicated to a level in the tree. A process assigned to a node u in the level corresponding to the current phase can use a children-to-parent communication primitive, provided by the abstraction, to communicate reliably with the process assigned to u's parent (assuming u is not the root). This process can also use a parent-to-children communication primitive to reliably communicate with the processes assigned to u's children (if any).

We begin below by formalizing the type of tree the abstraction uses to organize the nodes. We then detail the communication primitives provided by the abstraction, and present the parameters that describe a particular implementation's performance and correctness guarantees.

The Tournament Tree. The abstraction begins an execution in a *setup mode*, during which it organizes the processes into a *tournament tree* defined with respect to V (see below). Formally, this means that by the end of the setup mode, the abstraction outputs to each process information about its assigned nodes in the tree. When considered collectively, this output information defines a unique tree T.

In more detail, a tournament tree defined over V is a rooted tree that satisfies the following properties:

1. All leaf nodes are at the same level in the tree.
2. Each node in the tree is labeled with a process from V.[3]
3. The n leaves are labeled with the n unique processes from V.
4. Each non-leaf node in the tree is labeled with the label of one of its children.

We call a labeled tree of the type a *tournament tree* because the label of each non-leaf node can be interpreted as the winner of a competition among the processes labeling its children. Fix a tournament tree T. As is standard for rooted trees, we label the tree's levels starting with 0 for the root level and increasing the level number as we move downward toward the leaves. We define $h(T)$ to describe its height (i.e., the largest level in the tree). For a given level $\ell \in [0, h(T)]$, we define $V(T, \ell)$ to be the set of processes labeling nodes at ℓ.

[3] Technically, the node is labelled with the unique ID corresponding to process i, but for notational simplicity we use i and i's ID interchangeably.

During the setup period, the FadingMAC abstraction will identify a single tournament tree T defined with respect to V and output to each process $i \in V$ the following three pieces of information regarding the nodes it labels in T:

(1) the smallest level in T that includes i, which we denote $\ell(T, i) = \min\{\ell \in [0, h(T)] : i \in V(T, \ell)\}$;

(2) if $\ell(T, i) > 0$, i's parent in level $\ell(T, i) - 1$ (otherwise it receives some default value $\perp$);

(3) the height $h(T)$ of the tree.

Notice, the collective knowledge provided to the processes about T by the abstraction uniquely specifies T.

The Communication Primitives. After the setup period finishes producing the tree T, the abstraction informs all processes that it is shifting into *communication mode*. While in communication mode, the abstraction divides rounds into *phases*. Each phase is dedicated to a single level from the tree T organized during the setup period. The abstraction notifies all processes when each phase begins as well as the phase's corresponding level in T. Different applications using FadingMAC can use different orderings of the phases for visiting levels in the tree. (The applications we study later in this paper, for example, follow a pattern of sweeping from the leaves to the root and then back down again.)

During a phase dedicated to tree level ℓ, every process in $V(T, \ell)$ is provided the opportunity to communicate with processes assigned to nearby nodes in T using a pair of communication primitives provided by FadingMAC.

In more detail, the abstraction provides both a *parent-to-children* and *children-to-parent* broadcast primitive. During the first round of a phase dedicated to level ℓ, each process $i \in V(T, \ell)$ can provide as input to the abstraction a message m_1 for the *parent-to-children* primitive and a message m_2 for the *children-to-parent* primitive. By the definition of a tournament tree, if $i \in V(T, \ell)$, then this process is assigned to exactly one node u at this level of T. During the subsequent rounds of the phase the abstraction attempts to deliver m_1 to the process assigned to u's parent in T (if u is the root or process i is assigned to u's parent as well, then it will receive its own message back.) It also attempts to deliver m_2 to the processes assigned to u's children in T (if any).

Abstraction Parameters. The guarantees of a given implementation of the FadingMAC abstraction is characterized by the following parameters:

- t_{setup}: the number of rounds required to complete the setup period.
- t_{phase}: the number of rounds required for each phase during the communication mode.
- h_{max}: the maximum height of the tournament tree.

To accommodate probabilistic implementations of the abstraction, we include a failure probability, p_{fail}, describing the maximum probability that the abstraction fails to satisfy its specified behavior.

5 The Low-Level SINR Wireless Model

Here we define a low-level wireless model that captures the fading behavior of radio signals. We call this "the SINR" model, but note that there are many related variations on this same style of model (the variant defined below is a generalized version of the model defined in [7]).

The SINR model is motivated by the premise that transmissions are successfully received when the signal arriving at a given receiver is sufficiently strong compared to the sum of interference from other transmissions. The strength of the transmissions is modeled as

inversely proportional to a polynomial of the distance transmitted. The idea that signals fade with distance in this manner and interference sums at the receiver match experimental investigations (c.f., [21, 5]). This is the primary reason why the SINR model is viewed as one of the most realistic analytic models currently available for studying low-level wireless algorithms. We note, however, that earlier studies of the SINR model tended to place the devices in the plane and define "distance" to be Euclidean. This assumption weakens the realism of the model as real environments can distort and attenuate signals in complex ways. To accommodate this reality, we replace Euclidean distances in our below model definition with a more general metric that captures more diverse signal environments.

We now formalize the model definition. In doing so, we mostly follow notation from [7]. We assume that all processes transmit with the same power, which for simplicity we normalize as 1. The strength of a transmission from process u as received by a process v is $1/d(u,v)^\alpha$, where $d(u,v)$ is the distance from u to v and α is a constant. The transmission is *successful* if v is listening and the SINR formula holds:

$$SINR(u,v,I) := \frac{1/d(u,v)^\alpha}{N + \sum_{w \in I} 1/d(w,v)^\alpha} \geq \beta \ ,$$

where I is the set of other concurrent transmitters, β is a positive constant dependent on the hardware and coding schemes used, and N quantifies the ambient noise.

We assume that the distance metric satisfies the following doubling property: There are constants λ and d, with $\lambda > 0$ and $0 < d < \alpha$, such that if U is a set of processes of mutual distance at least d_{min}, and all processes in U are within a distance $x \cdot d_{min}$ from a given process, then $|U| \leq \lambda \cdot x^d$. For the Euclidean plane, this holds with $d = 2$. We use this relaxed definition to account for the various effects of signal propagation.

Let r_s denote the maximum distance at which successful transmission is possible, i.e., $1/r_s^\alpha = \beta N$, or $r_s = (\beta N)^{-1/\alpha}$. Let R denote the ratio between the largest and smallest process distance in the network.

We assume that processes have no collision detection or carrier sensing capability, and no advance knowledge of their distances to other processes. The processes know a polynomial upper bound on the standard deployment parameters: the network size $n = |V|$ and the distance diversity R. We also assume they know reasonable approximations of the network parameters α, β, n, and N.

6 Implementing FadingMAC in the SINR Model

We now describe and analyze a randomized distributed algorithm that implements the FadingMAC abstraction defined in Section 3 in the SINR model defined in Section 5. Our optimized version of this algorithm, presented in Section 6.3, implements the abstraction with $t_{setup} = O(\log R \cdot \log n \cdot \log^* n)$, $t_{phase} = O(1)$, $h_{max} = O(\log R)$, and $p_{fail} < n^{-c}$, for a constant $c > 1$.

We first define in Sec. 6.1 a family of graphs that captures succinctly when nodes can communicate effectively, and use it in Sec. 6.2 to construct a basic communication tree. In Sec. 6.3, we improve it so that each phase can be executed in constant number of rounds.

6.1 SINR-Induced Graphs

Our construction for each given level ℓ is based on constructing a bounded-degree graph H_ℓ. To distinguish, we shall refer to the nodes of the graph as *vertices*, reserving *nodes* for nodes of the tournament tree, and noting that computation on behalf of both types of objects is

executed by *processes*. To construct H_ℓ, we apply the concept of *SINR-induced graphs* as defined by Daum et al. in [7]. Let $U \subseteq V$ be a subset of the vertices, let p be a transmission probability, and let $\mu \in (0, p) \cap \Omega(1)$ be a reliability parameter. We define the SINR-induced graph $H_p^\mu[U]$ with vertex set U and edge set $E_p^\mu[U]$ as follows. Assume that each vertex in U transmits with probability p and each vertex in $V \setminus U$ transmits with probability 0. For $u, v \in U$, we have $\{u, v\} \in E_p^\mu[U]$ if and only if u receives a message from v with probability at least μ and also v receives a message from u with probability at least μ. Note that all distances between vertices/nodes are within the metric space, not in terms of path lengths in the graph/tree.

Distributed Computation of SINR-Induced Graphs. Given a set of vertices $U \subseteq V$ and parameters p and μ, the graph $H_p^\mu[U]$ cannot be computed exactly in a distributed way. However, it is possible to efficiently compute a close approximation of $H_p^\mu[U]$. In [7], an *ε-close approximation* of $H_p^\mu[U]$ is defined as a graph $\tilde{H}_p^\mu[U] = (U, \tilde{E}_p^\mu[U])$ for which:

$$E_p^\mu[U] \subseteq \tilde{E}_p^\mu[U] \subseteq E_p^{(1-\varepsilon)\mu}[U]$$

It is shown in [7] that an ε-close approximation $\tilde{H}_p^\mu[U]$ of $H_p^\mu[U]$ can be constructed as follows in $2T$ rounds, where $T = c\frac{\ln n}{\varepsilon^2 \mu}$ for a sufficiently large constant $c > 0$. In all $2T$ rounds, all vertices in U transmit with probability p. In the first T rounds, each vertex $u \in U$ compiles a list of all vertices of which u receives a message in at least $\left(1 - \frac{\varepsilon}{2}\right)\mu T$ of the T rounds. In the second T rounds, those lists are exchanged and an edge $\{u, v\}$ is added to $\tilde{E}_p^\mu[U]$ if and only if u is in v's list and v is in u's list. It is shown in Lemma 3 in [7] that this algorithm computes an ε-close approximation of $H_p^\mu[U]$, w.h.p. We note that because in each round, a vertex can only receive a message from a single vertex, the maximum degree of $\tilde{H}_p^\mu[U]$ is bounded by $1/\mu = O(1)$.

We can also show that $H_p^\mu[U]$ contains all relatively short edges. We first need the following lemma, which is actually somewhat surprising, even though its proof (in the appendix) deploys relatively standard techniques.

▶ **Lemma 1.** *Given a constant $\zeta \geq 1$, there exists a constant $\nu = \nu_\zeta > 0$ such that the following holds. If $U \subseteq V$ is a subset of vertices of minimum pairwise distance d_{min} and $u, v \in U$ are vertices of distance at most $d(u, v) \leq \zeta \cdot d_{\min}$, then whenever v transmits while other vertices within distance $\nu \cdot d_{\min}$ of u are silent, u is guaranteed to successfully receive the transmission of v.*

We now extend Lemma 4 in [7], to show that $H_p^\mu[U]$ contains all short edges.

▶ **Lemma 2.** *For all $p \in (0, 1/2]$ and every constant $\eta > 1$, there exists $\mu \in (0, p)$ such that $\forall U \subseteq V$ with minimum pairwise distance d_{min}, the graph $H_p^\mu[U]$ (and thus also the graph $\tilde{H}_p^\mu[U]$) contains all edges between vertices $u, v \in U$ with $d(u, v) \leq \eta \cdot d_{\min}$.*

Proof. Let ν be the constant for which Lemma 1 holds for $\zeta = \eta$. Let $u, v \in U$ be any pair of vertices with $d(u, v) \leq \eta \cdot d_{min}$. Let S be the set of vertices in V within distance $\nu \cdot d_{\min}$ of u, excluding v. By the doubling property of the metric, $|S| \leq \lambda \cdot \nu^d$. By Lemma 1, u receives the message from v if v transmits while the vertices in S are silent. The probability of the former is p while the probability of the latter is $(1-p)^{|S|}$. Since these events are independent, it follows that u successfully receives a message from v with probability $p(1-p)^{|S|} \geq p(1-p)^{\lambda \cdot \nu^d}$. By a symmetric argument, the same holds for the probability that v successfully receives a message from u. Hence, for $\mu \geq p^2(1-p)^{2\lambda \cdot \nu^d}$, $\{u, v\}$ satisfies the condition of being an edge in H_p^μ. ◀

6.2 Construction and Properties of the Basic Tournament Tree

We now first describe the construction of a tournament tree that supports basic communication primitives. In the next subsection, we extend this construction and show how to get a tree and a constant-length TDMA schedule for implementing the communication primitives on each level. For the construction of the tournament tree T, we fix a transmission probability p and a (constant) parameter $\mu > 0$ such that the conditions of Lemma 2 are satisfied for $\eta = 2$ and the graph H_p^μ contains all edges of length at most $2d_{\min}$ for every $U \subseteq V$.[4] Further, for the distributed construction of an ε-close approximation $\tilde{H}_p^\mu[U]$ of $H_p^\mu[U]$ for a vertex set U, we fix $\varepsilon := 1/10$ throughout the construction. We construct the tournament tree T bottom-up and level by level. That is, we start with level $\ell = h(T)$ connecting the leaf nodes to their parents and then move up the tree until we reach level 1 where we select a root node $r \in V(T,1)$ and connect all nodes $V(T,1)$ to the root node r. The construction for level ℓ returns the set of nodes $V(T,\ell-1)$ that proceed to the next level and it fixes the parent nodes in level $\ell-1$ for all nodes in $V(T,\ell) \setminus V(T,\ell-1)$.

In the following, we describe the construction for a given level $\ell \in \{1,\dots,h(T)\}$ (recall that $V(T,h(T)) = V$):

1. Compute the graph $H_\ell := \tilde{H}_p^\mu[V(T,\ell)]$.
2. Compute a maximal independent set (MIS) I of H_ℓ and set $V(T,\ell-1) := I$
3. Each node $u \in V(T,\ell)\setminus V(T,\ell-1)$ chooses its parent v arbitrarily among its H_ℓ-neighbors in $V(T,\ell-1)$.

The following theorem summarizes the basic properties of the resulting tournament tree T.

▶ **Theorem 3.** *With high probability, the constructed tournament tree T has the following properties:*

- *The height of the tree is $h(T) = O(\log R)$.*
- *Suppose all nodes in $V(T,\ell)$ transmit with probability p and the other nodes are silent in a given round r and level $\ell \in \{1,\dots,h(T)\}$. Then, for every node $u \in V(T,\ell)$, u receives a message from its parent in round r with constant probability, and the parent also receives a message from u in round r with constant probability.*
- *The tree T can be constructed in time $O(\log R \cdot \log n \cdot \log^* n)$ in the SINR model.*

Proof. We first prove that the height of the constructed tree T is $O(\log R)$, w.h.p. Let $d_{\min}^{(\ell)}$ be the minimum distance between vertices in $V(T,\ell)$. By Lemma 2 and the choice of the parameters p and μ, the graph H_ℓ contains an edge for every two vertices $u,v \in V(T,\ell)$ at distance at most $\min\left\{2d_{\min}^{(\ell)}, r_s\right\}$. Because $V(T,\ell-1)$ is chosen as an MIS of the graph H_ℓ, no two verticess in $V(T,\ell-1)$ can be neighbors in H_ℓ and we thus get that $d_{\min}^{(\ell-1)} > \min\left\{2d_{\min}^{(\ell)}, r_s\right\}$. Hence as long as $d_{\min}^{(\ell)} \le r_s/2$, we have $d_{\min}^{(\ell-1)} > 2d_{\min}^{(\ell)}$. As soon as $d_{\min}^{(\ell)} > r_s/2$, by Lemma 2, H_ℓ is a complete graph and thus we reached the last level ($\ell = 1$). The claim that $h(T) = O(\log R)$ now follows because from the definition of R, the minimum distance between vertices in V is at least r_s/R.

The second property follows directly from the definition of the edges of the graph H_p^μ and the fact that parents are neighbors in H_p^μ. In turn, it implies that every vertex in $V(T,\ell)$ successfully receives a message from each of its neighbors in H_ℓ within $O(\log n)$ rounds, w.h.p. Hence, a single communication round in a standard message passing model on H_ℓ can be simulated in $O(\log n)$ rounds in the SINR model. Because H_ℓ has bounded degree, an MIS of H_ℓ can be computed in $O(\log^* n)$ rounds in a standard message passing model on H_ℓ [9] and the also the third claim of the lemma follows. ◀

[4] We can for example fix $p = 1/2$ and choose $\mu > 0$ as a sufficiently small constant.

6.3 A Tournament Tree with an Efficient TDMA Schedule

We next show how to adapt and extend the construction of the previous subsection to obtain a tournament tree with the same properties as in Theorem 3, but now with the additional assumption of a constant-length TDMA schedule for implementing the communication primitives on each level.

As in Section 6.2, the tree is computed in a bottom-up fashion and we describe the construction of a single level ℓ in detail. We first give an outline of the construction. We fix the parameters p, μ, and ε as before and we again construct the graph $H_\ell = \tilde{H}^{\mu}_{p}[V(T,\ell)]$. As a next step, we compute a TDMA schedule of length $L = O(1)$ for communicating on graph H_ℓ. That is, we assign a color $\phi_\ell(v) \in \{1,\dots,L\}$ to each vertex $v \in V(T,\ell)$. We then define an L-round schedule $\mathcal{S}_\ell$ for level ℓ such that exactly the nodes $v \in V(T,\ell)$ with $\phi_\ell(v) = i$ transmit in round i. We say that an edge $\{u,v\} \in E[H_\ell]$ is successful w.r.t. schedule $\mathcal{S}_\ell$ if in round $\phi_\ell(u)$ of the schedule $\mathcal{S}_\ell$, v receives a message from u and in round $\phi_\ell(v)$ of $\mathcal{S}_\ell$, u receives a message from v. We define H'_ℓ as the subgraph of H_ℓ consisting of all edges that are successful w.r.t. $\mathcal{S}_\ell$. The level ℓ of the tree T is then constructed in the same way as before, but by using graph H'_ℓ instead of graph H_ℓ.

We will next show that the coloring ϕ_ℓ can be constructed such that the resulting subgraph H'_ℓ of H_ℓ still contains all edges of length $\min\left\{2d^{(\ell)}_{\min}, r_s\right\}$ and as a consequence, the constructed tree still satisfies the properties of Theorem 3.

In order to construct the coloring ϕ_ℓ, we consider an SINR-induced graph $H^{\mu'}_{p'}[V(T,\ell)]$ such that $H^{\mu'}_{p'}[V(T,\ell)]$ contains all edges between vertices in $V(T,\ell)$ of distance at most $\eta \cdot d^{(\ell)}_{\min}$, for a sufficiently large constant $\eta > 2$. Note that by Lemma 2, we can choose constants p' and μ' such that the graph $H^{\mu'}_{p'}[V(T,\ell)]$ satisfies this. We choose η as the value of ν obtained from applying Lemma 1 with $\zeta = 2$. The coloring ϕ_ℓ will be computed as a standard vertex coloring of an ε-close approximation of the SINR-induced graph $H^{\mu'}_{p'}[V(T,\ell)]$.

As in the basic construction, we construct the tree T in a bottom-up way, starting at level $\ell = h(T)$ where the leaf nodes are connected to their parents and ending at level 1 where a root node $r \in V(T,1)$ selected. We again describe the construction for a given level $\ell \in \{1,\dots,h(T)\}$. In the following description, $\Delta(H^+_\ell)$ denotes the maximum degree of graph H^+_ℓ.

1. Compute the graph $H^+_\ell := \tilde{H}^{\mu'}_{p'}[V(T,\ell)]$.
2. Compute a $(\Delta(H^+_\ell)+1)$-vertex coloring ϕ_ℓ of the graph H^+_ℓ
3. Compute the graph $H_\ell := \tilde{H}^{\mu}_{p}[V(T,\ell)]$.
4. Compute the graph H'_ℓ consisting of all the edges of H_ℓ that are successful w.r.t. the TDMA schedule induced by the coloring ϕ_ℓ.
5. Compute a maximal independent set (MIS) I of H'_ℓ and set $V(T,\ell-1) := I$
6. Each node $u \in V(T,\ell)\setminus V(T,\ell-1)$ chooses its parent v arbitrarily among the H'_ℓ-neighbors in $V(T,\ell-1)$.

Analogously to Theorem 3, the following theorem summarizes the properties of the constructed extended tournament tree T. The proof is deferred to the appendix.

▶ **Theorem 4.** *With high probability, the constructed tournament tree T has the following properties:*

- *The height of the tree is $h(T) = O(\log R)$.*
- *For every $\ell \in \{1,\dots,h(T)\}$, when using the TDMA schedule induced by the coloring ϕ_ℓ of $V(T,\ell)$, (bidirectional) communication between every node $u \in V(T,\ell)$ and its parent in $V(T,\ell-1)$ can be carried out in $O(1)$ rounds.*
- *The tree T can be constructed in time $O(\log R \cdot \log n \cdot \log^* n)$ in the SINR model.*

7 High-Level Algorithms

Here we describe and analyze several useful high-level algorithms that run on the FadingMAC abstraction. The key subroutine used for these applications is a *distributed prefix scan* over the processes' input values. We begin below by describing and analyzing an algorithm that computes such a scan using a small number of FadingMAC phases, before moving on to describe the algorithms that leverage these scans.

7.1 Distributed Prefix Scans Using FadingMAC

During its setup mode, the FadingMAC abstraction returns each process information about the nodes it labels in a tournament tree T. We can transform T into an ordered tree by using a comparison operator defined over process labels to order the siblings of each parent from left to right (i.e., the operator that compares the process unique ids). This ordered version of T then allows us to order the n processes in the network from 1 to n by considering the leaf labels from left to right.

In the following, in a slight abuse of notation, we use *process* i, for $i \in [1, n]$, to indicate the process ranked in position i in this ordering. Assume each process i has an input value a_i from some value domain S that includes an identity element ϵ. Let $\oplus$ be a binary associative operator defined over S. (This combination of S and $\oplus$ defines a monoid.) The goal of a *distributed prefix scan* for a given monoid and input value assignment in our model is for each process i to learn: $s_i = a_1 \oplus a_2 \oplus ... \oplus a_i$.

We emphasize that prefix scans (also called prefix sums) are a core primitive in algorithm design, as they provide a building block for efficiently solving a variety of different problems. They are therefore studied in multiple models. In an exclusive-use radio network model it is straightforward to see that $\Omega(n)$ rounds are required (as each process needs its own round to communicate its value). Below we describe and analyze an algorithm that leverages the FadingMAC abstraction to calculate a prefix scan in $2h(T)$ communication mode phases. We emphasize that for the implementation of FadingMAC provided in this paper, this translates to $O(\log n)$ rounds for most networks.[5]

Strategy and Comparison to Parallel Prefix Scans. Our distributed strategy adapts general ideas from the classical parallel prefix scan algorithms; c.f., [2]. Similar to our setup, these parallel algorithms tend to also consider an ordered tree with one input value assigned to each of the n leaves. They move up the tree level by level, starting from the leaves and heading toward the root, at each node u summing the values of u's children. The result is that each node u learns the sum of the values in the leaves of the subtree rooted at u. The algorithm then moves down the tree level by level, with each internal node u informing each child the sum of its siblings to its left in the tree ordering.

Our distributed prefix scan algorithm implements this same basic strategy. To do so, however, it must overcome two difficulties that separate our model from parallel models:

(1) in the parallel setting the tree used is fixed by the algorithm and known to all processes, while in our model the tree is returned by the abstraction and each process is only provided partial information about the tree's structure;

[5] Our implementation requires only a constant number of rounds per phase and provides $h(T) = O(\log R)$. For most realistic networks, R will be at most polynomial in n.

(2) in the parallel setting a different process can be assigned to each node in the tree, while in our model each of the n processes might be responsible for multiple tree nodes on the path from its leaf to the root, complicating the algorithm description.

Algorithm Preliminaries. The algorithm described below uses the FadingMAC abstraction parameterized to visit the tree levels in order starting from the leaves and heading up to the root (called the *up sweep* in the following), and then visiting the tree levels in order starting from the root and heading back down to the leaves (called the *down sweep* in the following). Only a single up sweep followed by a single down sweep are required for our algorithm to compute it prefix scan.

Fix some node u in the ordered tournament tree. Let u.left indicate the rank of the leftmost leaf in this interval, and r.right indicate the rank of the rightmost leaf. By definition, r.left $\leq r$.right, but it is possible that u.left $= r$.right. We define the *sum of the subtree rooted at* u, denoted $s(u)$, to be the sum (defined with respect to the $\oplus$ operator) of the input values associated with the leaves of u's subtree. Formally: $s(u) = a_{u.\text{left}} \oplus a_{u.\text{left}+1} \oplus ... \oplus a_{u.\text{right}}$.

Algorithm Description. We now describe our algorithm which computes a distributed prefix scan using the FadingMAC abstraction. We first describe its behavior during the up sweep and then during the down sweep. The following assumes we have already entered the abstraction's communication mode.

Up Sweep. During the up sweep phases, the goal is to calculate the sum of the subtree rooted at each node in the ordered tournament tree T. That is, for each node u in T, our algorithm will calculate and store $s(u)$. Recall, in a tournament tree, each node is labelled with a process. In our algorithm, it will be the responsibility of the process that labels u to calculate and store $s(u)$. This process will also end up storing a copy of $s(u')$ for each child u' of u in T, as this will be needed during the subsequent down sweep.

To accomplish this goal, each process i maintains an accumulation variable $x(i)$ initialized to the identity element ϵ at the beginning of the upsweep. It will use this variable to help calculate the sum values for which it is responsible.

In more detail, each process i has two types of responsibilities for each level ℓ visited during the upsweep. The first type of responsibilities applies if $\ell > 0$ and there exists a node u at level $\ell - 1$ that is labelled with process i. If these conditions are met, then process i will calculate $s(u)$ by the end of this phase. To do so, process i first waits to see if node u receives any values from its children at level ℓ during this phase, delivered through the children-to-parent broadcast primitive provided by the abstraction. Assume this occurs and that we label the received values $q_1, q_2, ..., q_j$. In this case, process i updates $x(i)$ as follows: $x(i) \leftarrow x(i) \oplus q_1 \oplus q_2 \oplus ... \oplus q_j$. Process i will also save these received values which, as will soon be made clear, describe the sum of the subtrees rooted at these children – knowledge process i will need during the down sweep.

Regardless of whether or not u received any values, at the end of this phase, process i locally calculates and stores $s(u)$ as follows: $s(u) \leftarrow a_i \oplus x(i)$.

The second type of responsibility applies if there exists a node v at the current level ℓ that is labelled with process i. If v is a leaf (i.e., $\ell = h(T)$), then process i can directly set $s(v) = a_i$. If v is not a leaf, then process i would have calculated $s(v)$ during the previous phase for level $\ell + 1$ as described above. Therefore, in both cases, $s(v)$ is defined. If v's parent is *not* labelled with process i, then process i will use the children-to-parent broadcast primitive to send $s(v)$ to (the process labelling) u's parent.

Down Sweep. We now describe the behavior of our algorithm during the down sweep. The goal during the down sweep is for the process labelling each node u in T to learn:

$$pred(u) = \begin{cases} a_1 \oplus a_2 \oplus ... \oplus a_{u.\text{left}-1} & \text{if } u.\text{left} > 1, \\ \epsilon & \text{else.} \end{cases}$$

That is, $pred(u)$ is the sum of the values (defined with respect to $\oplus$) associated with all the leaves (if any) to the left of the subtree rooted at u in T.

We accomplish this goal as follows. Before the down sweep begins, the root u_0 of the tree can directly set $pred(u_0) \leftarrow \epsilon$. Now assume inductively that we arrive at the phase associated with level ℓ, $0 \leq \ell < h(T)$, and $pred$ is already calculated for all nodes at level ℓ. During this phase, (the processes labeling) these nodes will send (the processes labelling) their children at level $\ell + 1$ their $pred$ values.

In more detail, fix some u at level ℓ. Assume u is labelled with process i. It is the responsibility of process i to calculate the $pred$ value for each of u's children and to then send it to the processes labelling them using the parent-to-children broadcast primitive. To do so, let $u_1, u_2, ..., u_j$ be u's $j \geq 1$ children nodes, ordered by the same comparison operator we used to order T. By the postcondition of the up sweep, process i knows $s(u_i)$ for each child u_i of u. Process i can therefore calculate $pred(u_i)$, for each child u_i, as follows:

$$pred(u_i) \leftarrow \begin{cases} pred(u) \oplus s(u_1) \oplus s(u_2) \oplus ... \oplus s(u_{i-1}) & \text{if } i > 1, \\ pred(u) & \text{else.} \end{cases}$$

Final Calculation of Scan Values. Each process i labels a single leaf node u in tree T. By the postcondition of the down sweep described above, process i knows $pred(u)$ by the end of the down sweep. Because $pred(u)$ is the sum of all values in s_i except a_i, process i can conclude the algorithm by calculating $s_i \leftarrow pred(u) \oplus a_i$.

Analysis. The correctness of the above distributed prefix scan algorithm follows from a straightforward inductive argument that establishes that during the up sweep each $s(u)$ is correctly calculated. If these subtree sum values are correct then the correctness of the $pred$ values calculated during the subsequent down sweep follows directly from the operation of the algorithm. We formalize this correctness with the following theorem, where p_{fail} describes the maximum probability that abstraction fails to satisfy its t_{phase} and h_{max} guarantees.

▶ **Theorem 5.** *This algorithm computes a distributed prefix scan using the FadingMAC abstraction in $O(t_{phase} \cdot h_{max})$ rounds after the abstraction has entered the communication mode, with probability at least $1 - p_{fail}$.*

7.2 Applications

We now describe multiple applications that leverage the distributed prefix scan algorithm from the previous section as a subroutine to efficiently solve useful numerical and distributed coordination problems. Every application below requires at most a constant number of distributed prefix scans, which each require at most $O(t_{phase} \cdot h_{max})$ rounds. When combined with our FadingMAC implementation in the SINR model, we obtain solutions to the below problems that run with high probability in the SINR model in at most $O(\log R)$ rounds, in addition to the one-time time cost of $O(\log R \cdot \log n \cdot \log^* n)$ rounds for the abstraction setup mode. As argued above, R is at most polynomial in n for most realistic networks, meaning these SINR solutions require only $O(\log n)$ rounds per instance, and a one-time setup cost of $O(\log^2 n \cdot \log^* n)$ rounds.

Renaming and Network Sizing. Our system model assumes that the n processes are only provided comparable unique IDs. In many applications it might useful if the processes were assigned unique IDs from the set $\{1, 2, ..., n\}$. This can be easily accomplished with a single distributed prefix scan. In more detail, set the domain S to be the natural numbers and $\oplus$ to describe addition. Each process sets its initial value to 1 then computes a prefix scan. It follows directly from the definition of the scan that each process will end up with a sum that describes its rank order in the tournament tree leaves – providing the needed renaming from 1 to n. To rename only a subset of k participating processes, it is sufficient for these participants to set their initial value to 1 and the non-participants to set their initial value to 0. The sum values at the participants provide a renaming of the participants from 1 to k. In both cases, the process labeling the root of the tournament tree will learn the sum of all values during the prefix scan. It can then announce this information to the rest of the network by transmitting alone in the first round after the scan or disseminating the information during the scan's down sweep. The result is that all processes learn the number of participants in the renaming, which, in the case of all processes participating, is also the network size n.

Basic Aggregation Functions: Max, Min, Sum, Mean. It is straightforward to compute basic aggregation function on the process inputs using distributed prefix scans. Here we consider max, min, sum and mean functions (where the mean function assumes the values are numerical). In all four cases, the first step is to run the renaming and network sizing strategy described above. At the end of step, the processes are renamed from 1 to n and all processes know n. This requires one prefix scan. To calculate max or min, the processes can perform another prefix scan with $\oplus$ as the max or min operator, respectively. In both cases, both the root and the process renamed n will end up with the correct value. To calculate the sum of the initial values, it is sufficient to perform a scan with $\oplus$ describing addition. As before, the root and the process renamed n will end up with the sum. Either of these processes can then calculate the mean by simply dividing this sum by the network size n.

Leader Election and Consensus. Leader election and consensus are key primitives in distributed system design. If we assume all processes are participating in the primitive, we can solve both problems directly after the abstraction setup mode completes: the process that labels the root can declare itself leader or announce its initial value as the decision for consensus. On the other hand, if we assume only an arbitrary subset of the processes have an initial value or are interested in becoming leader, we can leverage the distributed prefix scan to break symmetry among this unknown set of participants. In more detail, the processes execute a max computation using the prefix scan as described above. In the case of leader election, the participating processes use their unique id as their input value, and in the case of consensus they use their input value. The non-participating processes set their input value to some default minimum value from the domain that is guaranteed to be at least as small as any id or initial value. At the end of the scan, the initial value sum identifies a single leader or correct decision value, which can then be spread to the full network as above.

Packet Scheduling. The packet scheduling problem assumes that each process i has some request $r_i \geq 0$ describing the number of rounds it needs to broadcast alone on the channel to deliver some important information to the network or perhaps a nearby base station. Let $T = \sum_{i=1}^{n} r_i$ be the total number of rounds needed by all the processes. The packet scheduling problem requires the processes to agree on a transmission schedule that provides

each process i at least r_i consecutive rounds to send its information. A distributed prefix scan provides an eloquent and efficient solution to this problem. In more detail, the processes use the natural numbers as the value domain and use addition for $\oplus$. Each process i then sets its initial value for the scan as $a_i \leftarrow r_i$. At the end of the scan, the sum s_i learned by each process i describes the sum of all request values of processes at its rank or below, with $s_n = T$. The processes can use these values to define a T-round packet schedule. In particular, each process i claims the rounds $s_i - r_i + 1$ to s_i in the schedule. The result is an optimal length schedule in which each process gets its requested number of rounds in a contiguous interval of the schedule.

References

1. R. Bar-Yehuda, O. Goldreich, and A. Itai. On the time-complexity of broadcast in multi-hop radio networks: An exponential gap between determinism and randomization. *J. Computer and System Sciences*, 45(1):104–126, 1992.
2. Guy E Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, 38(11):1526–1538, 1989.
3. Marijke H.L. Bodlaender, Magnús M. Halldórsson, and Pradipta Mitra. Connectivity and aggregation in multihop wireless networks. In *PODC*, pages 355–364. ACM, 2013.
4. K. Censor-Hillel, S. Gilbert, F. Kuhn, N. Lynch, and C. Newport. Structuring unreliable radio networks. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, 2011.
5. Yin Chen and Andreas Terzis. On the mechanisms and effects of calibrating RSSI measurements for 802.15.4 radios. In *EWSN*, pages 256–271. Springer, 2010.
6. I. Chlamtac and S. Kutten. On broadcasting in radio networks–problem analysis and protocol design. *IEEE Transactions on Communications*, 33(12):1240–1246, 1985.
7. Sebastian Daum, Seth Gilbert, Fabian Kuhn, and Calvin Newport. Broadcast in the ad hoc SINR model. In *Proc. 27th Symposium on Distributed Computing (DISC)*, pages 358–372, 2013.
8. Jeremy T Fineman, Seth Gilbert, Fabian Kuhn, and Calvin Newport. Contention resolution on a fading channel. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, pages 155–164. ACM, 2016.
9. A. Goldberg, S. Plotkin, and G. Shannon. Parallel symmetry-breaking in sparse graphs. *SIAM J. Discrete Math.*, 1(4):434–446, 1988.
10. Magnús M. Halldórsson, Stephan Holzer, and Nancy A. Lynch. A local broadcast layer for the SINR network model. In *PODC*, pages 129–138, 2015.
11. Magnús M. Halldórsson and Pradipta Mitra. Distributed connectivity of wireless networks. In *PODC*, pages 205–214, 2012.
12. Magnús M. Halldórsson and Pradipta Mitra. Wireless connectivity and capacity. In *SODA*, pages 516–526, 2012.
13. Magnús M Halldórsson, Yuexuan Wang, and Dongxiao Yu. Leveraging multiple channels in ad hoc networks. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 431–440. ACM, 2015.
14. Nathaniel Hobbs, Yuexuan Wang, Qiang-Sheng Hua, Dongxiao Yu, and Francis CM Lau. Deterministic distributed data aggregation under the SINR model. In *International Conference on Theory and Applications of Models of Computation*, pages 385–399. Springer, 2012.
15. Fabian Kuhn, Nancy Lynch, and Calvin Newport. The abstract MAC layer. *Distributed Computing*, 24(3):187–206, 2011.

16 Fabian Kuhn, Nancy Lynch, Calvin Newport, Rotem Oshman, and Andrea Richa. Broadcasting in unreliable radio networks. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 336–345. ACM, 2010.

17 Hongxing Li, Chuan Wu, Qiang-Sheng Hua, and Francis CM Lau. Latency-minimizing data aggregation in wireless sensor networks under physical interference model. *Ad Hoc Networks*, 12:52–68, 2014.

18 Thomas Moscibroda. The worst-case capacity of wireless sensor networks. In *IPSN*, pages 1–10, 2007.

19 Thomas Moscibroda and Roger Wattenhofer. The complexity of connectivity in wireless networks. In *INFOCOM*, pages 1–13, 2006.

20 Thomas Moscibroda, Roger Wattenhofer, and Aaron Zollinger. Topology control meets SINR: the scheduling complexity of arbitrary topologies. In *MobiCom*, pages 310–321, 2006.

21 Dongjin Son, Bhaskar Krishnamachari, and John Heidemann. Experimental study of concurrent transmission in wireless sensor networks. In *SenSys*, pages 237–250. ACM, 2006.

Two-Party Direct-Sum Questions Through the Lens of Multiparty Communication Complexity*

Itay Hazan[1] and Eyal Kushilevitz[2]

1 **Department of Computer Science, Technion – Israel Institute of Technology, Haifa, Israel**
itay.hzn@gmail.com

2 **Department of Computer Science, Technion – Israel Institute of Technology, Haifa, Israel**
eyalk@cs.technion.ac.il

Abstract

Direct-sum questions in (two-party) communication complexity ask whether two parties, Alice and Bob, can compute the value of a function f on ℓ inputs $(x_1, y_1), \ldots, (x_\ell, y_\ell)$ more efficiently than by applying the best protocol for f, independently on each input (x_i, y_i). In spite of significant efforts to understand these questions (under various communication-complexity measures), the general question is still far from being well understood.

In this paper, we offer a multiparty view of these questions: The direct-sum setting is just a two-player system with Alice having inputs $x_1, \ldots, x_\ell$, Bob having inputs $y_1, \ldots, y_\ell$ and the desired output is $f(x_1, y_1), \ldots, f(x_\ell, y_\ell)$. The naive solution of solving the ℓ problems independently, is modeled by a network with ℓ (disconnected) pairs of players Alice_i and Bob_i, with inputs x_i, y_i respectively, and communication only within each pair. Then, we consider an intermediate ("star") model, where there is one Alice having ℓ inputs $x_1, \ldots, x_\ell$ and ℓ players $\text{Bob}_1, \ldots, \text{Bob}_\ell$ holding $y_1, \ldots, y_\ell$, respectively (in fact, we consider few variants of this intermediate model, depending on whether communication between each Bob_i and Alice is point-to-point or whether we allow broadcast). Our goal is to get a better understanding of the relation between the two extreme models (i.e., of the two-party direct-sum question). If, for instance, Alice and Bob can do better (for some complexity measure) than solving the ℓ problems independently, we wish to understand what intermediate model already allows to do so (hereby understanding the "source" of such savings). If, on the other hand, we wish to prove that there is no better solution than solving the ℓ problems independently, then our approach gives a way of breaking the task of proving such a statement into few (hopefully, easier) steps.

We present several results of both types. Namely, for certain complexity measures, communication problems f and certain pairs of models, we can show gaps between the complexity of solving f on ℓ instances in the two models in question; while, for certain other complexity measures and pairs of models, we can show that such gaps do not exist (for any communication problem f). For example, we prove that if only point-to-point communication is allowed in the intermediate "star" model, then significant savings are impossible in the public-coin randomized setting. On the other hand, in the private-coin randomized setting, if Alice is allowed to broadcast messages to all Bobs in the "star" network, then some savings are possible. While this approach does not lead yet to new results on the original two-party direct-sum question, we believe that our work gives new insights on the already-known direct-sum results, and may potentially lead to more such results in the future.

1998 ACM Subject Classification F.1.3 Complexity Measures and Classes, F.2.2 Nonnumerical Algorithms and Problems

* This work was partially supported by ISF grant 1709/14, BSF grant 2012378, and NSF-BSF grant 2015782.

© Itay Hazan and Eyal Kushilevitz;
licensed under Creative Commons License CC-BY
31st International Symposium on Distributed Computing (DISC 2017).
Editor: Andréa W. Richa; Article No. 26; pp. 26:1–26:15

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

Keywords and phrases Communication Complexity, Direct Sum, Multiparty Communication

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.26

1 Introduction

Communication complexity, presented by Yao [27], studies computational problems in a distributed model, where the input is split between two parties or more. The parties exchange messages according to a predetermined protocol in order to solve the computational problem in question, e.g. computing a function on their inputs. The complexity of such protocol is measured by the number of bits exchanged, on the worst possible input. The communication complexity of a problem is the cost of the best protocol that solves it. The literature deals with finding both upper and lower bounds for various computational problems, and various types of protocols (deterministic, randomized, etc). In the two-party model, the most extensively studied model in communication complexity, Alice receives an x and Bob receives a y, both n-bit strings. Together they wish to solve the problem $P(x, y)$. For an overview of communication complexity and some of its applications, see [17].

Direct-sum questions ask whether solving several computational problems simultaneously can be done more efficiently than merely solving each problem separately. The direct sum question in two-party communication complexity, first presented in [16], is the following: suppose that Alice and Bob attempt to solve a computational problem $P(x, y)$, and suppose that the cost of the best protocol for solving it is C. Now suppose Alice and Bob are each given a sequence of ℓ inputs for P, i.e. Alice is given $x_1, \ldots, x_\ell$ and Bob is given $y_1, \ldots, y_\ell$. Alice and Bob wish to find a solution for each of the instances, namely to compute $P(x_i, y_i)$ for every $i \in \{1, \ldots, \ell\}$. Clearly, this can be done by running the best protocol that solves the problem ℓ times, using ℓC bits. However, perhaps Alice and Bob can utilize the fact that they are given all ℓ inputs at once, and solve P on all of them with fewer bits of communication. If this is possible, we say that a *saving* occurs. The question of existence of such savings is the direct-sum question, namely: does any protocol for ℓ inputs encapsulates a protocol for a single input whose cost is at most $1/\ell$ the cost of the original protocol?

1.1 Our Multiparty Models of Communication

In an attempt to achieve a better understanding of the source of hardness of direct-sum questions in the two-party case, we consider several "intermediate" *multiparty* communication models with one Alice and ℓ Bobs, denoted $\text{Bob}_1, \ldots, \text{Bob}_\ell$. Alice receives $x_1, \ldots, x_\ell$, and Bob_i receives y_i for every $i \in \{1, \ldots, \ell\}$. As in the classical direct sum question, they wish to compute $P(x_i, y_i)$ for every i. In our "intermediate" models, the question is whether a saving can be obtained when one party sees ℓ instances at once, and may send messages that are "global", while each of the other parties sees only one instance and sends messages that rely solely on its instance and its view of the communication.

Specifically, we consider five communication models, to which we refer as M_1 to M_5. The first model, M_1, is the classical two-party direct-sum model, and the last one, M_5, represents ℓ independent computations. Ultimately, the direct-sum question aims towards a better understanding of the connection between M_1 and M_5. In order to do so, M_2, M_3, and M_4 are defined, such that each model presents an additional constraint over the previous model. The definitions and motivations of these models are as follows:

- **The Classical Direct-Sum Model (M_1).** In this model, there are two parties, Alice and Bob. Alice receives an ℓ-tuple of n-bit inputs, $(x_1, \ldots, x_\ell) \in (\{0,1\}^n)^\ell$, and Bob

receives another ℓ-tuple of n-bit inputs, $(y_1, \ldots, y_\ell) \in (\{0,1\}^n)^\ell$. Together, they wish to compute $P(x_i, y_i)$ for every $i \in [\ell]$, for some computational problem P.

- **The Broadcast Model (M_2).** In this model, there are $(\ell + 1)$ parties; one Alice and ℓ Bobs, denoted $\text{Bob}_1, \ldots, \text{Bob}_\ell$. Alice receives an ℓ-tuple of n-bit inputs, $(x_1, \ldots, x_\ell) \in (\{0,1\}^n)^\ell$, and each Bob_i receives an n-bit input, $y_i \in \{0,1\}^n$. Together, they wish to compute $P(x_i, y_i)$ for every $i \in [\ell]$, such that when the protocol terminates, Alice knows all ℓ outputs. The communication is by broadcast among all parties; namely, every message sent by any party is received by all other parties.
- **The One-Way Broadcast Model (M_3).** This model is similar to M_2, only that the Bobs cannot hear each other; namely, every message Alice sends is heard by all Bobs, but a message sent by one of the Bobs is received by Alice alone. This model might be thought of as a communication between a satellite and ℓ ground stations – whatever the satellite transmits is heard by all ground stations, but messages from the ground stations are received only by the satellite.
- **The Point-to-Point Model (M_4).** As in M_2, M_3, the underlying setting in this model remains the $(\ell + 1)$-star. However, in this case the communication is point-to-point; namely, every message sent by one of the Bobs is received by Alice alone (i.e. the Bobs cannot send messages to each other), and every message Alice sends is sent to a single Bob of her choice.
- **The Independent Computations Model (M_5).** In this model, there are ℓ Alices and ℓ Bobs. For every $i \in [\ell]$, Alice_i is given $x_i \in \{0,1\}^n$ and Bob_i is given $y_i \in \{0,1\}^n$. They communicate over a point-to-point channel, which none of the other Alices or Bobs can hear, in order to compute $P(x_i, y_i)$. When the computation terminates, Alice_i should know $P(x_i, y_i)$. Since each Alice_i and Bob_i hear no other communication but their own, this model represents ℓ independent computations.

1.2 Previous Work

The direct sum question in communication complexity has been studied extensively, with respect to different types of protocols (e.g. deterministic, nondeterministic, and randomized). In spite of significant efforts it is far from being well understood.

The deterministic case was first studied in [16], where it was shown that if a certain two-party direct-sum result holds, then $\text{NC}^1 \neq \text{NC}^2$ and $\text{NC}^1 \neq \text{P}$. In [10] it is proved that for any (full) boolean function f, $\mathcal{D}(f^\ell) \geq \ell \left(\sqrt{\mathcal{D}(f)/2} - O(\log n) \right)$, while there exists a *partial* function f such that $\mathcal{D}(f) = \log n + 1$ but $\mathcal{D}\left(f^\ell\right) = O(\ell + \log n \log \ell) \ll \ell \cdot D(f)$. A setting in which the number of communication rounds is bounded has also been studied. For example, it was shown in [10, 15] that for one-round and two-round (deterministic) protocols, $\mathcal{D}(f^\ell) \geq \ell \left(\mathcal{D}(f) - O(\log n)\right)$ for any (possibly partial) function f. However, the direct-sum question for *full* functions remains open. Formally stated,

▶ **Question 1.** *Does* $\mathcal{D}\left(f^\ell\right) \geq \ell(\mathcal{D}(f) - O(1))$*, for every full function* f*?*

In the case of *randomized* protocols, one may consider several types of randomness. In the *private-coin* setting, each party has a private string of random bits that it can utilize in its computation. In the *public-coin* setting, the string of random bits is public, namely seen by both parties. In [10], a concrete (full) function f that satisfies $\mathcal{R}^{\text{priv}}(f) = O(\log n)$ and $\mathcal{R}^{\text{priv}}\left(f^\ell\right) = O(\ell + \log n)$ was shown, thus demonstrating that savings can be obtained in the private-coin randomized setting.

One might also consider a hybrid of the private and public-coin models, in which each party sees both a private and a public string of random bits. This setting arises naturally

when applying information theoretic techniques to communication complexity. Such notions were first introduced in [9] (later redefined in [3]), to measure the amount of *information* that must be revealed by the two parties, about their inputs, in order to solve a communication problem. Informally, since the amount of information revealed by the parties in a protocol is at most the number of bits transmitted throughout its execution, then one can obtain lower bounds on the communication complexity of a function by proving lower bounds on its information complexity. In recent years, *information complexity* became a powerful tool for understanding communication complexity and was used to prove many results, e.g. for reproving a lower bound of $\Omega(n)$ on the communication complexity of the set-disjointness function [2] (originally proved in [14, 23]). In fact, the main theorem in [2] is a direct-sum-like theorem for information complexity. Furthermore, it was shown in [6, 3] that $\mathcal{R}\left(f^{\ell}\right)$ approaches $\ell \cdot \mathrm{IC}\left(f\right)$ as ℓ tends to infinity. Therefore, a two-party direct-sum question in which both public- and private-coin randomization are allowed, can also be stated in terms of compression. Informally stated: given a protocol π, can one construct a "compressed" protocol τ such that $|\tau|$ is roughly equal to the information content of π? This was proven to be false in some settings; in [11, 13], a randomized setting in which the inputs are distributed according to some known distribution was studied, and it was shown that there might be an exponential gap between information and communication complexity in this setting. In [12], a randomized *non-distributional* setting was studied, and it was shown that exponential gaps between information and communication complexity can also be found in this setting, when considering search problems. Nonetheless, the question of compression remains open for *functions* in the randomized *non-distributional* setting.

Our proposed models are intended to naturally relate to the two-party direct-sum problem, and for that reason we require that the same function $f(x, y)$ is computed "on every edge". Contrary to our models, in most previous works, e.g. [8, 4, 5, 1], each of the ℓ parties receives an n-bit input, x_i, and together they compute some "global" function $g(x_1, \ldots, x_\ell)$ rather than a "local" function $f(x, y)$ "on every edge", in our case. To the best of our knowledge, a setting in which a function f is computed "on every edge" was only considered in [22] and [7]. In [22], a direct-sum-like theorem in the randomized case was proved, with respect to some communication complexity measure, denoted $ED^{\epsilon}_{\mu}(f)$ (on which we shall not elaborate), and it was shown that in the message passing model $\mathcal{R}^{\mathrm{pub}}\left(f^{\ell}, \epsilon\right) = \Omega\left(\ell \cdot ED^{\epsilon}_{\mu}(f)\right)$, for any function f and error probability ϵ. In [7], *quantum* nondeterministic multiparty communication complexity was considered, with which we do not deal in this work.

1.3 Our Results

In Section 3, public-coin randomized communication complexity is studied. First, we formalize several notions of randomized and distributional communication complexity in Section 3.1, and prove some useful connections between the different measures we present. Afterwards, in Section 3.2, we prove our main result in the public-coin setting (Theorem 12). It states that solving ℓ instances of a function f in M_4 in the public-coin model costs roughly ℓ times the cost of solving a single instance.

In the full version of the paper, a similar result is shown in the *private-coin* randomized setting, for functions f that satisfy a certain constraint. We also present a function for which there is a gap between M_3 and M_4 in this setting. For *nondeterministic* communication complexity, we prove that M_1, M_2, and M_3 are almost equivalent and that M_4 and M_5 are almost equivalent; this is also omitted here, for lack of space, and included in the full version.

Interestingly, the point-to-point model proved to be hard in both the randomized and nondeterministic settings. These results imply that the fact that Alice must send each Bob a separate message makes practically any savings impossible.

In Section 4, we consider the connections between our five models in a setting where the number of *rounds* is bounded. In particular, for one-round protocols, we show that a saving never occurs when solving a computational problem in either M_3, M_4, or M_5 (Observation 18); that saving never occurs in M_2 in the public-coin randomized setting (Claim 20); and we demonstrate a gap between M_2 and M_3 in the deterministic setting. Finally, in Theorem 25, we show a gap between M_1 and M_2 in the deterministic setting, for some relaxed notion of one-round protocols.

2 Preliminaries

Although most of our results apply to general functions, we focus our discussion on Boolean functions, for simplicity. Thus, unless explicitly stated otherwise, f is a function $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$. Given such f and a natural non-zero number ℓ, let $f^\ell : \{0,1\}^{n\ell} \times \{0,1\}^{n\ell} \to \{0,1\}^\ell$ denote the following function: for every $(\vec{x},\vec{y}) = ((x_1,\ldots,x_\ell),(y_1,\ldots,y_\ell)) \in \{0,1\}^{n\ell} \times \{0,1\}^{n\ell}$, let $f^\ell(\vec{x},\vec{y}) \triangleq (f(x_1,y_1),\ldots,f(x_\ell,y_\ell))$.

Our models, as defined in Section 1.1, aim to naturally relate to the two-party direct-sum question and, hence, throughout this work, we study *asynchronous* protocols whose communication complexity is defined to be the total number of bits sent between the parties. In a protocol π, in all models, the messages each party sends rely solely on its current view of the state of the protocol, i.e. its input (including, possibly, its randomness) and the previous messages it received in the protocol. We further assume that each party knows when it is its turn to speak based on its view of the state of the protocol. Therefore, all messages sent in the protocol are self terminating, e.g. drawn from some prefix code. One may also consider protocols that operate in *synchronous rounds*, as commonly done in the study of distributed computing (see, e.g., [21]). This may seem to be a minor difference but it is, in fact, crucial: In synchronous protocols, parties may exchange information even in rounds in which they do not speak; merely the fact that they remain silent may convey information.

We denote the worst-case communication complexity of π, i.e. the number of bits sent in π on the worst possible input, by $|\pi|$. Given a computational problem P, and a complexity measure $C \in \{\mathcal{D},\mathcal{N},\mathcal{R},\ldots\}$ (i.e. Deterministic, Nondeterministic, Randomized, etc.), we denote the communication complexity, with respect to the measure C, of a single instance of P in the two-party model by $C(P)$. Furthermore, given $\ell \geq 1$, we define $C_i(P^\ell)$ to be the communication complexity, w.r.t. the measure C, of computing ℓ instances of P in the model M_i. For example, using this notation, $C_5(P^\ell) = \ell C(P)$.

The models $M_1,\ldots,M_5$ were defined such that each model presents an additional constraint over the former models (see Section 1.1). Therefore, intuition suggests that for any $i \in [4]$ and any computational problem P, solving P^ℓ in M_{i+1} is at least as hard as solving it in M_i. This intuition can be easily formalized in a claim that informally states that $C_1 \leq C_2 \leq C_3 \lesssim C_4 \leq C_5$, for any complexity measure C. The formal claim, along with its proof, appear in the full version of the paper.

3 Public-coin Randomized Communication Complexity

In this section, we consider *randomized* communication complexity, in the *public-coin* setting, where the players have access to a common (global) random string. The *private-coin* case, where each player has its own randomness, is deferred, for lack of space, to the full version of the paper.

3.1 Randomized and Distributional Communication Complexity

Several definitions of the randomized and distributional settings, differentiated by the way error is measured, have been considered in the communication complexity literature. This section deals with the various definitions and relates the different measures to one another. The main result of this section, showing that M_4 and M_5 are "close", appears in Subsection 3.2. We note that all the definitions presented hereafter assume ℓ instances for some boolean function f, and the definitions for the two-party setting follow by fixing $\ell = 1$.

▶ **Definition 2** (Public-coin randomized protocols)**.** A protocol π is said to be *public-coin randomized* protocol if at the beginning of every execution of π, each party receives, in addition to its input, the same (public) random string r of unbounded length. Then, the parties communicate according to a predetermined (deterministic) protocol, where the type of communication between the parties is determined by the model in question (i.e. M_1, M_2, M_3, M_4, or M_5). A protocol π is said to compute f^ℓ with ϵ-error if $\Pr_r\left[\pi(\vec{x},\vec{y},r) = f^\ell(\vec{x},\vec{y})\right] \geq 1-\epsilon$ for every $(\vec{x},\vec{y}) \in \{0,1\}^{n\ell} \times \{0,1\}^{n\ell}$. Namely, the error is considered over *all instances simultaneously*. Let $\mathcal{R}_k^{\text{pub}}\left(f^\ell,\epsilon\right)$ denote the cost of the best public-coin randomized protocol that computes f^ℓ with ϵ-error in the model M_k.

▶ **Definition 3** (Distributional protocols)**.** Let ρ be a distribution over $\{0,1\}^{n\ell} \times \{0,1\}^{n\ell}$. A *deterministic* protocol π is said to be (ρ,ϵ)-*distributionally correct* for f^ℓ if it answers correctly on at least a $(1-\epsilon)$-fraction of the inputs, weighted by ρ, i.e. $\Pr_{(\vec{x},\vec{y})\sim\rho}\left[\pi(\vec{x},\vec{y}) = f^\ell(\vec{x},\vec{y})\right] \geq 1-\epsilon$. Let $\mathcal{D}_k^{(\rho,\epsilon)}\left(f^\ell\right)$ denote the cost of the best (ρ,ϵ)-distributional protocol for f^ℓ in M_k.

The following theorem relates the two measures defined above.

▶ **Theorem 4** (Yao's minimax principle)**.** $\mathcal{R}_k^{\text{pub}}\left(f^\ell,\epsilon\right) \geq \mathcal{D}_k^{(\rho,\epsilon)}\left(f^\ell\right)$ *for any distribution* ρ *over* $\{0,1\}^{n\ell} \times \{0,1\}^{n\ell}$*. Furthermore, if* $\ell = 1$*, there exists a distribution* ρ *over* $\{0,1\}^n \times \{0,1\}^n$ *for which* $\mathcal{R}^{\text{pub}}\left(f,\epsilon\right) = \mathcal{D}^{(\rho,\epsilon)}\left(f\right)$*.*

Yao's Minimax principle was first proved in [28] for the two-party case, and later generalized for the multiparty case; see, [25, 26]. It relates two settings: in the public-coin randomized setting, error is taken over the choice of randomness, while in the distributional setting, error is taken over the choice of inputs. One may also consider combinations of the two settings, e.g. the *randomized distributional setting* that appears e.g. in [11, 12, 13]. We now define several such measures.

▶ **Definition 5** (Randomized distributional protocols)**.** Let ρ be a distribution over $\{0,1\}^{n\ell} \times \{0,1\}^{n\ell}$. A *public-coin* randomized protocol π is said to be (ρ,ϵ)-*distributionally correct* for f^ℓ in M_k if it produces a correct answer with probability at least $1-\epsilon$, taken over *both* the choice of randomness and the choice of inputs, i.e. $\Pr_{r,(\vec{x},\vec{y})\sim\rho}\left[\pi(\vec{x},\vec{y},r) = f^\ell(\vec{x},\vec{y})\right] \geq 1-\epsilon$. Let $\mathcal{R}_k^\rho\left(f^\ell,\epsilon\right)$ denote the cost of the best public-coin randomized (ρ,ϵ)-distributional protocol for f^ℓ in M_k, and let $\overline{\mathcal{R}}_k^\rho\left(f^\ell,\epsilon\right)$ denote the minimal *expected* cost of any public-coin randomized (ρ,ϵ)-distributional protocol for f^ℓ in M_k, where the expectaion is taken both over the choice of randomness and the choice of input.

Next, we introduce another communication complexity measure for the classical two-party setting – *public-coin randomized communication complexity with* (ρ,δ)*-promise and* ϵ*-error*. Although this definition can be easily extended to other communication models, we only need the two-party version for our purposes.

▶ **Definition 6** (Public-coin randomized protocols with (ρ,δ)-promise and ϵ-error)**.** Let ρ be a distribution on $\{0,1\}^n \times \{0,1\}^n$, and let $\epsilon,\delta \in (0,1)$. A public-coin randomized protocol π is said to compute f *with* (ρ,δ)*-promise and* ϵ*-error* if:

Protocol 1 $\tau(x, y, r)$.

1: **Simulation**: Alice and Bob simulate $\pi(x, y, r)$.
2: **Early termination**: If more than $\frac{1}{\delta^2}|\pi|$ bits were sent, Alice and Bob terminate and output '0'. Otherwise, Alice answers like π.

1. On a δ-fraction of the inputs, weighted by ρ, there is no correctness promise, i.e. the protocol may always err. These are called *type-A* inputs of π.
2. On any other input, a $(1-\delta)$-fraction weighted by ρ, there is at most ϵ-error, weighted by the choice of the public random string. These are called *type-B* inputs of π.

Let $\mathcal{R}^{(\rho,\delta)}(f, \epsilon)$ denote the cost of the best public-coin randomized protocol that computes f with (ρ, δ)-promise and ϵ-error.

In the rest of this subsection, we discuss the settings defined above, and show how they relate to one another. We start by the following observation:

▶ **Observation 7.** *$\mathcal{R}^{\rho}_{k}\left(f^{\ell}, \epsilon\right) \leq \mathcal{R}^{\text{pub}}_{k}\left(f^{\ell}, \epsilon\right)$ for any f, ℓ, $\epsilon \in (0,1)$ and distribution ρ.*

This is immediate: suppose π is a public-coin randomized protocol that errs with probability at most ϵ on *every* input (over the choice of randomness). Clearly, π errs with probability at most ϵ if the input is also sampled from some distribution ρ.

▶ **Lemma 8.** *$\mathcal{R}^{(\rho,\delta)}(f, \epsilon) \geq \mathcal{D}^{(\rho,\delta+\epsilon)}(f)$, for any f, $\epsilon, \delta \in (0,1)$ and distribution ρ.*

The proof of Lemma 8 is an immediate generalization of the first part of Theorem 4 and is omitted for lack of space. We conclude this subsection with the following lemma:

▶ **Lemma 9.** *$\mathcal{R}^{\left(\rho,\sqrt{\epsilon}+\delta\right)}(f, \sqrt{\epsilon}+\delta) \leq \frac{1}{\delta^2}\overline{\mathcal{R}}^{\rho}(f, \epsilon)$ for any f, distribution ρ, and $\epsilon, \delta \in (0,1)$ that satisfy $2(\sqrt{\epsilon}+\delta) \leq 1$.*

Proof. Let π be an expected (ρ, ϵ)-distributional randomized protocol for f. For every input pair (x, y) and any public random string r, let $I(x, y, r)$ be the following $\{0,1\}$-indicator: $I(x, y, r) = 1$ if and only if π errs on (x, y) when the random string is r. Furthermore, let $I(x, y) \triangleq \mathop{\mathbb{E}}_{r}[I(x, y, r)]$. Observe that $I(x, y)$ is exactly $\Pr_r[\pi(x, y, r) \neq f(x, y)]$, and since π is a (ρ, ϵ)-distributional randomized protocol for f, then $\mathop{\mathbb{E}}_{(x,y)\sim\rho}[I(x, y)] \leq \epsilon$.

For every (x, y), let $E(x, y) \triangleq \mathop{\mathbb{E}}_{r}[|\pi(x, y, r)|]$ denote the expected communication complexity of π on (x, y), taken over the choice of public randomness r. By the definition of π, we have that $\mathop{\mathbb{E}}_{(x,y)\sim\rho}[E(x, y)] = |\pi|$, where here $|\pi|$ denotes the expected communication cost of π (since π is an expected randomized distributional protocol).

We construct a protocol τ for f in Protocol 1.

We claim that τ is a public-coin randomized protocol for f with $(\delta + \sqrt{\epsilon})$-promise and $(\delta + \sqrt{\epsilon})$-error. To do so, we separate the input space of π into two sets, type-A inputs and type-B inputs, as follows: an input (x, y) is said to be a type-A input of τ if and only if

$$E(x, y) \geq \frac{1}{\delta}|\pi| \quad \text{or} \quad I(x, y) \geq \sqrt{\epsilon}.$$

▶ **Claim 10.** *$\Pr_{(x,y)\sim\rho}[(x, y)$ is a type-A input of $\tau] \leq \delta + \sqrt{\epsilon}$.*

Proof. First, $\mathop{\mathbb{E}}_{(x,y)\sim\rho}[E(x, y)] = |\pi|$ and, by applying Markov's inequality,

$$\Pr_{(x,y)\sim\rho}\left[E(x, y) \geq \frac{1}{\delta}|\pi|\right] \leq \Pr_{(x,y)\sim\rho}\left[E(x, y) \geq \frac{1}{\delta}\mathop{\mathbb{E}}_{(x,y)\sim\rho}[E(x, y)]\right] \leq \delta.$$

Similarly, $\mathop{\mathbb{E}}_{(x,y)\sim\rho}[I(x,y)] \leq \epsilon$, and by applying Markov's inequality,

$$\Pr_{(x,y)\sim\rho}\left[I(x,y) \geq \sqrt{\epsilon}\right] \leq \Pr_{(x,y)\sim\rho}\left[I(x,y) \geq \frac{1}{\sqrt{\epsilon}}\mathop{\mathbb{E}}_{(x,y)\sim\rho}[I(x,y)]\right] \leq \sqrt{\epsilon}.$$

A union bound argument yields that

$$\Pr_{(x,y)\sim\rho}[(x,y) \text{ is a type-A input of } \tau] \leq \delta + \sqrt{\epsilon},$$

and the claim follows. ◀

▶ **Claim 11.** *Given that (x,y) is a type-B input of τ, $\Pr_r[\tau(x,y,r) \neq f(x,y)] \leq \delta + \sqrt{\epsilon}$.*

Proof. Since (x,y) is a type-B input of τ, then $\mathop{\mathbb{E}}_r[|\tau(x,y,r)|] \leq \frac{1}{\delta}|\pi|$ and $\mathop{\mathbb{E}}_r[I(x,y,r)] \leq \sqrt{\epsilon}$. Observe that τ might err in either of two cases: τ was early-terminated, or τ was not early terminated but the simulation of π answered incorrectly. By the union bound, we conclude the following:

$$\begin{aligned}
\Pr_r[\tau(x,y,r) \neq f(x,y)] &\leq \Pr_r\left[|\tau(x,y,r)| > \frac{1}{\delta^2}|\pi|\right] + \Pr_r[I(x,y,r) = 1] \\
&\leq \Pr_r\left[|\tau(x,y,r)| > \frac{1}{\delta}\mathop{\mathbb{E}}_r[|\tau(x,y,r)|]\right] + \mathop{\mathbb{E}}_r[I(x,y,r)] \\
&\leq \delta + \sqrt{\epsilon},
\end{aligned}$$

where the last inequality follows from Markov's inequality. ◀

In conclusion, Claim 10 proves that there are at most $(\delta + \sqrt{\epsilon})$ type-A inputs of τ, weighed by ρ, and Claim 11 proves that τ has at most $(\delta + \sqrt{\epsilon})$-error on type-B inputs. Therefore, τ is indeed a public-coin randomized protocol that computes f with $(\rho, \sqrt{\epsilon} + \delta)$-promise and $(\sqrt{\epsilon} + \delta)$-error. Step 2 (early termination) assures that $|\tau| \leq \frac{1}{\delta^2}|\pi|$, and that concludes the proof of Lemma 9. ◀

We remark that we have dealt with several communication complexity measures in this subsection and, for conciseness reasons, some of the connections between the different measures were omitted. However, the omitted connections can be shown, either by simply combining the connections we have proved, or by slightly modifying the arguments presented in our proofs. For instance, an argument similar to that of Lemma 9 proves that $\mathcal{R}^{(\rho,\sqrt{\epsilon})}(f, \sqrt{\epsilon}) \leq \mathcal{R}^{\rho}(f, \epsilon)$. For another example, one can prove that there exists a distribution ρ over $\{0,1\}^n \times \{0,1\}^n$ for which $\mathcal{R}^{(\rho,\delta)}(f,\epsilon) \geq \mathcal{R}^{\text{pub}}(f, \delta + \epsilon)$ using Theorem 4 and Theorem 8.

3.2 Pushing M_4 Towards M_5

In this section, we prove that computing ℓ instances of f in the point-to-point model, M_4, cannot be done much more efficiently than just solving each instance separately, as in the independent computations model, M_5. From a more philosophical point of view, designing protocols in which Alice sends "global" messages is virtually useless in the public-coin randomized setting when only point-to-point communication is allowed. Formally stated:

▶ **Theorem 12.** *$\mathcal{R}^{\text{pub}}(f, 2(\sqrt{\epsilon} + \delta)) \leq \frac{1}{\delta^2}\frac{1}{\ell}\mathcal{R}_4^{\text{pub}}\left(f^\ell, \epsilon\right)$, for any f, ℓ, and $\epsilon, \delta \in (0,1)$ such that $2(\sqrt{\epsilon} + \delta) \leq 1$.*

Protocol 2 $\tau(x,y,r)$.

1: **Preparation**: Alice and Bob split the random string r into two independent random strings, $r \triangleq (r_1, r_2)$.
2: **Augmentation**: Alice and Bob construct an input $(\vec{x},\vec{y})$ for π from their (x,y):
2.1: Alice and Bob sample $i \sim \text{Unif}\{[\ell]\}$ from the randomness r_1.
2.2: Alice samples $(\vec{u},\vec{v}) = ((u_1,\ldots,u_{\ell-1}),(v_1,\ldots,v_{\ell-1})) \sim \rho^{\ell-1}$ from r_1.
2.3: Let $\vec{x} \triangleq \text{aug}\,[\vec{u},x,i]$, and let $\vec{y} \triangleq \text{aug}\,[\vec{v},y,i]$.
3: **Simulation**: Alice and Bob simulate the i'th channel of $\pi(\vec{x},\vec{y},r_2)$.
That is, Bob plays the role of Dan_i, while Alice plays the role of Carol and all other Dans. In their simulation, Alice and Bob only send messages that are sent between Carol and and Dan_i in π. All other messages are simulated by Alice alone, with no additional communication.

Given a protocol π for f^ℓ in M_4, our proof constructs a protocol τ for f (a single instance in the two-party model) such that $|\tau| \leq \frac{1}{\delta^2}\frac{1}{\ell}|\pi|$. The construction of τ is based on the *symmetrization* technique, that was introduced in [22], and was later used in, e.g., [24, 25]. In the core of the symmetrization technique lies an intuitive averaging argument: suppose we fix some input to each of the parties in M_4. In that case, the average number of bits communicated on a uniformly-chosen channel is at most $\frac{1}{\ell}|\pi|$. In τ, Alice and Bob augment their single instance (x,y) to an input $(\vec{x},\vec{y})$ for π, that contains ℓ instances of f, and then simulate a channel of $\pi(\vec{x},\vec{y})$. We thus define an *augmentation operator*:

▶ **Definition 13** (The Augmentation operator)**.** Let Q be any set, and $k \in \mathbb{N}$ an integer. Given a k-tuple $\vec{q} = (q_1,\ldots,q_k) \in Q^k$, an element p, and an index $i \in [k+1]$, the *augmentation operator* $\text{aug}\,[\vec{q},p,i]$ is defined to be the $(k+1)$-tuple obtained by "inserting" p as an i'th element in $\vec{q}$, i.e. $\text{aug}\,[\vec{q},p,i] \triangleq (q_1,\ldots,q_{i-1},p,q_i,\ldots,q_k)$.

We now prove a central lemma from which we conclude Theorem 12:

▶ **Lemma 14.** $\overline{\mathcal{R}}^{\rho}(f,\epsilon) \leq \frac{1}{\ell}\mathcal{R}_4^{\rho^\ell}\left(f^\ell,\epsilon\right)$*, for any* f*,* ℓ*,* $\epsilon \in (0,1)$*, and* ρ *over* $\{0,1\}^n \times \{0,1\}^n$.

Proof. Let π be a randomized $\left(\rho^\ell,\epsilon\right)$-distributional protocol for f^ℓ in M_4. Given π, we construct a protocol τ for a single instance of f, such that τ is an expected randomized (ρ,ϵ)-distributional protocol. The construction of τ is presented in Protocol 2. To avoid confusion, we refer to the two parties in τ as Alice and Bob, and to the $\ell+1$ parties in π as Carol and Dans.

For every $(x,y) \in \{0,1\}^n \times \{0,1\}^n$, let $E(x,y) \triangleq \mathop{\mathbb{E}}_{r}[|\tau(x,y,r)|]$ denote the expected communication complexity of τ on (x,y), over the choice of public randomness.

▶ **Claim 15.** $\mathop{\mathbb{E}}_{(x,y)\sim\rho}[E(x,y)] \leq \frac{1}{\ell}|\pi|$.

Proof of Claim 15. For every $i \in [\ell]$ and every $(\vec{x},\vec{y}) \in \{0,1\}^{n\ell} \times \{0,1\}^{n\ell}$, let $\left|\pi^i(\vec{x},\vec{y})\right|$ denote the maximum number of bits communicated between Carol and Dan_i when running π on $(\vec{x},\vec{y})$. We thus have

$$\mathop{\mathbb{E}}_{(x,y)\sim\rho}[E(x,y)] = \mathop{\mathbb{E}}_{(x,y)\sim\rho}\left[\mathop{\mathbb{E}}_{i\sim\text{Unif}\{[\ell]\}}\left[\mathop{\mathbb{E}}_{(\vec{u},\vec{v})\sim\rho^{\ell-1}}\left[\left|\pi^i(\vec{x},\vec{y})\right|\right]\right]\right] \leq \mathop{\mathbb{E}}_{(\vec{x},\vec{y})\sim\rho^\ell}\left[\frac{1}{\ell}|\pi|\right] \leq \frac{1}{\ell}|\pi|$$

and the claim follows. ◀

For every $(\vec{x},\vec{y}) \in \{0,1\}^{n\ell} \times \{0,1\}^{n\ell}$ and for every random string r_2, let $J(\vec{x},\vec{y},r_2)$ be the following $\{0,1\}$-indicator: $J(\vec{x},\vec{y},r_2) = 1$ if and only if π errs on $(\vec{x},\vec{y})$ given the random string is r_2. Furthermore, for every $(x,y) \in \{0,1\}^n \times \{0,1\}^n$ and for every possible random string $r = (r_1, r_2)$, where $r_1 = (i,(\vec{u},\vec{v})) \in [\ell] \times \{0,1\}^{n(\ell-1)} \times \{0,1\}^{n(\ell-1)}$ and $r_2 \in \{0,1\}^*$, let $I(x,y,r) = J(\mathrm{aug}\left[\vec{u},x,i\right],\mathrm{aug}\left[\vec{v},y,i\right],r_2)$. Let $I(x,y) \triangleq \mathop{\mathbb{E}}_{r}\left[I(x,y,r)\right]$ for every $(x,y) \in \{0,1\}^n \times \{0,1\}^n$.

▶ **Claim 16.** $\mathop{\mathbb{E}}_{(x,y)\sim\rho}\left[I(x,y)\right] \leq \epsilon$.

Proof.

$$\begin{aligned}\mathop{\mathbb{E}}_{(x,y)\sim\rho}\left[I(x,y)\right] &= \mathop{\mathbb{E}}_{(x,y)\sim\rho}\left[\mathop{\mathbb{E}}_{i\sim\mathrm{Unif}\{[\ell]\}}\left[\mathop{\mathbb{E}}_{(\vec{u},\vec{v})\sim\rho^{\ell-1}}\left[\mathop{\mathbb{E}}_{r_2}\left[J(\mathrm{aug}\left[\vec{u},x,i\right],\mathrm{aug}\left[\vec{v},y,i\right],r_2)\right]\right]\right]\right] \\ &\leq \mathop{\mathbb{E}}_{(\vec{x},\vec{y})\sim\rho^{\ell}}\left[\mathop{\mathbb{E}}_{r_2}\left[J(\vec{x},\vec{y},r_2)\right]\right] \leq \epsilon,\end{aligned}$$

where the last inequality holds since π is an expected randomized (ρ^ℓ,ϵ)-distributional protocol for f^ℓ in M_4. ◀

To conclude, Claim 15 and Claim 16 show that τ is indeed an expected randomized distributional protocol for f with the desired properties, and together they imply Lemma 14. ◀

Proof of Theorem 12. As promised in Theorem 4, let ρ be a distribution on $\{0,1\}^n \times \{0,1\}^n$ that satisfies $\mathcal{D}^{(\rho,2(\delta+\sqrt{\epsilon}))}(f) = \mathcal{R}^{\mathrm{pub}}(f,2(\delta+\sqrt{\epsilon}))$. We therefore have that

$$\begin{aligned}\mathcal{R}^{\mathrm{pub}}\left(f,2\left(\delta+\sqrt{\epsilon}\right)\right) &= \mathcal{D}^{\left(\rho,2\left(\sqrt{\epsilon}+\delta\right)\right)}(f) && \text{(by choice of } \rho\text{)} \\ &\leq \mathcal{R}^{\left(\rho,\sqrt{\epsilon}+\delta\right)}\left(f,\sqrt{\epsilon}+\delta\right) && \text{(Theorem 8)} \\ &\leq \frac{1}{\delta^2}\overline{\mathcal{R}}^{\rho}(f,\epsilon) && \text{(Theorem 9)} \\ &\leq \frac{1}{\delta^2}\frac{1}{\ell}\mathcal{R}_4^{\rho^\ell}\left(f^\ell,\epsilon\right) && \text{(Theorem 14)} \\ &\leq \frac{1}{\delta^2}\frac{1}{\ell}\mathcal{R}_4^{\mathrm{pub}}\left(f^\ell,\epsilon\right), && \text{(Observation 7)}\end{aligned}$$

and the theorem holds. ◀

4 One-Round Communication

In this section, we consider several definitions of *one-round* communication protocols in our models, and examine the connections between them. We prove that savings cannot be obtained in M_3, M_4, and M_5 when considering one-round protocols, for any complexity measure (that is, deterministic, nondeterministic, and randomized). However, we prove that, under a certain definition of one-round protocols in M_2, gaps can be found between M_2 and M_3 in the private-coin randomized setting and, if search problems are taken into account, then gaps can also be found between M_1 and M_2 in the deterministic setting.

4.1 M_3, M_4 and M_5 in the One-Round Setting

▶ **Definition 17.** Let P be a computational problem, $\ell \geq 1$, and $k \in \{3,4,5\}$.
A protocol π in M_k is a *one-round protocol* if Alice does not send a message to any of the Bobs in any execution of the protocol. Given a complexity measure C, let $C_k^1\left(P^\ell\right)$ denote

the cost of the best one-round protocol that solves P^ℓ in M_k with respect to the measure C. Furthermore, let $C^1(P) \triangleq C_5^1(P)$ denote the cost of the best one-round protocol that solves P in the two-party setting with respect to the measure C.

Since the Bobs in M_3, M_4, and M_5 cannot hear each other directly in these models then, when considering one-round protocols, the message sent by each Bob_i is independent of the messages sent by the other Bobs. The following is therefore fairly easy to prove:

▶ **Observation 18.** *$C_k^1(P^\ell) \geq \ell \cdot C^1(P)$, for any $k \in \{3,4,5\}$, any computational problem P, any $\ell \geq 1$, and any complexity measure C (i.e. $C \in \{\mathcal{D}, \mathcal{N}, \mathcal{R}, \dots\}$).*

4.2 M_2 in the One-Round Setting

Contrary to the models M_3, M_4, and M_5, the Bobs are able to hear one another in M_2. This property of M_2 allows for several possible variations on one-round protocols:

1. *One-message-each protocols*, where each Bob sends one message to Alice. However, the messages are sent sequentially, as opposed to simultaneous protocols. Namely, each Bob can hear all messages sent by Bobs whose turn preceded his. The identity of the next speaker is determined by the previous messages sent in the protocol and, in the public-coin randomized setting, by the public randomness as well. Therefore, the order in which the Bobs speak may vary between different executions of the protocol.
2. *Bobs-only protocols* where Alice does not send any message but the Bobs are unconstrained, and can exchange as many messages as they wish. We remark that these are not one-round protocols per se, since the speaker may change multiple times. This collective view of the Bobs is reasonable since M_2 is an asymmetric model, in which Alice plays a different role than the Bobs.

▶ **Definition 19.** Let P be a communication problem and let $\ell \in \mathbb{N}$. Let C be any complexity measure. Let $C_2^1\left(P^\ell\right)$ denote the cost of the best one-message-each protocol that solves P^ℓ in M_2 with respect to the measure C, and let $C_2^B\left(P^\ell\right)$ denote the cost of the best Bobs-only protocol that solves P^ℓ in M_2 with respect to the measure C.

4.2.1 The Randomized Case

4.2.1.1 The Public-Coin Setting

The following claim proves that if public-coin randomness is allowed, then M_2 and M_5 are essentially equivalent when considering Bobs-only communication complexity, and significant gaps between them cannot be found with respect to this measure.

▶ **Claim 20.** *$\mathcal{R}^{1,\text{pub}}\left(f, 2(\sqrt{\epsilon}+\delta)\right) \leq \frac{1}{\delta^2}\frac{1}{\ell}\mathcal{R}_4^{B,\text{pub}}\left(f^\ell, \epsilon\right)$ for any f, ℓ, and $\epsilon, \delta \in (0,1)$ that satisfy $2(\sqrt{\epsilon}+\delta) \leq 1$.*

This claim, and its proof, resemble Theorem 12. In the proof of Theorem 12, a protocol τ that computes f in the two-party setting was constructed from a protocol π that computes f^ℓ in M_4. Intuitively, the two parties in τ choose a uniformly random $i \in [\ell]$ and simulate the i'th channel of π. Hence, $|\tau| = O\left(\frac{1}{\ell}|\pi|\right)$ by an averaging argument.

Let us try to extend this argument to M_2. As before, to avoid confusion, we refer to the two parties in τ as Alice and Bob, and to the $\ell+1$ parties in π as Carol and Dans. Assume, then, that π is a protocol for f^ℓ in M_2 (not necessarily a one-round protocol), and suppose that Carol sends at most c bits and that the Dans send at most d bits (combined) in any run of π. Since Carol uses broadcast communication in π, then the average number of bits sent

between Carol and Dan_i in π is roughly $c + \frac{1}{\ell}d$, for a uniformly-sampled $i \in [\ell]$. However, c might be very large in general protocols, and hence this averaging argument seems to provide a very weak bound in the general case. However, if $c = 0$, i.e. if π is an Bobs-only protocol, then $|\tau| = O(\frac{1}{\ell}d) = O\left(\frac{1}{\ell}|\pi|\right)$, as desired. Therefore, in the case of Bobs-only protocols, the exact same construction of τ as in Theorem 12 proves Claim 20. The formal proof is almost identical to that of Theorem 12, and is hence omitted.

4.2.1.2 The Private-Coin Setting

In the full version of the paper, we show that the equality function separates M_3 from M_4 in the unbounded-round private-coin setting, and that it also separates M_2 from M_3 in the one-round setting when considering one-message-each protocols, where the lower bound on M_3 follows immediately from Observation 18 and the known fact that $\mathcal{R}^{1,\text{priv}}(\text{EQ}, 1/3) \geq \mathcal{R}^{\text{priv}}(\text{EQ}, 1/3) = \Omega(\log n)$ (see, e.g., [17]). Thus $\mathcal{R}_3^{1,\text{priv}}\left(\text{EQ}^\ell, \epsilon\right) = \Omega(\ell \log n)$. As for the upper bound on M_2, it can be easily obtained using Newman's transformation from the public-coin setting to the private-coin setting [18]; we also show (in the full version) that $\mathcal{R}^{1,\text{pub}}(\text{EQ}, 1/3) = O(1)$ and also that $\mathcal{R}^{1,\text{priv}}\left(f^\ell, 1/3\right) = O\left(\mathcal{R}^{1,\text{pub}}\left(f^\ell, 1/3\right) + \log(n\ell)\right)$ for any function f and natural number ℓ. Using amplification, we have that $\mathcal{R}^{1,\text{pub}}\left(\text{EQ}^\ell, 1/3\right) = O(\ell \log \ell)$, and conclude that $\mathcal{R}_2^{1,\text{priv}}\left(\text{EQ}^\ell, 1/3\right) = O\left(\ell \log \ell + \log(n\ell)\right)$.

We remark that since Bobs-only protocols in M_2 are stronger than one-message-each protocols, then the gap presented also holds for Bobs-only protocols in M_2.

4.2.2 The Deterministic Case

The previous subsection shows a gap between M_2 and M_3 when considering one-message-each protocols in the private-coin setting. However, M_2 seems to behave differently in the deterministic setting, as suggested by the following claim:

▶ **Claim 21.** *$\mathcal{D}_2^1\left(P^\ell\right) \geq \ell \cdot \mathcal{D}^1(P)$, for any computational problem P and any $\ell \in \mathbb{N}$.*

Proof sketch. By induction on ℓ. The claim is clearly true for $\ell = 1$. Let $\ell \geq 2$, and let π be an optimal one-message-each protocol that solves P^ℓ in M_2. For every $i \in [\ell]$, let m_i denote the i'th message in π. We separate into cases:

Case 1. Suppose there exists a valid prefix of the transcript $m_1, \ldots, m_{\ell-1}$ such that $\sum_{i=1}^{\ell-1} |m_i| \geq \frac{\ell-1}{\ell}|\pi|$. In that case, we construct a protocol τ for P in the two-party setting by fixing these messages and letting Bob play the role of the last party in π. The communication complexity of τ is at most $\frac{1}{\ell}|\pi|$, and hence $|\pi| \geq \ell C(P)$.

Case 2. Suppose that every valid prefix of the transcript $m_1, \ldots, m_{\ell-1}$ satisfies $\sum_{i=1}^{\ell-1} |m_i| < \frac{\ell-1}{\ell}|\pi|$. In that case, we construct a protocol τ for $P^{\ell-1}$ in M_2 (with one Alice and $\ell - 1$ Bobs) by letting the Bobs play the roles of the first $\ell - 1$ parties of π, and letting Alice simulate the last Bob (with no communication). We therefore have that $|\tau| < \frac{\ell-1}{\ell}|\pi|$. Since we assumed π to be optimal, then clearly $|\pi| \leq \ell C(P)$, and we thus get $|\tau| < (\ell - 1)C(P)$, in contradiction to the induction hypothesis. ◀

Claim 21 shows that no gaps can be found between M_2 and M_3 when considering one-message-each protocols in the deterministic setting. We ask whether this is true for Bobs-only protocols as well; namely,

▶ **Question 22.** *Is it true that $\mathcal{D}_2^B\left(P^\ell\right) \geq \ell \cdot \mathcal{D}^1(P)$ for any P and any $\ell \geq 1$?*

4.2.2.1 Separating M_1 from M_2 for One-Message-Each Protocols

When considering the deterministic communication complexity of direct-sum problems, no saving is known to be achievable for full functions in the two-party setting. However, some saving can be achieved in the case of partial functions. In the rest of this section, we study such an example in our models.

▶ **Definition 23** (The NBA problem). Let $n \in \mathbb{N}$. For every $x = (u, v) \in \{0,1\}^n \times \{0,1\}^n$ and for every $y \in \{0,1\}^n$, the *NBA* function is defined as follows:

$$\mathrm{NBA}(x,y) \triangleq \begin{cases} \text{undefined} & y \notin \{u,v\} \vee u = v \\ 1 & y = u \\ 0 & y = v \end{cases}$$

Intuitively, Alice knows the names of two NBA teams, u and v, that played against each other last night. However, she does not know which of the teams had won the game. Bob, on the other hand, knows the name of the winning team, $y \in \{u, v\}$, but not the name of its opponent. The goal is for Alice to know which team had won the match.

The NBA problem was first studied in [19, 20], where it was called *The League Problem*, and it was proved that $\mathcal{D}(\mathrm{NBA}) = \log n + 1$. Then, in [10], it was proved that some saving can be achieved for its direct-sum version. In particular, an upper bound of $O(\ell + \log n \log \ell)$ was shown. The protocol can also be run in M_2 and M_3, and seems to heavily rely on Alice's ability to see all ℓ instances together. Therefore, intuition would suggest that switching the roles of Alice and Bob would put their ability to design a clever protocol for M_2 in question. We therefore define the *Inverted*-NBA problem:

▶ **Definition 24** (The INBA problem). For every $x \in \{0,1\}^n$ and $y \in \{0,1\}^n \times \{0,1\}^n$, the *Inverted-NBA* function is defined to be $\mathrm{INBA}(x,y) \triangleq \mathrm{NBA}(y,x)$.

▶ **Example 25.** Consider the INBA partial function. We claim that it presents a deterministic gap between M_1 and M_2 in the one-round setting. In particular, we examine one-message-each protocols in M_2. The lower bound in M_2 follows from Claim 21 and the fact that $\mathcal{D}(\mathrm{NBA}) = \log n + 1$, which together prove that $\mathcal{D}_2^B\left(\mathrm{INBA}^\ell\right) \geq \ell \log n$. For the upper bound on M_1, we first claim that any protocol that computes NBA^ℓ in M_1 can also be used to compute INBA^ℓ in M_1, simply by switching the roles of Alice and Bob. Furthermore, we claim (without proof) that the protocol presented in [10] can be easily turned into a one-round protocol for INBA^ℓ in M_1, and thus conclude that $\mathcal{D}_1^1\left(\mathrm{INBA}^\ell\right) = O(\ell + \log \ell \log n)$. The argument appears at length and in greater detail in the full version of the paper.

Proving that $D_2^B\left(\mathrm{INBA}^\ell\right) = \Omega(\ell \log n)$ would strengthen the gap presented in the example above. We conjecture that the INBA problem remains hard in M_2 even in the unbounded-round setting. That is: $D_2\left(\mathrm{INBA}^\ell\right) = \Omega(\ell \log n)$.

5 Conclusions and Future Work

In this work, we suggest a new approach to the study of two-party direct-sum questions in communication complexity. Future work may extend our approach in several directions.

One such direction would be to try and find more gaps and equivalences between the models we proposed with respect to various complexity measures; for instance, one may try

to extend Theorem 12 and prove that savings cannot be obtained in M_4 with respect to other measures, e.g. deterministic communication complexity. This would support our intuition that "point-to-point communication is hard". Another example is to try and separate M_1 and M_2 with respect to private-coin randomized setting. To the best of our knowledge, all currently known savings in this setting utilize Newman's transformation which also applies to the models M_2 and M_3. Therefore, the current techniques cannot be used to separate M_1 and M_2. Another interesting direction, that would require devising new functions for which savings can be obtained in the classical two-party direct-sum setting, would be to try to separate M_1 from M_2 with respect to private-coin randomized communication complexity. One may also consider a variant of the public coin randomized setting in which there is a common random string on each communication line, that is, Alice and Bob_i share a random string r_i. It may be interesting to study the connection between our five models in this setting, and compare it to other settings, e.g. to the public coin setting we discuss in the paper (in which there is one global random string, shared by all parties).

Finally, one can examine the two-party direct-sum question through the lens of other multiparty models, such as more complicated bipartite communication graphs (where one side has k Alices and the other side has t Bobs) or the clique network. Understanding these questions may also shed new light on the source of hardness of classical direct-sum questions in two-party communication complexity.

References

1 Noga Alon, Klim Efremenko, and Benny Sudakov. Testing equality in communication graphs. *arXiv preprint arXiv:1605.01658*, 2016.

2 Ziv Bar-Yossef, Thathachar S Jayram, Ravi Kumar, and D Sivakumar. An information statistics approach to data stream and communication complexity. In *Foundations of Computer Science, 2002. Proceedings. The 43rd Annual IEEE Symposium on*, pages 209–218. IEEE, 2002.

3 Mark Braverman. Interactive information complexity. *SIAM Journal on Computing*, 44(6):1698–1739, 2015.

4 Mark Braverman, Faith Ellen, Rotem Oshman, Toniann Pitassi, and Vinod Vaikuntanathan. A tight bound for set disjointness in the message-passing model. In *Foundations of Computer Science (FOCS), 2013 IEEE 54th Annual Symposium on*, pages 668–677. IEEE, 2013.

5 Mark Braverman and Rotem Oshman. The communication complexity of number-in-hand set disjointness with no promise. In *Electronic Colloquium on Computational Complexity (ECCC)*, volume 22, 2015.

6 Mark Braverman and Anup Rao. Information equals amortized communication. *IEEE Transactions on Information Theory*, 60(10):6058–6069, 2014.

7 Harry Buhrman, Matthias Christandl, and Jeroen Zuiddam. Nondeterministic quantum communication complexity: the cyclic equality game and iterated matrix multiplication. *arXiv preprint arXiv:1603.03757*, 2016.

8 Amit Chakrabarti, Subhash Khot, and Xiaodong Sun. Near-optimal lower bounds on the multi-party communication complexity of set disjointness. In *Computational Complexity, 2003. Proceedings. 18th IEEE Annual Conference on*, pages 107–117. IEEE, 2003.

9 Amit Chakrabarti, Yaoyun Shi, Anthony Wirth, and Andrew Yao. Informational complexity and the direct sum problem for simultaneous message complexity. In *Foundations of Computer Science, 2001. Proceedings. 42nd IEEE Symposium on*, pages 270–278. IEEE, 2001.

10 Tomas Feder, Eyal Kushilevitz, Moni Naor, and Noam Nisan. Amortized communication complexity. *SIAM Journal on computing*, 24(4):736–750, 1995.

11 Anat Ganor, Gillat Kol, and Ran Raz. Exponential separation of information and communication. In *Foundations of Computer Science (FOCS), 2014 IEEE 55th Annual Symposium on*, pages 176–185. IEEE, 2014.

12 Anat Ganor, Gillat Kol, and Ran Raz. Exponential separation of communication and external information. In *STOC*, pages 977–986, 2016.

13 Anat Ganor, Gillat Kol, and Ran Raz. Exponential separation of information and communication for boolean functions. *Journal of the ACM (JACM)*, 63(5):46, 2016.

14 Bala Kalyanasundaram and Georg Schintger. The probabilistic communication complexity of set intersection. *SIAM Journal on Discrete Mathematics*, 5(4):545–557, 1992.

15 Mauricio Karchmer, Eyal Kushilevitz, and Noam Nisan. Fractional covers and communication complexity. In *Structure in Complexity Theory Conference, 1992., Proceedings of the Seventh Annual*, pages 262–274. IEEE, 1992.

16 Mauricio Karchmer, Ran Raz, and Avi Wigderson. Super-logarithmic depth lower bounds via the direct sum in communication complexity. *Computational Complexity*, 5(3-4):191–204, 1995. Early version in CCC 1991.

17 Eyal Kushilevitz and Noam Nisan. *Communication Complexity*. Cambridge University Press, Cambridge, 1997.

18 Ilan Newman. Private vs. common random bits in communication complexity. *Information processing letters*, 39(2):67–71, 1991.

19 Alon Orlitsky. Worst-case interactive communication. i. two messages are almost optimal. *IEEE Transactions on Information Theory*, 36(5):1111–1126, 1990.

20 Alon Orlitsky. Worst-case interactive communication. ii. two messages are not optimal. *IEEE Transactions on Information Theory*, 37(4):995–1005, 1991.

21 David Peleg. *Distributed computing: a locality-sensitive approach*. SIAM, 2000.

22 Jeff M Phillips, Elad Verbin, and Qin Zhang. Lower bounds for number-in-hand multiparty communication complexity, made easy. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 486–501. SIAM, 2012.

23 Alexander A. Razborov. On the distributional complexity of disjointness. *Theoretical Computer Science*, 106(2):385–390, 1992.

24 David P Woodruff and Qin Zhang. Tight bounds for distributed functional monitoring. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pages 941–960. ACM, 2012.

25 David P Woodruff and Qin Zhang. When distributed computation is communication expensive. In *International Symposium on Distributed Computing*, pages 16–30. Springer, 2013.

26 David P Woodruff and Qin Zhang. An optimal lower bound for distinct elements in the message passing model. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 718–733. Society for Industrial and Applied Mathematics, 2014.

27 Andrew Yao. Some complexity questions related to distributed computing. In *Proc. 11th STOC*, pages 209–213, 1979.

28 Andrew C Yao. Lower bounds by probabilistic arguments. In *Foundations of Computer Science, 1983., 24th Annual Symposium on*, pages 420–428. IEEE, 1983.

Which Broadcast Abstraction Captures k-Set Agreement?*

Damien Imbs[1], Achour Mostéfaoui[2], Matthieu Perrin[3], and Michel Raynal[4]

1 **LIF, Université d'Aix-Marseille & CNRS, Marseille, France**
2 **LINA, Université de Nantes, Nantes, France**
3 **IMDEA Software Institute, Pozuelo de Alarcón, Madrid, Spain**
4 **IRISA, Université de Rennes, Rennes, France, and Institut Universitaire de France, Paris, France**

Abstract

It is well-known that consensus (one-set agreement) and total order broadcast are equivalent in asynchronous systems prone to process crash failures. Considering wait-free systems, this article addresses and answers the following question: which is the communication abstraction that "captures" k-set agreement? To this end, it introduces a new broadcast communication abstraction, called k-BO-Broadcast, which restricts the disagreement on the local deliveries of the messages that have been broadcast (1-BO-Broadcast boils down to total order broadcast). Hence, in this context, $k = 1$ is not a special number, but only the first integer in an increasing integer sequence.

This establishes a new "correspondence" between distributed agreement problems and communication abstractions, which enriches our understanding of the relations linking fundamental issues of fault-tolerant distributed computing.

1998 ACM Subject Classification C.2.4 Distributed Systems – distributed applications, network operating systems, D.4.5 Reliability – fault-tolerance, F.1.1 Models of Computation – computability theory

Keywords and phrases Agreement problem, Antichain, Asynchronous system, Communication abstraction, Consensus, Message-passing system, Partially ordered set, Process crash, Read/write object, k-Set agreement, Snapshot object, Wait-free model, Total order broadcast

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.27

1 Introduction

Agreement problems vs communication abstractions. Agreement objects are fundamental in the mastering and understanding of fault-tolerant crash-prone asynchronous distributed systems. The most famous of them is the *consensus* object. This object provides processes with a single operation, denoted propose(), which allows each process to propose a value and decide on (obtain) a value. The properties defining this object are the following: If a process invokes propose() and does not crash, it decides a value (termination); No two processes decide different values (agreement); The decided value was proposed by a process (validity). This object has been generalized by S. Chaudhuri in [7], under the name *k-set agreement*

* This work has been partially supported by the French ANR project 16-CE40-0023-03 DESCARTES, the French ANR project MACARON (anr-13-js02-0002), and the Franco-German DFG-ANR Project 14-CE35-0010-02 DISCMAT.

© Damien Imbs, Achour Mostéfaoui, Matthieu Perrin, and Michel Raynal;
licensed under Creative Commons License CC-BY

Table 1 Associating agreement objects and communication abstractions.

Concurrent object	Communication abstraction
Consensus	Total order broadcast [6]
Snapshot object [1, 2] (and R/W register)	SCD-broadcast [11]
k-set agreement object ($1 \leq k < n$)	k-BO-broadcast (this paper)

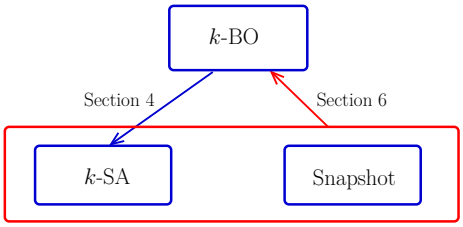


Figure 1 Global picture.

(k-SA), by weakening the agreement property: the processes are allowed to collectively decide up to k different values, i.e., k is the upper bound on the disagreement allowed on the number of different values that can be decided. The smallest value $k = 1$ corresponds to consensus.

On another side, communication abstractions allow processes to exchange data and coordinate, according to some message communication patterns. Numerous communication abstractions have been proposed. Causal message delivery [4, 19], total order broadcast, FIFO broadcast, to cite a few (see the textbooks [3, 15, 16, 17]). In a very interesting way, it appears that some high level communication abstractions "capture" exactly the essence of some agreement objects, see Table 1. The most famous –known for a long time– is the *Total Order broadcast* abstraction which, on one side, allows an easy implementation of a consensus object, and, on an other side, can be implemented from consensus objects. A more recent example is the SCD-Broadcast abstraction that we introduced in [11] (SCD stands for *Set Constrained Delivery*). This communication abstraction allows a very easy implementation of an atomic (Single Writer/Multi Reader or Multi Writer/Multi Reader) snapshot object (as defined in [1]), and can also be implemented from snapshot objects. Hence, as shown in [11], SCD-Broadcast and snapshot objects are the two sides of a same "coin": one side is concurrent object-oriented, the other side is communication-oriented, and none of them is more computationally powerful than the other in asynchronous wait-free systems (where "wait-free" means "prone to any number of process crashes").

Aim and content of the paper. As stressed in [10], Informatics is a science of abstractions. Hence, this paper continues our quest relating communication abstractions and agreement objects. It focuses on k-set agreement in asynchronous wait-free systems. More precisely, the paper introduces the k-BO-broadcast abstraction (BO stands for *Bounded Order*) and shows that it matches k-set agreement in these systems.

k-BO-broadcast is a *Reliable Broadcast* communication abstraction [3, 15, 16, 17], enriched with an additional property which restricts the disagreement on message receptions among the processes. Formally, this property is stated as a constraint on the width of a partial order whose vertices are the messages, and directed edges are defined by local message reception orders. This width is upper bounded by k. For the extreme case $k = 1$, k-BO-broadcast boils down to total order broadcast.

The correspondence linking k-BO-broadcast and k-set agreement, established in the paper, is depicted in Figure 1. The algorithm building k-SA on top of the k-BO-broadcast is surprisingly simple (which is important, as communication abstractions constitute the basic programming layer on top of which distributed applications are built). In the other direction, we show that k-BO-broadcast can be implemented in wait-free systems enriched with k-SA objects and snapshot objects. (Let us recall that snapshot objects do not require additional computability power to be built on top of wait-free read/write systems.) This direction is not as simple as the previous one. It uses an intermediary broadcast communication abstraction, named k-SCD-broadcast, which is a natural and simple generalization of the SCD-broadcast introduced in [11].

Roadmap. The paper is composed of 7 sections. Section 2 presents the basic crash-prone process model, the snapshot object, and k-set agreement. Section 3 defines the k-BO broadcast abstraction and presents a characterization of it. Then, Section 4 presents a simple algorithm implementing k-set agreement on top of the k-BO broadcast abstraction. Section 5 presents another simple algorithm implementing k-BO broadcast on top of the k-SCD-broadcast abstraction. Section 6 presents two algorithms whose combination implements k-SCD-broadcast on top of k-set agreement and snapshot objects. Finally, Section 7 concludes the paper. A global view on the way these constructions are related is presented in Figure 2 of the conclusion.

Due to page limitations, we recommend the reader to refer to the technical report [12] for the proofs of some lemmas and theorems, as well as some considerations about the scope of the results presented here.

2 Process Model, Snapshot, and k-Set Agreement

Process and failure model. The computing model is composed of a set of n asynchronous sequential processes, denoted $p_1, \ldots, p_n$. "Asynchronous" means that each process proceeds at its own speed, which can be arbitrary and always remains unknown to the other processes.

A process may halt prematurely (crash failure), but it executes its local algorithm correctly until its possible crash. It is assumed that up to $(n-1)$ processes may crash in a run (wait-free failure model). A process that crashes in a run is said to be *faulty*. Otherwise, it is *non-faulty*. Hence a faulty process behaves as a non-faulty process until it crashes.

Snapshot object. The snapshot object was introduced in [1, 2]. It is an array $REG[1..n]$ of single-writer/multi-reader atomic read/write registers which provides the processes with two operations, denoted write() and snapshot(). Initially, $REG[1..n] = [\bot, \ldots, \bot]$. The invocation of write(v) by a process p_i assigns v to $REG[i]$, and the invocation of snapshot() by a process p_i returns the value of the full array as if the operation had been executed instantaneously. Expressed in another way, the operations write() and snapshot() are atomic, i.e., in any execution of a snapshot object, its operations write() and snapshot() are linearizable.

If there is no restriction on the number of invocations of write() and snapshot() by each process, the snapshot object is multi-shot. Differently, a one-shot snapshot object is such that each process invokes once each operation, first write() and then snapshot(). The one-shot snapshot objects satisfy a very nice and important property, called *Containment*. Let $reg_i[1..n]$ be the vector obtained by p_i, and $view_i = \{\langle reg_i[x], i\rangle \mid reg_i[x] \neq \bot\}$. For any pair of processes p_i and p_j which respectively obtain $view_i$ and $view_j$, we have $(view_i \subseteq view_j) \vee (view_j \subseteq view_i)$.

Implementations of snapshot objects on top of read/write atomic registers have been proposed (e.g., [1, 2, 13, 14]). The "hardness" to build snapshot objects in read/write systems and associated lower bounds are presented in the survey [9].

k-Set agreement. k-Set agreement (k-SA) was introduced by S. Chaudhuri in [7] (see [18] for a survey of k-set agreement in various contexts). Her aim was to investigate the impact of the maximal number of process failures (t) on the agreement degree (k) allowed to the processes, where the smaller the value of k, the stronger the agreement degree. The maximal agreement degree corresponds to $k = 1$ (consensus).

k-SA is a one-shot agreement problem, which provides the processes with a single operation denoted propose(). When a process p_i invokes propose(v_i), we say that it "proposes value v_i". This operation returns a value v. We then say that the invoking process "decides v", and "v is a decided value". k-SA is defined by the following properties.

- Validity. If a process decides a value v, v was proposed by a process.
- Agreement. At most k different values are decided by the processes.
- Termination. Every non-faulty process that invoked propose() decides a value.

Repeated k-set agreement. This agreement abstraction is a simple generalization of k-set agreement, which aggregates a sequence of k-set agreement instances into a single object. Hence given such an object $RKSA$, a process p_i invokes sequentially $RKSA$.propose(sn_i^1, v_i^1), then $RKSA$.propose(sn_i^2, v_i^2), ..., $RKSA$.propose(sn_i^x, v_i^x), etc, where $sn_i^1, sn_i^2, \ldots, sn_i^x, \ldots$ are increasing (not necessarily consecutive) sequence numbers, and v_i^x is the value proposed by p_i to the instance number sn_i^x. Moreover, the sequences of sequence numbers used by two processes are sub-sequences of 0, 1, 2, etc., but are not necessarily the same sub-sequence. For each sequence number sn, the invocations of $RKSA$.propose(sn, v_i) verify the three properties of k-set agreement.

3 The k-BO-Broadcast Abstraction

Communication operations. The k-Bounded Ordered broadcast (k-BO-Broadcast) abstraction provides the processes with two operations, denoted kbo_broadcast() and kbo_deliver(). The first operation takes a message as input parameter. The second one returns a message to the process that invoked it. Using a classical terminology, when a process invokes kbo_broadcast(m), we say that it "kbo-broadcasts the message m". Similarly, when it invokes kbo_deliver() and obtains a message m, we say that it "kbo-delivers m"; in the operating system parlance, kbo_deliver() can be seen as an *up call* (the messages kbo-delivered are deposited in a buffer, which is accessed by the application according to its own code).

The partial order $\mapsto$. An *antichain* is a subset of a partially ordered set such that any two elements in the subset are incomparable, and a *maximum antichain* is an antichain that has the maximal cardinality among all antichains. The *width* of a partially ordered set is the cardinality of a maximum antichain.

Let $\mapsto_i$ be the local message delivery order at a process p_i defined as follows: $m \mapsto_i m'$ if p_i kbo-delivers the message m before it kbo-delivers the message m'. Let $\mapsto \stackrel{def}{=} \cap_i \mapsto_i$. This relation defines a partially ordered set relation which captures the order on message kbo-deliveries on which all processes agree. In the following, we use the same notation ($\mapsto$) for the relation and the associated partially ordered graph. Let width($\mapsto$) denote the width of the partially ordered graph $\mapsto$.

Properties on the operations. k-BO-broadcast is defined by the following set of properties, where we assume –without loss of generality– that all the messages that are kbo-broadcast are different and every non-faulty process keeps invoking the operation kbo_deliver() forever.

- KBO-Validity. Any message kbo-delivered has been kbo-broadcast by a process.
- KBO-Integrity. A message is kbo-delivered at most once by each process.
- KBO-Bounded. $\mathsf{width}(\mapsto) \leq k$.
- KBO-Termination-1. If a non-faulty process kbo-broadcasts a message m, it terminates its kbo-broadcast invocation and kbo-delivers m.
- KBO-Termination-2. If a process kbo-delivers a message m, every non-faulty process kbo-delivers m.

The reader can easily check that the Validity, Integrity, Termination-1, and Termination-2 properties define *Uniform Reliable Broadcast*.

The KBO-Bounded property, which gives its meaning to k-BO-broadcast, is new. Two processes p_i and p_j *disagree* on the kbo-deliveries of the messages m and m' if p_i kbo-delivers m before m', while p_j kbo-delivers m' before m. Hence we have neither $m \mapsto m'$ nor $m' \mapsto m$.

k-Bounded Order captures the following constraint: processes can disagree on message sets of size at most k. (Said differently, there is no message set ms such that $|ms| > k$ and for each pair of messages $m, m' \in ms$, there are two processes p_i and p_j that disagree on their kbo-delivery order.) Let us consider the following example to illustrate this constraint.

An example. Let m_1, m_2, m_3, m_4, m_5, and m_6, be messages that have been kbo-broadcast by different processes. Let us consider the following sequences of kbo-deliveries by the 3 processes p_1, p_2 and p_3.

- at p_1: m_1, m_2, m_3, m_4, m_5, m_6.
- at p_2: m_2, m_1, m_5, m_3, m_4, m_6.
- at p_3: m_2, m_3, m_1, m_5, m_4, m_6.

The set of messages $\{m_1, m_2\}$ is such that processes disagree on their kbo-delivery order. We have the same for the sets of messages $\{m_1, m_3\}$ and $\{m_4, m_5\}$. It is easy to see that, when considering the set $\{m_1, m_2, m_3, m_4\}$, the message m_4 does not create disagreement with respect to the messages in the set $\{m_1, m_2, m_3\}$.

The reader can check that there is no set of cardinality greater than $k = 2$ such that processes disagree on all the pairs of messages they contain. On the contrary, when looking at the message sets of size ≤ 2, disagreement is allowed, as shown by the sets of messages $\{m_1, m_2\}$, $\{m_1, m_3\}$, and $\{m_4, m_5\}$. In conclusion, these sequences of kbo-deliveries are compatible with 2-BO broadcast.

Let us observe that if two processes disagree on the kbo-deliveries of two messages m and m', these messages define an antichain of size 2. It follows that 1-BO-broadcast is total order broadcast (which is computationally equivalent to Consensus [6]), while $k = n$ imposes no constraint on message deliveries.

Underlying intuition: the non-deterministic k-TO-channel notion. Let us define the notion of a *non-deterministic* k-TO-*channel* as follows (TO stands for *Total Order*). There are k different broadcast channels, each ensuring total order delivery on the messages broadcast through it. The invocation of kbo_broadcast(m) by a process entails a broadcast on one and only one of these broadcast channels, but the channel is selected by an underlying daemon, and the issuing process never knows which channel has been selected for its invocation.

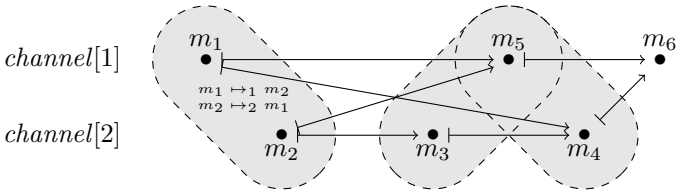
Let us consider the previous example, with $k = 2$. Hence, there are two TO-channels, *channel*[1] and *channel*[2]. As shown by the following figure, they contained the following

```
operation propose(nb, v) is
(1)   kbo_broadcast(⟨nb, v⟩); wait(∃ ⟨nb, x⟩ ∈ decisions_i); return(x).

when a message ⟨sn, x⟩ is kbo-delivered do
(2)   if (⟨sn, −⟩ never added to decisions_i) then decisions_i.insert(⟨sn, x⟩) end if.
```

Algorithm 1 From k-BO-broadcast to repeated k-set agreement.

sequences of messages: $channel[1] = m_1, m_5, m_6$ and $channel[2] = m_2, m_3, m_4$. On this figure, encircled grey areas represent maximum antichains.



It is easy to check that the sequence of messages delivered at any process p_i is a merge of the sequences associated with these two channels.

The assignment of messages to channels is not necessarily unique, it depends on the behavior of the daemon. Considering $k = 3$ and a third channel $channel[3]$, let us observe that the same message kbo-deliveries at p_1, p_2, and p_3, could have been obtained by the following channel selection by the daemon: $channel[1]$ as before, $channel[2] = m3, m4$, and $channel[3] = m2$. Let us observe that, with $k = 3$ and this daemon behavior, the message kbo-delivery $m_3, m_1, m_5, m_4, m_2, m_6$ would also be correct at p_3.

A characterization. The previous non-deterministic k-TO-channel interpretation of k-BO-broadcast is captured by the following characterization theorem.

▶ **Theorem 1.** *A non-deterministic* k-TO-channel *and the* k-BO-broadcast *communication abstraction have the same computational power.*

▶ Remark. It is important to see that k-BO-broadcast and k-TO-channels are not only computability equivalent but are two statements of the very same communication abstraction (there is no way to distinguish them from a process execution point of view).

4 From k-BO-Broadcast to Repeated k-Set Agreement

Algorithm 1 implements repeated k-set agreement in a wait-free system enriched with k-BO-Broadcast. Its simplicity demonstrates the very *high abstraction level* provided by k-BO-Broadcast. All "implementation details" are hidden inside its implementation (which has to be designed only once, and not for each use of k-BO-Broadcast in different contexts). In this sense, k-BO-Broadcast is the abstraction communication which captures the essence of (repeated) k-set agreement.

When a process p_i invokes propose(nb, v), it kbo-broadcasts a message containing the pair $\langle nb, v\rangle$ and waits until a pair $\langle nb, -\rangle$ appears in its local set $decisions_i$ (line 1). Such a pair is added in $decisions_i$ the first time p_i k-BO-delivers a pair $\langle nb, x\rangle$ (line 2). Let us observe that this algorithm is purely based on the k-BO-Broadcast communication abstraction.

▶ **Lemma 2.** *If the invocation of* propose(nb, v) *returns* x *to a process, some process invoked* propose(nb, x).

▶ **Lemma 3.** *If a non-faulty process invokes* propose$(nb, -)$*, it eventually decides a value x such that $\langle nb, x\rangle$ is the first (and only) message $\langle nb, -\rangle$ it kbo-delivers.*

▶ **Lemma 4.** *The set of values returned by the invocations of* propose$(nb, -)$ *contains at most k different values.*

Proof. Let Π_{nb} be the set of processes returning a value from their invocations propose$(nb, -)$. For each $p_i \in \Pi_{nb}$, let $\langle nb, x_i\rangle$ denote the first message $\langle nb, -\rangle$ received by p_i. By Lemma 3, $X_{nb} = \{x_i : p_i \in \Pi_{nb}\}$ is the set of all values returned by the invocations of propose$(nb, -)$.

For any pair x_i and x_j of distinct elements of X_{nb}, we have that p_i kbo-delivered x_i before x_j, and p_j kbo-delivered x_j before x_i. Hence, $\langle nb, x_j\rangle \not\mapsto_i \langle nb, x_i\rangle$ and $\langle nb, x_i\rangle \not\mapsto_j \langle nb, x_j\rangle$, which means $\langle nb, x_i\rangle$ and $\langle nb, x_j\rangle$ are not ordered by $\mapsto$. Therefore, $\{\langle nb, x_i\rangle : p_i \in \Pi_{nb}\}$ is an antichain of $\mapsto$. It then follows from the KBO-Bounded property that $|\{x_i : p_i \in \Pi_{nb}\}| = |\{\langle nb, x_i\rangle : p_i \in \Pi_{nb}\}| \leq k$. ◀

▶ **Theorem 5.** *Algorithm 1 implements repeated k-set agreement in any system model enriched with the communication abstraction k-*BO-broadcast*.*

5 From k-SCD-Broadcast to k-BO-Broadcast

5.1 The intermediary k-SCD-Broadcast abstraction

This communication abstraction is a simple strengthening of the SCD-Broadcast abstraction introduced in [11], where it is shown that SCD-Broadcast and snapshot objects have the same computability power (SCD stands for Set Constrained Delivery).

SCD-Broadcast: definition. SCD-broadcast consists of two operations scd_broadcast() and scd_deliver(). The first operation takes a message to broadcast as input parameter. The second one returns a non-empty set of messages to the process that invoked it. By a slight abuse of language, we say that a process "scd-delivers a message m" when it delivers a message set ms containing m.

SCD-broadcast is defined by the following set of properties, where we assume –without loss of generality– that all the messages that are scd-broadcast are different and that every non-faulty process keeps invoking the operation scd_deliver() forever.

- SCD-Validity. If a process scd-delivers a set containing a message m, then m was scd-broadcast by some process.
- SCD-Integrity. A message is scd-delivered at most once by each process.
- SCD-Ordering. If a process p_i scd-delivers first a message m belonging to a set ms_i and later a message m' belonging to a set $ms'_i \neq ms_i$, then no process scd-delivers first m' in some scd-delivered set ms'_j and later m in some scd-delivered set $ms_j \neq ms'_j$.
- SCD-Termination-1. If a non-faulty process scd-broadcasts a message m, it terminates its scd-broadcast invocation and scd-delivers a message set containing m.
- SCD-Termination-2. If a process scd-delivers a message set containing m, every non-faulty process scd-delivers a message set containing m.

***k*-SCD-Broadcast: definition.** This communication abstraction is SCD-Broadcast strengthened with the following additional property:

- KSCD-Bounded. No set ms kscd-delivered to a process contains more than k messages.

In the following, all properties of k-SCD-broadcast are prefixed by "KSCD".

operation kbo_broadcast(v) **is** kscd_broadcast(m).

when a message set ms **is kscd-delivered do for each** $m \in ms$ **do** kbo_deliver(m) **end for**.

Algorithm 2 From k-SCD-broadcast to k-BO-broadcast.

An example. Like in Section 3, let m_1, m_2, m_3, m_4, m_5, and m_6 be messages that have been kbo-broadcast by different processes. Let us consider the following sequences of message sets k-scd-delivered by the 3 processes p_1, p_2 and p_3.

- at p_1: $\{m_1, m_2\}$, $\{m_3\}$, $\{m_4, m_5\}$, $\{m_6\}$.
- at p_2: $\{m_2\}$, $\{m_1, m_3\}$, $\{m_4, m_5\}$, $\{m_6\}$.
- at p_3: $\{m_1, m_2\}$, $\{m_3, m_5\}$, $\{m_4, m_6\}$.

The processes do not agree on the message sets they k-scd-deliver. For example, p_1 and p_3 k-scd-deliver m_2 in the same set as m_1, whereas p_2 k-scd-deliver m_2 in the same set as m_3. However, at any time, the union of message sets previously k-scd-delivered by any process is part of the following sequence of message sets: $\{m_2\}$, $\{m_1, m_2\}$, $\{m_1, m_2, m_3\}$, $\{m_1, m_2, m_3, m_5\}$, $\{m_1, m_2, m_3, m_4, m_5\}$, $\{m_1, m_2, m_3, m_4, m_5, m_6\}$, which implies the SCD-Ordering property. Moreover, all k-scd-delivered message sets are of size at most $k = 2$.

5.2 From k-SCD-Broadcast to k-BO-Broadcast

Description of the algorithm. Algorithm 2 implements k-BO-Broadcast on top of any system model providing k-SCD-Broadcast. It is an extremely simple self-explanatory algorithm.

▶ **Theorem 6.** *Algorithm 2 implements k-BO-broadcast in any system model enriched with the communication abstraction k-SCD-broadcast.*

Proof. k-BO-Validity, k-BO-Integrity, k-BO-Termination-1 and k-BO-Termination-2 are direct consequences of their homonym SCD-broadcast properties.

To prove the additional k-BO-Bounded property, let us consider a message set ms containing at least $(k+1)$ messages. For each process p_i, let fms_i (resp. lms_i) denote the first (resp. last) set containing a message in ms received by p_i. Thanks to the KSCD-Ordering property, there exists a message $fm \in \cap_i\ fms_i$ and a message $lm \in \cap_i\ lms_i$. (Otherwise, we will have messages m and m' such that $m \in fms_i \wedge m \notin fms_j$ and $m' \notin fms_i \wedge m' \in fms_j$.)

Let ums_i denote the union of all the message sets k-scd-delivered by p_i starting with the set including fms_i and finishing with the set including lms_i. As, for each process p_i, ums_i contains at least the $(k+1)$ messages of ms, we have $fms_i \neq lms_i$. Therefore, we have $fm \neq lm$ and $fm \mapsto lm$. It follows that ms cannot be an antichain of $\mapsto$. Consequently, the antichains of $\mapsto$ cannot contain more than k messages, hence $\mathsf{width}(\mapsto) \leq k$. ◀

6 From Repeated k-Set Agreement and Snapshot to k-SCD-Broadcast

6.1 The K2S abstraction

Definition. The following object, denoted K2S, is used by Algorithm 4 to implement k-SCD-broadcast. "K2S" stands for k-set agreement plus two snapshots. A K2S object provides a single operation k2s_propose(v) that can be invoked once by each process. Its output is a set of sets whose size and elements are constrained by both k-set agreement and the input size (number of different values proposed by processes). The output $sets_i$ of each process p_i is a

```
operation k2s_propose(v) is
(1)  val_i ← KSET.propose(v);
(2)  SNAP1.write(val_i); snap1_i ← SNAP1.snapshot();
(3)  view_i ← {snap1_i[j] | snap1_i[j] ≠ ⊥};
(4)  SNAP2.write(view_i); snap2_i ← SNAP2.snapshot();
(5)  sets_i ← {snap2_i[j] | snap2_i[j] ≠ ⊥};
(6)  return(sets_i).
```

Algorithm 3 An implementation of a K2S object.

non-empty set of non-empty sets, called views and denoted $view$, satisfying the following properties. Let $inputs$ denote the set of different input values proposed by the processes.

- K2S-Validity. $\forall\ i$: $\forall\ view \in sets_i$: $(m \in view) \Rightarrow (m$ was k2s-proposed by a process$)$.
- Set Size. $\forall\ i$: $1 \leq |sets_i| \leq \min(k, |inputs|)$.
- View Size. $\forall\ i$: $\forall\ view \in sets_i$: $(1 \leq |view| \leq \min(k, |inputs|))$.
- Intra-process Inclusion.$\forall\ i$: $\forall\ view1, view2 \in sets_i$: $view1 \subseteq view2 \vee view2 \subseteq view1$.
- Inter-process Inclusion. $\forall\ i, j$: $sets_i \subseteq sets_j \vee sets_j \subseteq sets_i$.
- K2S-Termination. If a non-faulty process p_i invokes k2s_propose(), it returns a set $sets_i$.

Algorithm. Algorithm 3 implements a K2S object. It uses an underlying k-set agreement object $KSET$, and two one-shot snapshot objects denoted $SNAP1$ and $SNAP2$.

- Phase 1 (line 1). When a process p_i invokes k2s_propose(v), it first proposes v to the k-set agreement object, from which it obtains a value val_i (line 1).
- Phase 2 (lines 2-3). Then p_i writes val_i in the first snapshot object $SNAP1$, reads its content, saves it in $snap1_i$, and computes the set of values ($view_i$) that, from its point of view, have been proposed to the k-set agreement object.
- Phase 3 (lines 4-6). Process p_i then writes its view $view_i$ in the second snapshot object $SNAP2$, reads its value, and computes the set of views ($sets_i$) obtained – as far as it knows – by the other processes. Process p_i finally returns this set of views $sets_i$.

▶ **Theorem 7.** *Algorithm 3 satisfies the properties defining a* K2S *object.*

Repeated K2S. In the following we consider a repeated K2S object, denoted KSS. A process p_i invokes KSS.k2s_propose(r, v) where v is the value it proposes to the instance number r. The instance numbers used by each process are increasing (but not necessarily consecutive). Hence, two snapshot objects are associated with every K2S instance, and line 1 of Algorithm 3 becomes $KSET$.propose(r, v).

6.2 From k-Set Agreement and Snapshot to k-SCD-Broadcast

Algorithm 4 builds the k-SCD-Broadcast abstraction on top of k-set agreement and snapshot objects.

Shared objects and local objects.

- The processes cooperate through two concurrent objects: $MEM[1..n]$, a multishot snapshot object, such that $MEM[i]$ contains the set of messages kscd-broadcast by p_i, and a repeated K2S object denoted KSS.
- A process p_i manages two local copies of MEM denoted $mem1_i$ and $mem2_i$, two auxiliary sets $to\_deliver1_i$ and $to\_deliver2_i$, and a set $delivered_i$, which contains all the messages it has locally kscd-delivered; $mem1_i[i]$ is initialized to an empty set.

```
operation kscd_broadcast(m) is
(1)  MEM.write(mem1_i[i] ∪ {m}); mem1_i ← MEM.snapshot();
(2)  to_deliver1_i ← (∪_{1≤j≤n} mem1_i[j]) \ delivered_i; wait(to_deliver1_i ⊆ delivered_i).

background task T is
(3)  repeat forever
(4)    prop_i ← ⊥;
(5)    if (seq_i = ϵ) then mem2_i ← MEM.snapshot();
(6)                        to_deliver2_i ← (∪_{1≤j≤n} mem2_i[j]) \ delivered_i;
(7)                        if (to_deliver2_i ≠ ∅) then prop_i ← a message ∈ to_deliver2_i end if
(8)                 else   prop_i ← a message of the first message set of seq_i
(9)    end if;
(10)   if (prop_i ≠ ⊥)
(11)     then r_i ← |delivered_i|; sets_i ← KSS.k2s_propose(r_i, prop_i); new_seq_i ← ϵ;
(12)          while (sets_i ≠ {∅}) do
(13)                min_set_i ← non-empty set of minimal size in sets_i;
(14)                new_seq_i ← new_seq_i ⊕ min_set_i;
(15)                for each set s ∈ sets_i do sets_i ← (sets_i \ {s}) ∪ {s \ min_set_i} end for
(16)          end while;
(17)          let aux_i = all the messages in the sets of new_seq_i;
(18)          for each set  s ∈ seq_i do s ← s \ aux_i end for;
(19)          seq_i ← new_seq_i ⊕ seq_i; let first_i = head(seq_i); let rest_i = tail(seq_i);
(20)          kscd_deliver(first_i); delivered_i ← delivered_i ∪ first_i; seq_i ← rest_i
(21)   end if
(22) end repeat.
```

Algorithm 4 From k-set agreement and snapshot objects to k-SCD-broadcast (code for p_i).

- r_i denotes the next round number that p_i will execute; $sets_i$ is a local set whose aim is to contain the set of message sets returned by the last invocation of a K2S object.
- Each process p_i manages two sequences of messages sets, both initialized to ϵ (empty sequence), denoted seq_i and new_seq_i; $\mathsf{head}(sq)$ returns the first element of the sequence sq, and $\mathsf{tail}(sq)$ returns sq without its first element; $\oplus$ denotes sequence concatenation. The aim of the local sequence new_seq_i is to contain a sequence of message sets obtained from $sets_i$ (last invocation of a K2S object) such that no message belongs to several sets. As far as seq_i is concerned, we have the following (at line 19 of Algorithm 4). Let $seq_i = ms_1,\ ms_2,\ \cdots, ms_\ell$, where $1 \leq \ell \leq k$ and each ms_x is a message set. This sequence can be decomposed into two (possibly empty) sub-sequences $ms_1,\ ms_2,\ \cdots, ms_y$ and $ms_{y+1} \cdots, ms_\ell$ such that:
  - $ms_1,\ ms_2,\ \cdots, ms_y$ can be in turn decomposed as follows:
    $(ms_1 \cup ms_2 \cup \cdots \cup ms_a), (ms_{a+1} \cup ms_{a+2} \cup \cdots \cup ms_b),\ \cdots, (ms_c \cup \cdots \cup ms_y)$
    where each union set (e.g., $ms_{a+1} \cup ms_{a+2} \cup \cdots \cup ms_b$) is a message set that has been kscd-delivered by some process (some union sets can contain a single message set)[1].
  - For each $x:\ y+1 \leq x \leq \ell:\ m_x$ is a message set whose messages have not yet been kscd-delivered by a process.

Operation kscd_broadcast(). When it invokes $\mathsf{kscd\_broadcast}()$, a process p_i first adds m to the shared memory MEM, which contains all the messages it has already kscd-broadcast (line 1). Then p_i reads atomically the whole content of MEM, which is saved in $mem1_i$ (line 1). Then, p_i computes the set of messages not yet locally kscd-delivered and waits

[1] Let us remark that it is possible that, while a process kscd-delivered the message set $ms = ms_1 \cup ms_2 \cup \cdots \cup ms_a$, another process kscd-delivered the messages in ms in several messages sets, e.g., first the message set $ms_1 \cup ms_2 \cup ms_3$ and then the message set $ms_4 \cup \cdots \cup ms_a$.

until all these messages appear in kscd-delivered message sets (line 2). Let us notice that, it follows from these statements, that a process has kscd-delivered its previous message when it issues its next kscd_broadcast().

Underlying task T. This task is the core of the algorithm. It consists of an infinite loop, which implements a sequence of asynchronous rounds (lines 11-20). Each process p_i executes a sub-sequence of non-necessarily consecutive rounds. Moreover, any two processes do not necessarily execute the same sub-sequence of rounds. The current round of a process p_i is defined by the value of $|delivered_i|$ (number of messages already locally kscd-delivered).

The progress of a process from a round r to its next round $r' > r$ depends on the size of the message set (denoted $first_i$ in the algorithm, line 20) it kscd-delivers at the end of round r ($delivered_i$ becomes then $delivered_i \cup first_i$). The message set $first_i$ depends on the values returned by the K2S object associated with the round r, as explained below.

Underlying task T: proposal computation. (Lines 4-9) Two rounds executed by a process p_i are separated by the local computation of a message value ($prop_i$) that p_i will propose to the next K2S object. This local computation is as follows (lines 5-9), where seq_i (computed at lines 18-20) is a sequence of message sets that, after some "cleaning", are candidates to be locally kscd-delivered. There are two cases.

- Case 1: $seq_i = \epsilon$. In this case (similarly to line 2) p_i computes the set of messages ($to\_deliver2_i$) it sees as kscd-broadcast but not yet locally kscd-delivered (lines 5-6). If $to\_deliver2_i \neq \emptyset$, a message of this set becomes its proposal $prop_i$ for the K2S object associated with the next round (line 7). Otherwise, we have $prop_i = \epsilon$, which, due to the predicate of line 10, entails a new execution of the loop (skipping lines 11-20).
- Case 2: $seq_i \neq \emptyset$. In this case, $prop_i$ is assigned a message of the first set of seq_i (line 8).

Underlying task T: benefiting from a K2S object to kscd-deliver a message set. (Lines 11-20) If a proposal has been previously computed (predicate of line 10), p_i executes its next round, whose number is $r_i = |delivered_i|$. The increase step of $|delivered_i|$ can vary from round to round, and can be any value $\ell \in [1..k]$ (lines 14 and 15). As already indicated, while the round numbers have a global meaning (the same global sequence of rounds is shared by all processes), each process executes a subset of this sequence (as defined by the increasing successive values of $|delivered_i|$). Despite the fact processes skip/execute different rounds, once combined with the use of K2R objects, round numbers allow processes to synchronize in a consistent way. This round synchronization property is captured by Lemmas 11-12.

From an operational point of view, a process starts a round with the invocation KSS.k2s_propose($r_i, prop_i$) where $r_i = |delivered_i|$, which returns a set of message sets $sets_i$ (line 11). Then ("while" loop at lines 12-16), p_i builds from the message sets belonging to $sets_i$ a sequence of message sets new_seq_i, that will be used to extract the next message set kscd-delivered by p_i (lines 17-20). The construction of new_seq_i is as follows. Iteratively, p_i takes the smallest set of $sets_i$ (min_set_i, line 13), adds it at the end of new_seq_i (line 14), and purges all the sets of $sets_i$ from the messages in min_set_i (line 15), so that no message will locally appear in two different messages sets of new_seq_i.

When new_seq_i is built, p_i first purges all the sets of the sequence seq_i from the messages in new_seq_i (lines 17-18), and adds then new_seq_i at the front of seq_i (line 19). Finally, p_i kscd-delivers the first message set of seq_i, and updates $delivered_i$ and seq_i (lines 20).

6.3 Proof of the algorithm

▶ **Lemma 8.** *A message set kscd-delivered (line* 20*) contains at most k messages.*

▶ **Lemma 9.** *If a process kscd-delivers a message set containing a message m, m was kscd-broadcast by a process.*

Notations.

- $msg\_set_i(r)$ = message set kscd-delivered by process p_i at round r if p_i participated in it, and $\emptyset$ otherwise.
- $seq_i(r)$ = value of seq_i at the end of the last round $r' \leq r$ in which p_i participated.
- $msgs_i(r, r')$ = set of messages contained in message sets kscd-delivered by p_i between rounds r (included) and $r' > r$ (not included), i.e. $msgs_i(r, r') = \bigcup_{r \leq r'' < r'} msg\_set_i(r'')$.
- $KSS(r)$ = K2S instance accessed by KSS.k2s_propose$(r, -)$ (line 11).
- $sets_i(r)$ = set of message sets obtained by p_i from $KSS[r]$.

▶ **Lemma 10.** *Let p_i and p_j be two processes that terminate round r, with $|msg\_set_i(r)| \leq |msg\_set_j(r)|$. Then* (i) *$msg\_set_i(r) \subseteq msg\_set_j(r)$, and* (ii) *there is a prefix $pref_i$ of $seq_i(r)$ such that $msg\_set_j(r) = msg\_set_i(r) \cup (\bigcup_{msg\_set \in pref_i} msg\_set)$.*

Proof. Let p_i and p_j be two processes that kscd-deliver the message sets $msg\_set_i(r)$ and $msg\_set_j(r)$, respectively, these sets being such that $|msg\_set_i(r)| \leq |msg\_set_j(r)|$. Let us observe that, as both p_i and p_j invoked KSS.k2s_propose$(r, -)$ (lines 11 and 20), we have $sets_i(r) \subseteq sets_j(r)$ or $sets_j(r) \subseteq sets_i(r)$ (Inter-process Inclusion).

As $|msg\_set_i(r)| \leq |msg\_set_j(r)|$, it follows from the Inter-process and Intra-process inclusion properties of $KSS(r)$, and the definition of $msg\_set_i(r) = first_i = min\_set_i \in sets_i(r)$, and $msg\_set_j(r) = first_j = min\_set_j \in sets_j(r) \subseteq sets_i(r)$, that $msg\_set_i(r) \subseteq msg\_set_j(r)$, which completes the proof of (i).

As far as (ii) is concerned, we have the following. If $msg\_set_i(r) = msg\_set_j(r)$, we have $pref_i = \epsilon$ and the lemma follows. So, let us assume $msg\_set_i(r) \subsetneq msg\_set_j(r)$. As $msg\_set_i(r)$ is the smallest message set of $sets_i(r)$ (lines 13-14 and 19-20), and $msg\_set_j(r)$ is the smallest message set of $sets_j(r)$, it follows that $sets_j(r) \subset sets_i(r)$. The property $msg\_set_j(r) = msg\_set_i(r) \cup (\bigcup_{msg\_set \in pref_i} msg\_set)$ follows then from the following observation. Let $sets_i(r) = \{s_1, s_2, ..., s_\ell\}$, where $\ell \leq k$ and $s_1 \subsetneq s_2 \subsetneq \cdots \subsetneq s_\ell$. As $sets_j(r) \subset sets_i(r)$, one s_x is $msg\_set_j(r)$. It follows that the union of the sets min_set_i computed by p_i in the while loop of round r (lines 13-15) eventually includes all the messages of $msg\_set_j(r)$, from which we conclude that there is a prefix $pref_i$ of $seq_i(r)$ (lines 12-19, namely a prefix of the sequence new_seq_i, which is defined from the sequence of the sets min_set_i), such that $msg\_set_j(r) = msg\_set_i(r) \cup (\bigcup_{msg\_set \in pref_i} msg\_set)$, which completes the proof of the lemma. ◀

Lemmas 11-12 capture the global message set delivery synchronization among the processes.

▶ **Lemma 11.** *Let p_i and p_j be two processes that terminate round $r' \geq r + |msg\_set_j(r)|$, and are such that $|msg\_set_i(r)| \leq |msg\_set_j(r)|$. Then* (i) *$msgs_i(r, r + |msg\_set_j(r)|) = msgs_j(r, r + |msg\_set_j(r)|)$, and* (ii) *$p_i$ and p_j will both participate in round $r + |msg\_set_j(r)|$.*

Proof. If $|msg\_set_i(r)| = |msg\_set_j(r)| = \alpha$, both p_i and p_j are such that $|delivered_i| = |delivered_j| = r + \alpha$ when they terminate round r. Consequently, they both proceed from round r to round $r+\alpha$, thereby skipping the rounds from $r+1$ until $r+\alpha-1$. We then have (i) $msgs_i(r, r + |msg\_set_j(r)|) = msg\_set_i(r) = msg\_set_j(r) = msgs_j(r, r + |msg\_set_j(r)|)$, (ii) both p_i and p_j will participate in round $r + |msg\_set_j(r)|$, and the lemma follows.

Hence, let us consider that $|msg\_set_i(r)| = \alpha < |msg\_set_j(r)| = \alpha + \beta$. The next round executed by p_i will be the round $r + \alpha$, while the next round executed by p_j will be the round $r + \alpha + \beta$. Moreover, to simplify and without loss of generality, let us assume that $msg\_set_i(r)$ (resp. $msg\_set_j(r)$) is the smallest (resp. second smallest) message set in the sets of message sets $sets$ output by $KSS(r)$.

According to Lemma 10, after round r, the first element of seq_i is $msg\_set_j(r) \setminus msg\_set_i(r)$. This also applies to any other process that delivered $msg\_set_i(r)$ at round r. At round $r+\alpha$, all these processes will then propose a message in $msg\_set_j(r) \setminus msg\_set_i(r)$. Because of the K2S-Validity property of $KSS(r + \alpha)$, all these processes will then deliver a subset of $msg\_set_j(r) \setminus msg\_set_i(r)$. For the same reason, until round $r + \alpha + \beta$, no process will propose a message not in $msg\_set_j(r) \setminus msg\_set_i(r)$. At round $r + \alpha + \beta$, they will then have delivered all the messages in $msg\_set_j(r) \setminus msg\_set_i(r)$, and they will participate in round $r + \alpha + \beta$, from which the lemma follows. ◀

▶ **Lemma 12.** *Let r be a round in which all the non-faulty processes participate. There is a round r' with $r < r' \leq r + k$ in which all non-faulty processes participate and such that, for any pair of non-faulty processes p_i and p_j, we have $msgs_i(r, r') = msgs_j(r, r')$.*

Proof. As initially $\forall i : |delivered_i| = 0$, KSS.k2s_propose$(0, -)$ is invoked by all non-crashed processes. We prove that there is a round $r \in [1..k]$ in which all the non-crashed processes participate, and for any pair of them p_i and p_j, we have $msgs_i(0, r) = msgs_j(0, r)$. This constitutes the base case of an induction. Then, the same reasoning can be used to show that if the non-faulty processes participate in a round r, there is a round r' with $r < r' \leq r+k$ and such that, for any pair of non-faulty processes p_i and p_j, we have $msgs_i(r, r') = msgs_j(r, r')$.

Let us consider any two p_i and p_j that terminate round 0. Moreover, without loss of generality, let us assume that, among the sets of message sets output by $KSS(0)$, $sets_i(0)$ is the greatest and $sets_j(0)$ is the smallest. It follows from the Inter-process inclusion property that $sets_j(0) \subseteq sets_i(0)$, and from line 13 plus the Intra-process inclusion property that $msg\_set_i(0) \subseteq msg\_set_j(0)$. Hence, $|msg\_set_i(0)| \leq |msg\_set_j(0)|$. Moreover, due to the View size property of $KSS(0)$ we have $|msg\_set_i(0)| \leq |msg\_set_j(0)| = r \leq k$. Applying Lemma 11, we have $msg_i(0, 0 + r) = msg_j(0, 0 + r)$, which concludes the proof. ◀

▶ **Lemma 13.** *If a process p_i kscd-delivers first a message m belonging to a set ms_i and later a message m' belonging to a set $ms'_i \neq ms_i$, then no process kscd-delivers first m' in some kscd-delivered set ms'_j and later m in some kscd-delivered set $ms_j \neq ms'_j$.*

Proof. Let us first note that, at each process, the kscd-delivery of message sets establishes a partial order on messages. Given a process p_i, let $\rightarrow_i$ be the partial order defined as follows[2]: $m \rightarrow_i m'$ if p_i kscd-delivered first a message set ms_i including m, and later kscd-delivered a message set ms'_i including m'. Hence, if m and m' were kscd-delivered in the same message set by p_i, we have $m \not\rightarrow_i m'$ and $m' \not\rightarrow_i m$.

Let us also note that, along the execution of a process p_i, the partial order $\rightarrow_i$ can only be extended, i.e. if $m \rightarrow_i m'$ at time t, we cannot have $m \not\rightarrow_i m'$ at time $t' > t$. This, along with the fact that a faulty process executes its algorithm correctly until it crashes, allows us to consider, in the context of this proof, that p_i and p_j are non-faulty.

In order to prove the lemma, we then have to show that the partial orders $\rightarrow_i$ and $\rightarrow_j$ are compatible, i.e. for any two messages m and m', $(m \rightarrow_i m') \Rightarrow (m' \not\rightarrow_j m)$ and $(m \rightarrow_j m') \Rightarrow (m' \not\rightarrow_i m)$.

[2] This definition is similar to the definition of $\mapsto_i$ given in Section 3 devoted to kBO-broadcast.

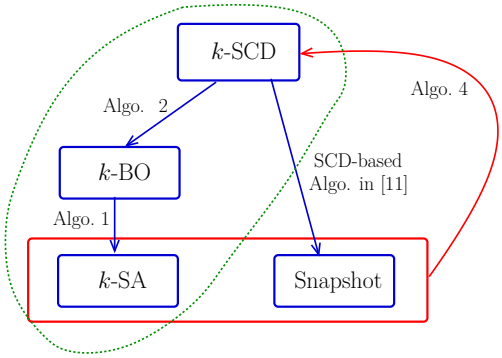


Figure 2 Detailing the global view.

According to Lemma 12, for each round r in which all processes participate, there is a round $r' > r$ in which all processes participate. Moreover, for any two non-faulty process p_i and p_j, we have $msgs_i(r, r') = msgs_j(r, r')$. For any such round r, we then have that if p_i delivered message m strictly before round r and delivered m' at round r or afterwards, we have both $(m \rightarrow_i m')$ and $(m' \rightarrow_j m)$. We will then consider the messages delivered between two such rounds r and r'.

Without loss of generality, suppose that the message set kscd-delivered by p_i at round r is smaller than, or equal to, the message set kscd-delivered by p_j at the same round, i.e. $|msg\_set_i(r)| \leq |msg\_set_j(r)|$. It follows from Lemma 11 that $msgs_i(r, |msg\_set_j(r)|) = msgs_j(r, |msg\_set_j(r)|)$. Moreover, as all the messages in $msg\_set_j(r)$ were kscd-delivered by p_j in a single set, they are all incomparable when considering $\rightarrow_j$. The partial orders $\rightarrow_i$ and $\rightarrow_j$, when restricted to the messages in $msg\_set_j(r)$, are thus compatible.

According to Lemma 11, p_i and p_j will both participate in round $r+\alpha = r+|msg\_set_j(r)|$. If $r+\alpha = r'$, the lemma follows. Otherwise, let $\beta = \max(|msg\_set_i(r+\alpha)|, |msg\_set_j(r+\alpha)|)$. The previous reasoning, again due to Lemma 11, can then be applied again to the messages in $msgs_i(r+\alpha, r+\alpha+\beta) = msgs_j(r+\alpha, r+\alpha+\beta)$, and p_i and p_j will both participate in round $r+\alpha+\beta$. This can be repeated until round r', showing that the partial orders $\rightarrow_i$ and $\rightarrow_j$ are compatible, which concludes the proof of the lemma. ◀

▶ **Lemma 14.** *No message m is kscd-delivered twice by a process p_i.*

▶ **Lemma 15.** *Let m be a message that has been deposited into MEM. Eventually, m is kscd-delivered (at least) by the non-faulty processes.*

▶ **Lemma 16.** *If a process kscd-delivers a message m, every non-faulty process kscd-delivers a message set containing m.*

▶ **Lemma 17.** *If a non-faulty process p_i kscd-broadcasts a message m, it terminates its kscd-broadcast invocation and kscd-delivers a message set containing m.*

▶ **Theorem 18.** *Algorithm* 4 *implements* KSCD-broadcast *from k-set agreement and snapshot objects.*

7 Conclusion

This paper has introduced a new communication abstraction, denoted k-BO-broadcast, which captures k-set agreement in asynchronous crash-prone wait-free systems. In the case $k = 1$ (consensus is 1-set agreement), 1-BO-broadcast boils down to Total Order broadcast.

"Capture" means here that (i) k-set agreement can be solved in any system model providing the k-BO-broadcast abstraction, and (ii) k-BO-broadcast can be implemented from k-set agreement in any system model providing snapshot objects. It follows that, when considering asynchronous crash-prone wait-free systems where basic communication is through a set of atomic read/write, or the asynchronous message-passing system enriched with the failure detector Σ [5, 8], k-BO-broadcast and k-set agreement are the two faces of the same coin: one is its communication-oriented face while the other one is its agreement-oriented face.

From a technical point of view, a complete picture of the content of the paper appears in Figure 2. It is important to notice that the two constructions inside the dotted curve are free from concurrent objects: each rests only on an underlying (appropriate) communication abstraction.

References

1 Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993.

2 James H. Anderson. Multi-writer composite registers. *Distributed Computing*, 7(4):175–195, 1994.

3 Hagit Attiya and Jennifer L. Welch. *Distributed computing: fundamentals, simulations and advanced topics*. Wiley-Interscience, 2004.

4 Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, 1987.

5 François Bonnet and Michel Raynal. A simple proof of the necessity of the failure detector sigma to implement an atomic register in asynchronous message-passing systems. *Inf. Process. Lett.*, 110(4):153–157, 2010.

6 Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

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

8 Carole Delporte-Gallet, Hugues Fauconnier, and Rachid Guerraoui. Tight failure detection bounds on atomic object implementations. *J. ACM*, 57(4):22:1–22:32, 2010.

9 Faith Ellen. How hard is it to take a snapshot? In *SOFSEM 2005: Theory and Practice of Computer Science, 31st Conference on Current Trends in Theory and Practice of Computer Science, Liptovský Ján, Slovakia, January 22-28, 2005, Proceedings*, pages 28–37, 2005.

10 Michael J. Fischer and Michael Merritt. Appraising two decades of distributed computing theory research. *Distributed Computing*, 16(2-3):239–247, 2003.

11 Damien Imbs, Achour Mostéfaoui, Matthieu Perrin, and Michel Raynal. Set-constrained delivery broadcast: Definition, abstraction power, and computability limits. *CoRR*, abs/1706.05267, 2017. URL: `http://arxiv.org/abs/1706.05267`.

12 Damien Imbs, Achour Mostéfaoui, Matthieu Perrin, and Michel Raynal. Which broadcast abstraction captures k-set agreement? *CoRR*, abs/1705.04835, 2017. URL: `http://arxiv.org/abs/1705.04835`.

13 Damien Imbs and Michel Raynal. Help when needed, but no more: Efficient read/write partial snapshot. *J. Parallel Distrib. Comput.*, 72(1):1–12, 2012.

14 Michiko Inoue, Toshimitsu Masuzawa, Wei Chen, and Nobuki Tokura. Linear-time snapshot using multi-writer multi-reader registers. In *Distributed Algorithms: 8th International Workshop, WDAG '1994 Terschelling, The Netherlands, September 29 – October 1, 1994 Proceedings*, pages 130–140. Springer Berlin Heidelberg, 1994.

15 Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

16 Michel Raynal. *Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2010.

17 Michel Raynal. *Distributed Algorithms for Message-Passing Systems*. Springer, 2013.

18 Michel Raynal. Set agreement. In *Encyclopedia of Algorithms*, pages 1956–1959. 2016.

19 Michel Raynal, André Schiper, and Sam Toueg. The causal ordering abstraction and a simple way to implement it. *Inf. Process. Lett.*, 39(6):343–350, 1991.

Extending Hardware Transactional Memory Capacity via Rollback-Only Transactions and Suspend/Resume*†

Shady Issa[1], Pascal Felber[2], Alexander Matveev[‡3], and Paolo Romano[4]

1 **INESC-ID / Instituto Superior Técnico, University of Lisbon, Portugal**
2 **University of Neuchatel, Switzerland**
3 **MIT, Cambridge, USA**
4 **INESC-ID / Instituto Superior Técnico, University of Lisbon, Portugal**

Abstract

Transactional memory (TM) aims at simplifying concurrent programming via the familiar abstraction of atomic transactions. Recently, Intel and IBM have integrated hardware based TM (HTM) implementations in commodity processors, paving the way for the mainstream adoption of the TM paradigm. Yet, existing HTM implementations suffer from a crucial limitation, which hampers the adoption of HTM as a general technique for regulating concurrent access to shared memory: the inability to execute transactions whose working sets exceed the capacity of CPU caches. In this paper we propose P8TM, a novel approach that mitigates this limitation on IBM's POWER8 architecture by leveraging a key combination of techniques: uninstrumented read-only transactions, Rollback Only Transaction-based update transactions, HTM-friendly (software-based) read-set tracking, and self-tuning. P8TM can dynamically switch between different execution modes to best adapt to the nature of the transactions and the experienced abort patterns. In-depth evaluation with several benchmarks indicates that P8TM can achieve striking performance gains in workloads that stress the capacity limitations of HTM, while achieving performance on par with HTM even in unfavourable workloads.

1998 ACM Subject Classification D.1.3 Concurrent Programming

Keywords and phrases hardware transactional memory, self tuning

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.28

1 Introduction

Transactional memory (TM) has emerged as a promising paradigm that aims at simplifying concurrent programming by bringing the familiar abstraction of atomic and isolated transactions to the domain of parallel computing. Unlike when using locks to synchronize access to shared data or code portions, with TM programmers need only to specify *what* is synchronized and not *how* synchronization should be performed. This results in simpler designs that are easier to write, reason about, maintain, and compose [4].

* This work was supported by Portuguese funds through Fundação para a Ciência e Tecnologia via projects UID/CEC/50021/2013 and PTDC/EEISCR/1743/2014.
† The extended version [18] can be found at `http://www.inesc-id.pt/ficheiros/publicacoes/12747.pdf`
‡ Alexander Matveev was supported by the NSF under grants IIS-1447786 and CCF- 1563880


© Shady Issa, Pascal Felber, Alexander Matveev, and Paolo Romano;
licensed under Creative Commons License CC-BY

Over the last years, the relevance of TM has been growing along with the maturity of available supports for this new paradigm, both in terms of integration at the programming language as well as at the architectural level. On the front of integration with programming languages, a recent milestone has been the official integration of TM in mainstream languages, such as C/C++ [2]. On the architecture's side, the integration of hardware supports in Intel's and IBM's processors, a technology that goes under the name of hardware transactional memory (HTM), has represented a major breakthrough, thanks to enticing performance gains that such an approach can, at least potentially, enable [15, 17, 24].

Existing hardware implementations share various architectural choices, although they do come in different flavours [19, 22, 25]. The key common trait of current HTM systems is their best effort nature: current implementations maintain transactional metadata (e.g., memory addresses read/written by a transaction) in the processor's cache and rely on relatively non-intrusive modification to the pre-existing cache coherency protocol to detect conflict among concurrent transactions. Due to the inherently limited nature of processor caches, current HTM implementations impose stringent limitations on the number of memory accesses that can be performed within a transaction,[1] hence providing no progress guarantee even for transactions that run in absence of concurrency. As such, HTM requires a fallback synchronization mechanism (also called *fallback path*), which is typically implemented via a pessimistic scheme based on a single global lock.

Despite these common grounds, current HTM implementations have also several relevant differences. Besides internal architectural choices (e.g., where and how in the cache hierarchy transactional metadata are maintained), Intel's and IBM's implementations differ notably by the programming interfaces they expose. In particular, IBM POWER8's HTM implementation extends the conventional transactional demarcation API (to start, commit and abort transactions) with two additional, unique features [5]:

- *Suspend/resume*: the ability to suspend and resume a transaction, allowing, between the suspend and resume calls, for the execution of instructions/memory accesses that escape from the transactional context.
- *Rollback-only transaction (ROT)*: a lightweight form of transaction that has lower overhead than regular transactions but also weaker semantics. In particular ROTs avoid tracking load operations, i.e., they are not isolated, but still ensure the atomicity of the stores issued by a transaction, which appear to be all executed or not executed at all.

In this work we present POWER8 TM (P8TM), a novel TM that exploits these two specific features of POWER8's HTM implementation in order to overcome (or at least mitigate) what is, arguably, the key limitation stemming from the best-effort nature of existing HTM systems: the inability to execute transactions whose working sets exceed the capacity of CPU caches. P8TM pursues this objective via an innovative hardware-software co-design that leverages several novel techniques, which we overview in the following:

- **Uninstrumented read-only transactions (UROs).** P8TM executes read-only transactions outside of the scope of hardware transactions, hence sparing them from spurious aborts and capacity limitations, while still allowing them to execute concurrently with update transactions. This result is achieved by exploiting the POWER8's suspend/resume mechanism to implement a RCU-like quiescence scheme that shelters UROs from observing inconsistent snapshots that reflect the commit events of concurrent update transactions.

[1] The list of restrictions is actually longer, including the lack of support for system calls and other non-undoable instructions, context switches and ring transitions.

- **ROT-based update transactions.** In typical TM workloads the read/write ratio tends to follow the 80/20 rule, i.e., transactified methods tend to have large read-sets and much smaller write sets [12]. This observation led us to develop a novel concurrency control scheme based on a novel hardware-software co-design: it combines the hardware-based ROT abstraction—which tracks only transactions' write sets, but not their read-sets, and, as such, does not guarantee isolation—with software based techniques aimed to preserve correctness in presence of concurrently executing ROTs, UROs, and plain HTM transactions. Specifically, P8TM relies on a novel mechanism, which we called Touch-To-Validate (T2V), to execute concurrent ROTs safely. T2V relies on a lightweight software instrumentation of reads within ROTs' and a hardware aided validation mechanism of the read-set during the commit phase.
- **HTM-friendly (software-based) read-set tracking.** A key challenge that we had to tackle while implementing P8TM was to develop a "HTM-friendly" software-based read-set tracking mechanism. In fact, all the memory writes issued from within a ROT, including those needed to track the read-set, are transparently tracked in hardware. As such, the read-set tracking mechanism can consume cache capacity that could be otherwise used to accommodate application-level writes. P8TM integrates two read-set tracking mechanisms that explore different trade-offs between space and time efficiency.
- **Self-tuning.** To ensure robust performance in a broad range of workloads, P8TM integrates a lightweight reinforcement learning mechanism (based on the UCB algorithm [21]) that automates the decision of whether: i) to use upfront ROTs and UROs, avoiding at all to use HTM; ii) to first attempt transactions in HTM, and then fallback to ROTs/UROs in case of capacity exceptions; iii) to completely switch off ROTs/UROs, and use only HTM.

We evaluated P8TM by means of an extensive study that encompasses synthetic micro-benchmarks and the benchmarks in the Stamp suite [7]. The results of our study show that P8TM can achieve up $\sim$5$\times$ throughput gains with respect to plain HTM and extend its capacity by more than one order of magnitude, while remaining competitive even in unfavourable workloads.

2 Related Work

Since the introduction of HTM support in mainstream commercial processors by Intel and IBM, several experimental studies have aimed to characterize their performance and limitations [15, 17, 24]. An important conclusion reached by these studies is that HTM's performance excels with workloads that fit the hardware capacity limitations. Unfortunately, though, HTM's performance and scalability can be severely hampered in workloads that contain even a small percentage of transactions that do exceed the hardware's capacity. This is due to the need to execute such transactions using a sequential fallback mechanism based on a single global lock (SGL), which causes the immediate abort of any concurrent hardware transactions and prevents any form of parallelism.

Hybrid TM [9, 20] (HyTM) attempts to address this issue by falling back to software-based TM (STM) implementations when transactions cannot successfully execute in hardware. Hybrid NoRec (Hy-NoRec) is probably one of the most popular and effective HyTM designs proposed in the literature. Hy-NoRec [8] falls back on using the NoRec STM, which lends itself naturally to serve as fallback for HTM. In fact, NoRec uses a single versioned lock for synchronizing (software) transactions. Synchronization between HTM and STM can hence be attained easily, by having HTM transactions update the versioned lock used by NoRec.

Unfortunately, the coupling via the versioned lock introduces additional overheads on both the HTM and STM side, and can induce spurious aborts of HTM transactions.

Recently, RHyNoRec [23] proposed to decompose a transaction running on the fallback path into multiple hardware transactions: a read-only prefix and a single post-fix that encompasses all the transaction's writes, with regular NoRec shared operations in between. This can reduce the false aborts that would otherwise affect hardware transactions in HyNoRec. Unfortunately, though, this approach is only viable if the transaction's postfix, which may potentially encompass a large number of reads, does fit in hardware. Further, the technique used to enforce atomicity between the read-only and the remaining reads relies on fully instrumenting every read within the prefix hardware transaction, this utterly limits the capacity—and consequently the practicality—of these transactions. Unlike RHyNoRec, P8TM can execute read-only transactions of arbitrary length in a fully uninstrumented way. Further, the T2V mechanism employed by P8TM to validate update transactions relies on a much lighter and efficient read-set tracking and validation schemes that can even further increase the capacity of transactions.

Our work is also related to the literature aimed to enhance HTM's performance by optimizing the management of the SGL fallback path. A simple, yet effective optimization, which we include in P8TM, is to avoid the, so called, *lemming effect* [11] by ensuring that the SGL is free before starting a hardware transaction. An alternative solution to the same problem is the use of an auxiliary lock [3]. In our experience, these two solutions provide equivalent performance, so we opted to integrate in P8TM the former, simpler, approach. Herihly et al. [6] suggested lazy subscription of the SGL in order to decrease the vulnerability window of HTM transactions. However, this approach was shown to be unsafe in subtle scenarios that are hard to fix using automatic compiler-based techniques [10].

P8TM integrates a self-tuning approach that shares a common theoretical framework (the UCB reinforcement learning algorithm [21]) with Tuner [13]. However, Tuner addresses an orthogonal self-tuning problem to the one we tackled in P8TM: Tuner exploits UCB to identify the optimal retry policy before falling back to the SGL path upon a capacity exception; in P8TM, conversely, UCB is to determine which synchronization to use (e.g., ROTs/UROs vs. plain HTM). Another recent work that makes extensive use of self-techniques to optimize HTM's performance is SEER [14]. Just like Tuner, SEER addresses an orthogonal problem—defining a scheduling policy that seeks an optimal trade-off between throughput and contention probability—and could, indeed, be combined with P8TM.

Finally, P8TM builds on and extends on HERWL[16], where we introduced the idea of using POWER8's suspend-resume and ROT facilities to elide read-write locks. Besides targeting a different application domain (transactional programs vs. lock elision), P8TM integrates a set of novel techniques. Unlike HERWL, P8TM supports the concurrent execution of update transactions in ROTs. Achieving this result implied introducing a novel concurrency control mechanism (which we named Touch-To-Validate). Additionally, P8TM integrates self-tuning techniques that ensure robust performance also in unfavourable workloads.

3 Background on POWER8's HTM

This section provides background on POWER8's HTM system, which is relevant to the operation of P8TM. Analogously to other HTM implementations, POWER8 provides an API to begin, commit and abort transactions. When programs request to start a transaction, a *started* code is placed in the, so called, status buffer. If, later, the transaction aborts, the program counter jumps back to just after the instruction used to begin the transaction.

Hence, in order to distinguish whether a transaction has just started, or has undergone an abort, programs must test the status code returned after beginning the transaction.

POWER8 detects conflicts with granularity of a cache line. The transaction capacity (64 cache lines) in POWER8 is bound by a 8KB cache, called TMCAM, which stores the addresses of the cache lines read or written within the transaction.

As mentioned, in addition to HTM transactions, POWER8 also supports Rollback-Only Transactions (ROT). The main difference being that in ROTs, only the writes are tracked in the TMCAM, giving virtually infinite read-set capacity. Reads performed by ROTs are essentially treated as non transactional reads. From this point on, whenever we use the term *transaction*, we refer to a plain HTM transaction.

Both transactions and ROTs detect conflict eagerly, i.e., they are aborted as soon as they incur a conflict. The only exception is when they incur a conflict while in suspend mode: in this case, they abort only once they resume. Finally, P8TM exploits how POWER8 manages conflicts that arise between non-transactional code and transactions/ROTs, i.e., if a transaction/ROT issues a write on X and, before it commits, a non-transactional read/write is issued on X, the transaction/ROT is immediately aborted by the hardware.

4 The P8TM Algorithm

This section describes P8TM (*POWER8 Transactional Memory*). We start by overviewing the algorithm. Next, we detail its operation and present several optimizations.

4.1 Overview

The key challenge in designing execution paths that can run concurrently with HTM is efficiency: it is hard to provide a software-based path that executes concurrently with the HTM path, while preserving correctness and speed. The main problem is that the protocol must make the hardware aware of concurrent software memory reads and writes, which requires to introduce expensive tracking mechanisms in the HTM path.

P8TM tackles this issue by exploiting two unique features of the IBM POWER8 architecture:

(1) suspend/resume for hardware transactions, and

(2) ROTs.

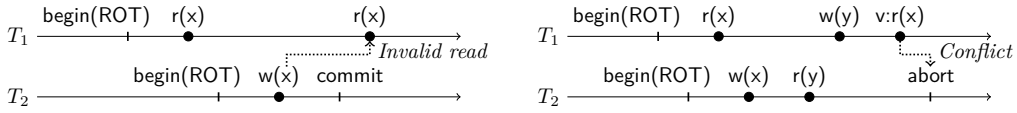
P8TM combines these new hardware features with an RCU-like quiescence scheme in a way that avoids the need to track reads in hardware. This can in particular reduce the likelihood of capacity aborts that would otherwise affect transactions that perform a large number of reads.

The key idea is to provide two novel execution paths alongside the HTM path:

(i) a, so called, *ROT path*, which executes write transactions that do not fit in HTM as ROTs, and

(ii) a, so called, *URO path*, which executes read-only transactions without any instrumentation.

Transactions and ROTs exploit the speculative hardware supports to hide writes from concurrent reads. This allows to cope with read-write conflicts that occur during ROTs/UROs, but it does not cover read-write conflicts that occur after the commit of an update transaction. For this purpose, before a write transaction commits, either as a transaction or a ROT, it first suspends itself and then executes a quiescence mechanism that waits for the completion of currently executing ROTs/URO transactions. In addition to that, in case of ROTs, it further executes an original *touch-based validation* step, which is described next, before



(a) ROTs do not track reads and may observe different values when reading the same variable multiple times.

(b) By re-reading x during *rot-rset* validation at commit time (denoted by v:r), T_1 forces an abort of T_2 that has updated x in the meantime.

Figure 1 Operation scenarios.

resuming and committing. This process of "suspending and waiting" ensures that the writes of an update transaction will be committed only if they do not target/overwrite any memory location that was previously read by any concurrent ROT/URO transaction.

4.2 Touch-based Validation

Touch-To-Validate (T2V) is a core mechanism of our algorithm that enables safe and concurrent execution of ROTs. Indeed, ROTs do not track read accesses within the transaction, therefore it is unsafe to execute them concurrently, as they are not serializable.

Consider the example shown in Figure 1a. Thread T_1 starts a ROT and reads x. At this time, thread T_2 starts a concurrent ROT, writes a new value to x, and commits. As ROTs do not track reads, the ROT of T_1 does not get aborted and can read inconsistent values (e.g., the new value of x), hence yielding non-serializable histories. To avoid such scenarios T2V leverages two key mechanisms that couple:

(i) software-based tracking of read accesses; and
(ii) hardware- and software-based read-set validation during the commit phase.

For the sake of clarity, let us assume that threads only execute ROTs—we will consider other execution modes later. A thread can be in one of three states: *inactive*, *active*, and *committing*. A thread that executes non-transactional code is inactive. When the thread starts a ROT, it enters the active phase and starts tracking, in software, each read access to shared variables by logging the associated memory address in a special data structure called *rot-rset*. Finally, when the thread finishes executing its transaction, it enters the committing phase. At this point, it has to wait for concurrent threads that are in the active phase to either enter the commit phase or become inactive (upon abort). Thereafter, the committing thread traverses its *rot-rset* and re-reads each address before eventually committing.

The goal of this validation step is to "touch" each previously read memory location in order to abort any concurrent ROT that might have written to the same address. For example, in Figure 1b, T_1 re-reads x during *rot-rset* validation. At that time, T_2 has concurrently updated x but has not yet committed, and it will therefore abort (remember that ROTs track and detect conflicts for writes). This allows T_1 to proceed without breaking consistency: indeed, ROTs buffer their updates until commit and hence the new value of x written by T_2 is not visible to T_1. Note that adding a simple quiescence phase before commit, without performing the *rot-rset* validation, cannot solve the problem in this scenario.

The originality of the T2V mechanism is that the ROT does not use read-set validation for verifying that its read-set is consistent, as many STM algorithms do, but to trigger hardware conflicts detection mechanisms. This also means that the values read during *rot-rset* validation are irrelevant and ignored by the algorithm.

Algorithm 1 P8TM: ROT path only algorithm.

```
1:  Shared variables:
2:      status[N] ← {⊥, ⊥, ..., ⊥}                          ▷ One per thread
3:  Local variables:
4:      tid ∈ [0..N]                                        ▷ Identifier of current thread
5:      rot-rset ← ∅                                        ▷ Transaction's read-set
6:  function READ(addr)                                     ▷ Read shared variable
7:      rot-rset ← rot-rset ∪{addr}                         ▷ Track ROT reads
8:  function SYNCHRONIZE
9:      s[N] ← status                                       ▷ Read and copy all status variables
10:     for i ← 0 to N−1 do                                 ▷ Wait until all threads...
11:         if s[i] = ACTIVE then                           ▷ ...that are active...
12:             wait until status[i] ≠ s[i]                 ▷ ...cross barrier
13: function TOUCH_VALIDATE
14:     for addr ∈ rot-rset do                              ▷ Re-read all elements...
15:         read addr                                       ▷ ...from read-set
16: function BEGIN_ROT
17:     repeat                                              ▷ Retry ROT forever
18:         status[tid] ← ACTIVE                            ▷ Indicate we are active
19:         MEM_FENCE                                       ▷ Make sure others know
20:         rot-rset ← ∅                                    ▷ Clear read-set
21:         tx ← TX_BEGIN_ROT                               ▷ HTM ROT begin
22:     until tx = STARTED                                  ▷ Repeat until success...
23: function COMMIT
24:     TX_SUSPEND                                          ▷ Suspend transaction
25:     status[tid] ← ROT-COMMITTING                        ▷ Tell others...
26:     MEM_FENCE                                           ▷ ...we are committing
27:     TX_RESUME                                           ▷ Resume transaction
28:     SYNCHRONIZE                                         ▷ Quiescence inside ROT
29:     TOUCH_VALIDATE                                      ▷ Touch to validate
30:     TX_COMMIT                                           ▷ End transaction
31:     status[tid] ← ⊥
```

4.3 Basic Algorithm

We first present below the basic version of the P8TM algorithm (Algorithm 1) assuming we only have ROTs and we blindly retry to execute failed ROTs irrespective of the abort cause.

To start a transaction, a thread first lets others know that it is *active* and initializes its data structures before actually starting a ROT (Lines 18–21). Then, during ROT execution, it just keeps track of reads to shared data by adding them to the thread-local *rot-rset* (Line 7). To complete the ROT, the thread first announces that it is *committing* by setting its shared *status* variable. Note that this is performed while the ROT is suspended (Lines 24–27) because otherwise the write would be buffered and invisible to other threads.

Next, the algorithm quiesces by waiting for all threads that are in a ROT to at least reach their commit phase (Lines 8–12). It then executes the touch-based validation mechanism, which simply consists in re-reading all address in the *rot-rset* (Lines 13–15), before finally committing the ROT (Line 30) and resetting the *status*.

4.4 Complete Algorithm

The naive approach of the basic algorithm to only use ROTs is unfortunately not practical nor efficient in real-world settings for two main reasons: (1) ROTs only provide "best effort" properties and thus a fallback is needed to guarantee liveness; and (2) using ROTs for short critical sections set that fit in a regular transaction is inefficient, because of the overhead of software-hardware read-set tracking and validation upon commit. Therefore, we extend the previous algorithm so that it first tries to use regular transactions, then switches to ROTs, and finally falls back to a global lock (GL) in order to guarantee progress. The pseudo-code of the complete algorithm is available in the extended version [18].

For transactions and ROTs to execute concurrently, the former must delay their commit until completion of all active ROTs. This is implemented using an RCU-like quiescence mechanism as in the basic algorithm. Transactions try to run in HTM and ROT modes a limited number of times, switching immediately if the cause of the failure is a capacity abort. The GL fallback uses a basic spin lock, which is acquired upon transaction begin and released upon commit. Note that the quiescence mechanism must also be called after acquiring the lock to wait for completion of ROTs that are in progress and might otherwise see inconsistent updates. Further, the GL fallback must also wait for ROTs to fully complete.

Read-only transactions. We finally describe the URO path, i.e., the execution mode optimized for read-only (RO) transactions in which reads are not tracked, hence significantly decreasing runtime overheads. This would also allow to execute large RO transactions that do not fit in hardware, and would otherwise be doomed to execute in the GL path.

To understand the intuition behind the URO path, note that whenever a URO develops a read-after-write with any concurrent transaction/ROT T, T is immediately aborted by the hardware. As for write-after-read conflicts, since transactions and ROTs buffer their writes and quiesce before committing, they cannot propagate inconsistent updates to RO transactions: this feature allows P8TM to achieve concurrency between UROs and transactions/ROTs, even when they encounter write-after-read conflicts (by serializing the URO before the T).

Finally, GL and RO transactions cannot conflict with each other as long as they do not run concurrently. This is ensured by performing a quiescence phase after acquiring the global lock, and executing RO transactions only when the lock is free. Note that, if the lock is taken, RO transactions defer to the writer by resetting their status before waiting for the lock to be free and retrying the whole procedure; otherwise we could run into a deadlock.

Correctness argument. When the GL path is active, concurrency is disabled. This is guaranteed since:

- **(i)** transactions in HTM path subscribe eagerly to the GL, and are thus aborted upon the activation of this path;
- **(ii)** after the GL is acquired, a quiescence phase is performed to wait for active ROTs or UROs.

Atomicity of a transaction in the HTM path is provided by the hardware against concurrent transactions/ROTs and by GL subscription.

As for the UROs, the quiescence mechanism guarantees two properties:

- UROs activated after the start of an update transaction T, and before the start of T's quiescence phase, can be safely serialized before T because they are guaranteed not to see any of T's updates, which are only made atomically visible when the corresponding transaction/ROT commits;

- UROs activated after the start of the quiescence phase of an update transaction T can be safely serialized after T because they are guaranteed to either abort T, in case they read a value written by T before T commits, or see all the updates produced by T's commit. It is worth noting here though that this is only relevant when UROs may conflict with T, in case of disjoint operation both serialization orders are equivalent.

Now we are only left with transactions running on the ROT path. The same properties of quiescence for UROs apply here and avoid ROTs reading inconsistent states produced by concurrent HTM transactions. Nevertheless, since ROTs do modify the shared state, they can still produce non-serializable histories; such as the scenario in Figure 2. Assume a ROT, say T_1, issued a read on X, developing a read-write conflict, with some concurrently active ROT, say T_2. There are two cases to consider: T_1 commits before T_2, or vice-versa.

If T_1 commits first, then if it reads X after T_2 (which is still active) wrote to it, then T_2 is aborted by the hardware conflict detection mechanism. Else, we are in presence of a write-after-read conflict. T_1 finds $status[T_2] := ACTIVE$ (because T_2 issues a fence before starting) and waits for T_2 to enter its commit phase (or abort). Then T_1 executes its T2V, during which, by re-reading X, would cause T_2 to abort.

Consider now the case in which T_2 commits before T_1. If T_1 reads X, as well as any other memory position updated by T_2, before T_2 writes to it, then T_1 can be safely serialized before T_2 (as T_1 observed none of T_2's updates). If T_1 reads X, or any other memory position updated by T_2, after T_2 writes to it and before T_2 commits, then T_2 is aborted by the hardware conflict detection mechanism; a contradiction. Finally, it is impossible for T_1 to read X after T_2 commits: in fact, during T_2's commit phase, T_2 must wait for T_1 to complete its execution; hence, T_1 must read X after T_2 writes to it and before T_2 commits, falling in the above case and yielding another contradiction.

Self-tuning. In workloads where transactions fit the HTM's capacity restrictions, P8TM forces HTM transactions to incur the overhead of suspend/resume, in order to synchronize them with possible concurrent ROTs. In these workloads, the ideal decision would be to just disable the ROT path, so to spare the HTM path from any overhead. However, it is not trivial to determine when it is beneficial to do so; this choice is workload dependent and is not trivial to determine via static code analysis techniques.

We address this issue by integrating into P8TM a self-tuning mechanism based on a lightweight reinforcement learning technique, UCB [21]. UCB determines, in an automatic fashion, which of the following modes to use:

(M1) HTM falling back to ROT, and then to GL;
(M2) HTM falling back directly to the GL;
(M3) starting directly in ROT before falling back to the GL.

Note that UROs and ROTs impose analogous overheads to HTM transactions. Thus, in order to reduce the search space to be explored by the self-tuning mechanism, whenever we disable ROTs (i.e., case (M2)), we also disable UROs (and treat RO transactions as update ones).

5 Read-set Tracking

The T2V mechanism requires to track the read-sets of ROTs for later replaying them at commit time. The implementation of the read-set tracking scheme is crucial for the performance of P8TM. In fact, as discussed in Section 3, ROTs do not track loads at the TMCAM level, but they do track stores and the read-set tracking mechanism must issue

stores in order to log the addresses read by a ROT. The challenge, hence, lies in designing a software mechanism that can exploit the TMCAM's capacity in a more efficient way than the hardware would do. In the following we describe two alternative mechanisms that tackle this challenge by exploring different trade-offs between computational and space efficiency.

Time-efficient implementation uses a thread local, cache aligned array, where each entry is used to track a 64-bit address. Since the cache lines of the POWER8 CPU are 128 bytes long, this means that 16 consecutive entries of the array, each storing an arbitrary address, will be mapped to the same cache line and occupy a single TMCAM entry. Therefore, this approach allows for fitting up to 16× larger read-sets within the TMCAM as compared to the case of HTM transactions. Given that they track 64 cache lines, each thread-local array is statically sized to store exactly 1024 addresses. It is worth noting here that since conflicts are detected at the cache line level granularity, it is not necessary to store the 7 least significant bits, as addresses point to the same cache line. However, we omit this optimization as this will add extra computational overhead, yielding a space saving of less than 10%.

Space-efficient implementation seeks to exploit the spatial data locality in the application's memory access patterns to compress the amount of information stored by the read-set tracking mechanism. This is achieved by detecting a common prefix between the previously tracked address and the current one, and by storing only the differing suffix and the size (in bytes) of the common prefix. The latter can be conveniently stored using the 7 least significant bits of the suffix, which, as discussed, are unnecessary. With applications that exhibit high spatial locality (e.g., that sequentially scan memory), this approach can achieve significant compression factors with respect to the time-efficient implementation. However, it introduces additional computational costs, both during the logging phase (to identify the common prefix) and in the replay phase (as addresses need to be reconstructed).

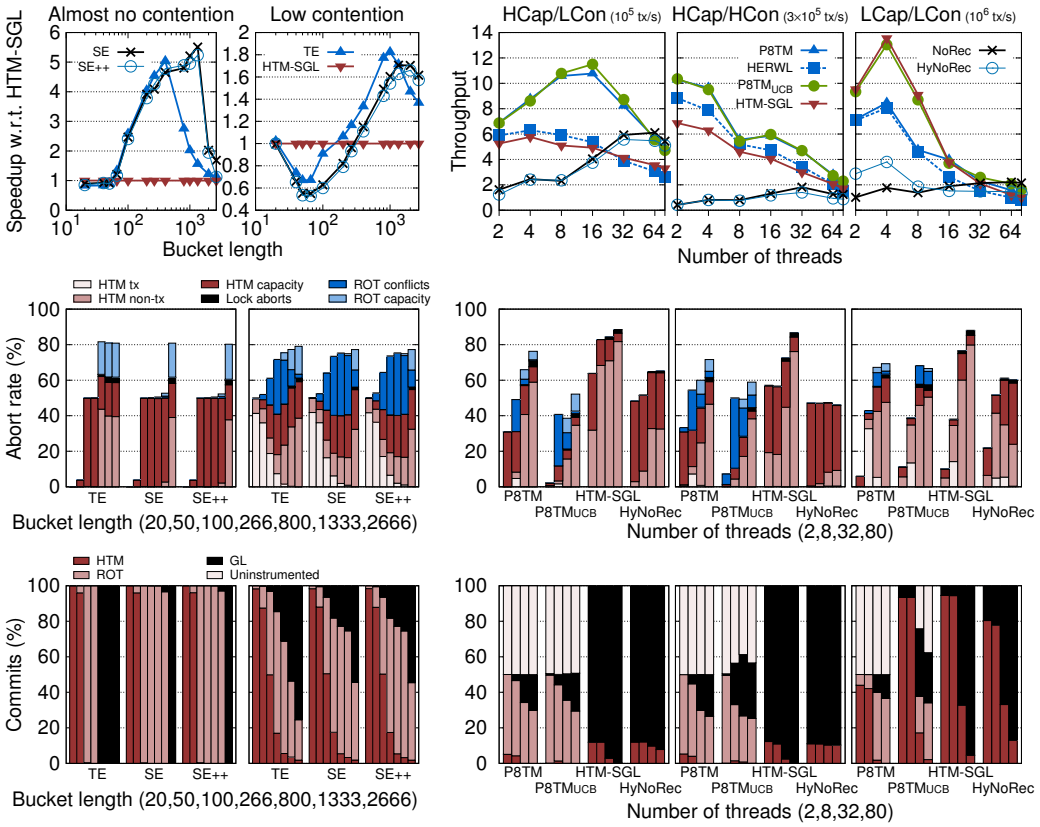
6 Evaluation

In this section we evaluate P8TM against state-of-the-art TM systems using a set of synthetic micro-benchmarks and complex, real-life applications. First, we start by evaluating both variants of read-set tracking to show how they are affected by the size of transactions and degree of contention. Then we conduct a sensitivity analysis aimed to investigate various factors that affect the performance of P8TM. To this end, we used a micro-benchmark that emulates a hashmap via lookup, insert, and delete transactions that accesses locations uniformly at random. This is a synthetic data structure composed of b buckets, where each bucket points to a linked-list, with an average length of l. By varying b and l we can control the degree of contention and probability of triggering capacity aborts respectively, which allows us to precisely stress different design aspects. Finally, we test P8TM using the popular STAMP benchmark suite [7].

We compare our solution with the following baselines:

(i) plain HTM with a global lock fallback (HTM-SGL),
(ii) NoRec with write back configuration,
(iii) the Hy-NoRec algorithm with three variables to synchronize transactions and NoRec fallback, and, finally,
(iv) the reduced hardware read-write lock elision algorithm HERWL (in this case, update transactions acquire the write lock while read-only transactions acquire the read lock).

Regarding the retry policy, we execute the HTM path 10 times and the ROT path 5 times before falling back to the next path, except upon a capacity abort when the next path is directly activated. These values and strategies were chosen after an extensive offline



(a) Evaluation of different implementations of read-set tracking.

(b) Sensitivity analysis: throughput and breakdown of abort rate and commits.

Figure 2 Micro-benchmarks (H=high, L=low, Cap=capacity, Con=contention).

experimentation and selecting the best configuration on average, regarding the number of retries and policies for capacity aborts (e.g., fallback immediately vs treating it as a conflict-induced abort). All results presented in this section represent the mean value of at least 5 runs. The experiments were conducted on a machine equipped with an IBM Power8 8284-22A processor that has 10 physical cores, with 8 hardware threads each, summing up to a total of 80 hardware threads. The source code, which is publicly available [1], was compiled with GCC 6.2.1 using `-O2` flag on top of Fedora 24 with Linux 4.5.5. Thread pinning was use to pin a thread per core at the beginning of each run for all the solutions, and threads were distributed evenly across the cores.

6.1 Read-set Tracking

The goal of this section is to understand the trade-off between the time-efficient and the space-efficient implementations of read-set tracking that were explained earlier in Section 5. We compare three variants of P8TM: i) a version using the time-efficient read-set tracking (TE), ii) a variant of space-efficient read-set tracking that only checks for prefixes of length 4 bytes, and otherwise stores the whole address (SE), and, finally, iii) a more aggressive variant of space-efficient read-set tracking that looks for prefixes of either 6 or 4 bytes (SE++).

Throughout this section, we fixed the number of threads to 10 (number of physical cores) and the percentage of update transactions at 100%, disabled the self-tuning module, and varied the buckets' lengths(l) across orders of magnitude to stress the ROT-path. First, we start with an almost contention-free workload (using $b = 10k$) to highlight the effect of capacity aborts alone. The speedup with respect to HTM-SGL, breakdown of abort rate (calculated as the aborts divided by the sum of aborted and committed transactions) and commits for this workload are shown in the left column of Figure 2a. As we can notice, the three variants of P8TM achieve almost the same performance as HTM-SGL with small transaction sizes that fit inside regular HTM transactions, as seen from the commits breakdown (l =∼20–50). However, when moving to larger transactions, the three variants start outperforming HTM-SGL achieving up to 5.5× higher throughput due to their ability to fit transactions within ROTs. By looking at the aborts breakdown in this region (l =∼100–266), we see that all P8TM variants suffer from almost 50% capacity aborts when first executing in HTM, and almost no capacity aborts when using the ROT path. This shows the clear advantage of the T2V mechanism and how it can fit more than 10× larger transactions in hardware.

Comparing TE with SE and SE++, we see that both space-efficient variants are able to execute larger transactions as ROTs: they do not suffer ROT capacity up to buckets of length ∼1333 items. Nevertheless, they incur an extra overhead, which is reflected as a slightly lower speedup than TE, before TE starts to experience ROT capacity aborts; only then their ability to further compress the *rot-rset* pays off. Again, by looking at the commits and aborts breakdown, we see that both space-efficient variants manage to commit all transactions as ROTs when TE is already using the GL (l =∼800–1333). Finally, when comparing SE and SE++, we notice that trying harder to find longer prefixes is not useful, due to the much lower probability of addresses sharing longer prefixes.

The right column of Figure 2a shows the results for a workload that exhibits a higher degree of contention ($b = 1k$). In this case, with transactions that fit inside regular HTM transactions, we see that HTM-SGL can outperform both SE and SE++ by up to 2× and TE by up to ∼30%. Since P8TM tries to execute transactions as ROTs after failing 10 times with HTM due to conflicts, the ROT path may be activated even in absence of capacity aborts; hence, the overhead of synchronizing ROTs and transaction becomes relevant also with small transactions. With larger transactions, we notice that the computational costs of SE and SE++ are more noticeable in this workload where they are always outperformed by TE, as long as this is able to fit at least 50% of transactions inside ROTs (up to l =∼800 items). Furthermore, the gains of SE and SE++ w.r.t. TE are much lower when compared to the contention-free workload. From this, we deduce that TE is more robust to contention. This was also confirmed with the other workloads that we will discuss next.

6.2 Sensitivity analysis

We now report the results of a sensitivity analysis that aimed to assess the impact of the following factors on P8TM's performance:

(i) the size of transactions,
(ii) the degree of contention, and
(iii) the percentage of read-only transactions.

We explored these three dimensions using the following configurations:

(i) high capacity, low contention, ($b = 1k$ and $l = 800$),
(ii) high capacity, high contention, ($b = 10$ and $l = 800$), and
(iii) low capacity, low contention ($b = 1k$ and $l = 40$).

We omitted showing the results for low capacity, high contention workload due to space restrictions, especially since they do not convey any extra information with respect to the low capacity, low contention scenario (which is actually even more favourable for HTM).

In these experiments we show two variants of P8TM, both equipped with the TE read-set tracking: with (P8TM$_{ucb}$) and without (P8TM) the self-tuning module enabled.

High capacity, low contention. The left most column of Figure 2b shows the throughput, abort rate and commits breakdown for the high capacity, low contention configuration with 50% update transactions.. We observe that, both variants of P8TM are able to outperform all the other TM solutions by up to 2×. This can be easily explained by looking at the commits breakdown, where both P8TM and P8TM$_{ucb}$ commit 50% of their transactions as UROs while the other 50% are committed mainly as ROTs up to 8 threads. On the contrary, HTM-SGL commits only 10% of the transactions in hardware and falls back to GL in the rest, due to the high capacity aborts it incurs. It is worth noting that the decrease in the percentage of capacity aborts, along with the increase of number of threads, is due to the activation of the fallback path, which forces other concurrent transactions to abort.

Although HERWL benefits from the URO path, P8TM was able to achieve ∼2× higher throughput, thanks to its ability of executing ROTs concurrently. Another interesting point is that P8TM$_{ucb}$ can outperform P8TM due to its ability to decrease the abort rate, as shown in the aborts breakdown. This is achieved by deactivating the HTM path, which spares from the cost of trying once in HTM before falling back to ROT (upon a capacity abort).

High capacity, high contention. The middle column of Figure 2b reports the results for the high capacity, high contention configuration with 50% update transactions. We can notice that although this workload is not scalable due to the high conflict rate, P8TM manages to achieve the highest throughput. Again this is due to P8TM's ability to fit large transactions into ROTs, almost all update transactions are executed in hardware up to 8 threads as can be seen from the commits breakdown. P8TM$_{ucb}$ also achieves higher throughput than P8TM after it disables the HTM path, which decreases the abort rate.

Low capacity, low-contention. In workloads where transactions fit inside HTM, it is expected that HTM-SGL will outperform all other TM solutions and that the overheads of P8TM will prevail. The results in the right most column of Figure 2b confirm this expectation: HTM-SGL outperforms all other solutions, achieving up to ∼1.75× higher throughput than P8TM. However, P8TM$_{ucb}$, thanks to its self-tuning ability, is the, overall, best performing solution, achieving performance comparable to HTM-SGL at low thread count, and outperforming it at high thread count. By inspecting the commits breakdown plots we see that P8TM$_{ucb}$ does not commit any transaction using ROTs up to 8 threads, avoiding the synchronization overheads that, instead, affect P8TM.

We note that, even though Hy-NoRec commits the same or higher percentage of HTM transactions than HTM-SGL, it is consistently outperformed by P8TM. This can be explained by looking at the performance of NoRec, which fails to scale due to the high instrumentation overheads it incurs with such short transactions. As for Hy-NoRec, its poor performance is a consequence of the inefficiency inherited by its NoRec fallback.

6.3 STAMP benchmark suite

STAMP is a popular benchmark suite that encompasses applications with different characteristics that share a common trait: they do not have any read-only transactions. Therefore, P8TM will not utilize the URO path and any gain it can achieve stems solely from executing

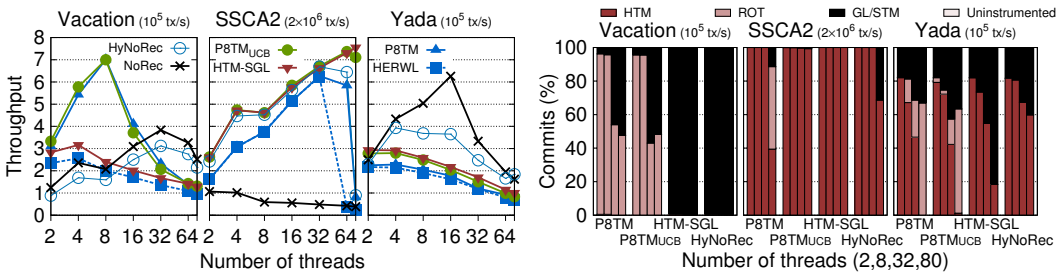


Figure 3 Throughput and breakdown of commits for the STAMP benchmarks.

ROTs in parallel. For space constraints we can only report the results for a subset of the STAMP benchmarks. The remaining benchmarks exhibit analogous trends and are available in an extended technical report [18].

Vacation is an application with medium sized transactions and low contention; hence, it behaves similarly to the previously analyzed high capacity, low contention workload. When looking at Figure 3, we can see trends very similar to the left most column of Figure 2b. P8TM is capable of achieving the highest throughput and outperforming HTM-SGL by up to ∼3.2× in this case. When looking at the breakdown of commits, we notice also the ability of P8TM to execute most of transactions as ROT up to 8 threads, while HTM-SGL never manages to commit transactions in hardware.

At high thread count we notice that NoRec and Hy-NoRec start to outperform both P8TM and P8TM$_{ucb}$. his can be explained by two reasons:

- with larger numbers of threads there is higher contention on hardware resources (note that starting from 32 threads ROT capacity aborts start to become frequent) and
- the cost of quiescence becomes more significant as threads have to wait longer.

Nevertheless, it is worth noting that the maximum throughput achieved by P8TM (at 8 threads) is ∼2× higher than NoRec (at 32 threads). This is due to the instrumentation overheads of these solutions. These overheads are completely eliminated in case of write accesses within P8TM and are much lower for read accesses—recall we only need to log the addresses and read them during validation.

SSCA2 generates transactions with small read/write sets and low contention. These are HTM friendly characteristics, and by looking at the throughput results in Figure 3 we see that HTM-SGL is able to outperform all the other baselines and scale up to 80 threads. This is also reflected in its ability to commit almost all transactions in hardware as shown in the commits breakdown. Although Hy-NoRec is able to achieve performance similar to HTM up to 32 threads, it is then outperformed due to the extra overheads it incurs to synchronize with the NoRec fallback.

Although P8TM commits almost all transactions using HTM up to 64 threads, it performed worse than both HTM-SGL and Hy-NoRec due to the costs of synchronization. An interesting observation is that the overhead is almost constant up to 32 threads. In fact, up to 64 threads there are no ROTs running and the overhead is is dominated by the cost of suspending and resuming the transaction. At 64 and 80 threads P8TM started to suffer also from capacity aborts similarly to Hy-NoRec. This led to a degradation of performance, with HTM-SGL achieving 7× higher throughput at 80 threads. This is a workload where P8TM$_{ucb}$ comes in handy as it manages to disable the ROT path and thus tends to employ HTM-SGL.

Yada has long transactions, large read/write set and medium contention. This is an example of a workload that is not hardware friendly and where hardware solutions are expected to be outperformed by software based ones. Figure 3 shows the clear advantage

of NoRec over any other solution, achieving up to 3× higher throughput than hardware based solutions. When looking at the commits and abort break down, one can see that up to 8 threads P8TM commits ∼80% of the transactions as either HTM or ROTs. Yet, despite P8TM manages to reduce the frequency of acquisition of the GL path with respect to HTM-SGL, it incurs overheads that end up outweighing the benefits provided by P8TM in terms of increased concurrency.

7 Conclusion

We presented P8TM, a TM system that tackles what is, arguably, the key limitation of existing HTM systems: the inability to execute transactions whose working sets exceed the capacity of CPU caches. This is achieved by novel techniques that exploit hardware capabilities available in POWER8 processors. Via an extensive experimental evaluation, we have shown that P8TM provides robust performance across a wide range of benchmarks, ranging from simple data structures to complex applications, and achieves remarkable speedups.

The importance of P8TM stems from the consideration that the best-effort nature of current HTM implementations is not expected to change in the near future. Therefore, techniques that mitigate the intrinsic limitations of HTM can broaden its applicability to a wider range of real-life workloads. We conclude by arguing that the performance benefits achievable by P8TM thanks to the use of the ROT and suspend/resume mechanisms represent a relevant motivation for integrating these features in future generations of HTM-enabled processors (like Intel's ones).

References

1 `https://github.com/shadyalaa/POWER8TM`, 2017.

2 A. Adl-Tabatabai, , T. Shpeisman, and J. Gottschlich. "draft specification of transactional language constructs for c++. *Intel*, 2012.

3 Y. Afek, A. Levy, and A. Morrison. Programming with hardware lock elision. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'13)*, pages 295–296, 2013. `doi:10.1145/2442516.2442552`.

4 H. Boehm, J. Gottschlich, V. Luchangco, M. Michael, M. Moir, C. Nelson, T. Riegel, T. Shpeisman, and M. Wong. Transactional language constructs for c++. *ISO/IEC JTC1/SC22 WG21 (C++)*, 2012.

5 H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, and H. Le. Robust architectural support for transactional memory in the power architecture. *SIGARCH Comput. Archit. News*, 41(3), 2013.

6 I. Calciu, T. Shpeisman, G. Pokam, and M. Herlihy. Improved single global lock fallback for best-effort hardware transactional memory. *9th ACM SIGPLAN Wkshp. on Transactional Computing*, 2014.

7 Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC'08*, 2008.

8 L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear. Hybrid norec: A case study in the effectiveness of best effort hardware transactional memory. In *ASPLOS'11*, 2011.

9 P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS'06*, 2006.

10 D. Dice, T. L. Harris, A. Kogan, Y. Lev, and M. Moir. Hardware extensions to make lazy subscription safe. *CoRR*, abs/1407.6968, 2014.

11 D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *ASPLOS'09*, 2009.

12 D. Dice and N. Shavit. Understanding tradeoffs in software transactional memory. In *CGO'07*, 2007.

13 N. Diegues and P. Romano. Self-tuning intel transactional synchronization extensions. In *ICAC'14*, 2014.

14 N. Diegues, P. Romano, and S. Garbatov. Seer: Probabilistic scheduling for hardware transactional memory. In *SPAA'15*, 2015.

15 N. Diegues, P. Romano, and L. Rodrigues. Virtues and limitations of commodity hardware transactional memory. In *PACT'14*, 2014.

16 P. Felber, S. Issa, A. Matveev, and P. Romano. Hardware read-write lock elision. In *EuroSys'16*, 2016.

17 B. Goel, R. Titos-Gil, A. Negi, S. A. McKee, and P. Stenstrom. Performance and energy analysis of the restricted transactional memory implementation on haswell. In *IPDPS'14*, 2014.

18 S. Issa, P. Felber, A. Matveev, and P. Romano. Extending hardware transactional memory capacity via rollback-only transactions and suspend/resume. Technical report, INESC-ID'17, 2017.

19 C.C Jacobi, T. Slegel, and D. Greiner. Transactional memory architecture and implementation for ibm system z. In *MICRO-45*, 2012.

20 S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *PPoPP'06*, 2006.

21 T. Lai and H. Robbins. Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, 1985.

22 H.Q. Le, G.L. Guthrie, D.E. Williams, M.M. Michael, B.G. Frey, W.J. Starke, C. May, R. Odaira, and T. Nakaike. Transactional memory support in the ibm power8 processor. *IBM Journal of Research and Development*, 59(1), 2015.

23 A. Matveev and N. Shavit. Reduced hardware norec: A safe and scalable hybrid transactional memory. In *ASPLOS'15*, 2015.

24 T. Nakaike, R. Odaira, M. Gaudet, M. M. Michael, and H. Tomari. Quantitative comparison of hardware transactional memory for blue gene/q, zenterprise ec12, intel core, and power8. In *ISCA'15*, 2015.

25 R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of intel transactional synchronization extensions for high-performance computing. In *SC'13*, 2013.

Some Lower Bounds in Dynamic Networks with Oblivious Adversaries*†

Irvan Jahja[1], Haifeng Yu[2], and Yuda Zhao[‡3]

1 **National University of Singapore, Singapore**
irvan@comp.nus.edu.sg
2 **National University of Singapore, Singapore**
haifeng@comp.nus.edu.sg
3 **Grab, Singapore**
yudazhao@gmail.com

Abstract

This paper considers several closely-related problems in synchronous dynamic networks with *oblivious adversaries*, and proves novel $\Omega(d + \text{poly}(m))$ lower bounds on their time complexity (in rounds). Here d is the dynamic diameter of the dynamic network and m is the total number of nodes. Before this work, the only known lower bounds on these problems under oblivious adversaries were the trivial $\Omega(d)$ lower bounds. Our novel lower bounds are hence the first non-trivial lower bounds and also the first lower bounds with a $\text{poly}(m)$ term. Our proof relies on a novel reduction from a certain two-party communication complexity problem. Our central proof technique is unique in the sense that we consider the communication complexity with a special *leaker*. The leaker helps Alice and Bob in the two-party problem, by disclosing to Alice and Bob certain "non-critical" information about the problem instance that they are solving.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases dynamic networks, oblivious adversary, adaptive adversary, lower bounds, communication complexity

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.29

1 Introduction

Dynamic networks [22] is a flourishing topic in recent years. We consider a synchronous setting where the m (fixed) nodes in the network proceed in synchronous rounds. Each node has a unique id of size $O(\log m)$, and the messages are of size $O(\log m)$ as well. The nodes never fail. The topology of the dynamic network can change from round to round, as determined by an *adversary*, subject to the only constraint that the topology in each round must be a connected and undirected graph. The *time complexity* of a protocol is the number of rounds needed for all nodes to generate the final output, over the worst-case adversary, worst-case initial values, and average coin flips of the protocol. We consider a number of fundamental distributed computing problems within such a context:

* The authors of this paper are alphabetically ordered. This work is partly supported by the research grant MOE2014-T2-2-030 from Singapore Ministry of Education Academic Research Fund Tier-2.
† The full version of this paper is available at http://www.comp.nus.edu.sg/~yuhf/oblivious-disc17-technicalreport.pdf.
‡ This work was done while this author was in National University of Singapore.


© Irvan Jahja, Haifeng Yu, and Yuda Zhao;
licensed under Creative Commons License CC-BY

- Consensus: Each node has a binary input. The nodes aim to achieve a consensus (with the standard agreement, validity, and termination requirements) and output the final decision.
- LeaderElect: Each node should output the leader's id.
- ConfirmedFlood: A certain node ν aims to propagate a token of size $O(\log m)$ to all other nodes, and wants to further confirm that all nodes have received the token.[1] Formally, node ν's output is correct only if by the time that ν outputs, the token has already been received by all the nodes. (The value of the output is not important.) The remaining nodes can output any time.
- Aggregation: Each node has a value of $O(\log m)$ bits, and the nodes aim to compute a certain aggregation function over all these values. We consider two specific aggregation functions, Sum and Max.

Let d be the *(dynamic) diameter* (see definition later) of the dynamic network. (Note that since the topology is controlled by an adversary, the protocol never knows d beforehand.) Given an optimal protocol for solving any of the above problems, let $\mathrm{tc}(d, m)$ denote the protocol's time complexity, when it runs over networks with d diameter and m nodes. It is easy to see that $\mathrm{tc}(d, m)$ crucially depends on d, since we trivially have $\mathrm{tc}(d, m) = \Omega(d)$. Given such, this paper focus on the following central question:

Ignoring polylog(m) terms, is tc(d, m) independent of the network size m?

Answering this fundamental question will reveal whether the complexity of all these basic problems is due to the diameter or due to both the diameter and the network size.

Existing results. If the network were *static*, then building a spanning tree would solve all these problems in either $O(d)$ or $O(d \log m)$ rounds, implying a **yes** answer to the above question. In dynamic networks, the picture is more complex. In a dynamic network model without congestion (i.e., message size unlimited), Kuhn et al. [20] have proposed elegant upper bound protocols with $O(d)$ complexity for all these problems. Hence the answer is **yes** as well. For dynamic networks with congestion (i.e., message size limited to $O(\log m)$), Yu et al. [25] recently have proved that $\mathrm{tc}(d, m) = O(d \log m)$ for Consensus and LeaderElect, if the nodes know a *good* estimate on m.[2] Hence the answer is **yes** in such cases. On the other hand, if nodes' estimate on m is *poor*,[3] then Yu et al. [25] prove a lower bound of $\Omega(d + \mathrm{poly}(m))$ for Consensus and LeaderElect, implying a **no** answer. For ConfirmedFlood and Aggregation, they have also proved $\mathrm{tc}(d, m) = \Omega(d + \mathrm{poly}(m))$, even if the nodes know m. This implies a **no** answer for those two problems.

All the lower bound proofs in [25], however, critically relies on a powerful *adaptive adversary*: In each round, the adaptive adversary sees all the coin flip outcomes so far of the protocol $\mathcal{P}$ and manipulates the topology based on those. In particular, in each round the adversary sees whether each node will be sending (and can then manipulate the topology accordingly), *before* the nodes actually send their messages. Their proof breaks

[1] Such confirmation does not have to come from explicit acknowledgements, and can be via implicit means, such as counting the number of rounds.

[2] More precisely, if the nodes know m' such that $|\frac{m'-m}{m}| \leq \frac{1}{3} - c$ for some positive constant c. Obviously, this covers the case where the nodes know m itself.

[3] More precisely, if the nodes only knows m' such that $|\frac{m'-m}{m}|$ reaches $\frac{1}{3}$ or above. Obviously, this covers the case where the nodes do not have any knowledge about m.

under *oblivious adversaries*, which do not see $\mathcal{P}$'s coin flip outcomes and have to decide the topologies in all the rounds before $\mathcal{P}$ starts.[4]

In summary, our central question of whether $\text{tc}(d, m)$ is largely independent of the network size m has been answered in:

(i) static networks,
(ii) dynamic networks without congestion under both adaptive and oblivious adversaries, and
(iii) dynamic networks with congestion under adaptive adversaries.

Our results. This work gives the last piece of the puzzle for answering our central question. Specifically, we show that in dynamic networks with congestion and under oblivious adversaries, for Consensus and LeaderElect, the answer to the question is **no** when the nodes' estimate on m is poor. (If the nodes' estimate on m is good, results from [25] already implied a **yes** answer.) Specifically, we prove a novel $\Omega(d + \text{poly}(m))$ lower bound on Consensus under oblivious adversaries, when the nodes' estimate on m is poor. This is the first non-trivial lower bound and also the first lower bound with a $\text{poly}(m)$ term, for Consensus under oblivious adversaries. The best lower bound before this work was the trivial $\Omega(d)$ lower bound. Our Consensus lower bound directly carries over to LeaderElect since Consensus reduces to LeaderElect [25].

Our approach will also lead to a $\Omega(d + \text{poly}(m))$ lower bound under oblivious adversaries for ConfirmedFlood, which in turn reduces to Sum and Max [25]. Such a lower bound similarly gives a **no** answer for ConfirmedFlood and Aggregation. But since the lower bound proof for ConfirmedFlood is similar to and in fact easier than our Consensus proof, for clarity, we will not separately discuss it in this paper.

Different adversaries. In dynamic networks, different kinds of adversaries often require different algorithmic techniques and also yield different results. Hence it is common for researchers to study them separately. For example, lower bounds for information dissemination were proved separately, under adaptive adversaries [13] and then later under oblivious adversaries [1]. Dynamic MIS was investigated separately under adaptive adversaries [17] and later under oblivious adversaries [8]. Broadcasting was first studied under adaptive adversaries [18], and later under oblivious adversaries [14].

Our approach. Our novel Consensus lower bound under oblivious adversaries is obtained via a reduction from a two-party communication complexity (CC) problem called *Gap Disjointness with Cycle Promise* or Gdc. Our reduction partly builds upon the reduction in [25] for adaptive adversaries, but has two major differences. In fact, these two novel aspects also make our central proof technique rather unique, when compared with other works that use reductions from CC problems [9, 12, 21].

The first novel aspect is that we reduce from Gdc with a special *leaker* that we design. The leaker is an oracle in the Gdc problem, and is separate from the two parties Alice and Bob . It helps Alice and Bob, by disclosing to them certain "non-critical" information in the following way. For a CC problem Π, let $\Pi_n(X, Y)$ be the answer to Π for length-n inputs X and Y. Let x_i and y_i denote the i-th character of X and Y, respectively. A pair (a, b) is defined to be a *leakable pair* if for all n, X, Y, and $i \in [0, n]$, $\Pi_n(x_1 x_2 \dots x_n, y_1 y_2 \dots y_n) =$

[4] Note however that all upper bounds, from [20] and [25], will directly carry over to oblivious adversaries.

$\Pi_{n+1}(x_1x_2...x_iax_{i+1}x_{i+2}...x_n, y_1y_2...y_iby_{i+1}y_{i+2}...y_n)$. Intuitively, inserting or removing a leakable pair does not impact the answer to Π. For each index i where (x_i, y_i) is leakable, independently with probability $\frac{1}{2}$, our leaker *leaks* the index i, by letting both Alice and Bob know for free the value of i and the value of the pair (x_i, y_i), before Alice and Bob start running their protocol.

Our reduction from GDC (with our leaker) to CONSENSUS still does not allow us to directly use an oblivious adversary. Instead, as the second novel aspect, we will use a special kind of adaptive adversaries which we call *sanitized adaptive adversaries*. These adversaries are still adaptive, but their adaptive decisions have been "sanitized" by taking XOR with independent coin flips. We then show that a sanitized adaptive adversary is no more powerful than an oblivious adversary, in terms of incurring the cost of a protocol.

2 Related Work

This section discusses related works beyond those already covered in the previous section.

Related work on Consensus and LeaderElect. Given the importance of CONSENSUS and LEADERELECT in dynamic networks, there is a large body of related efforts and we can only cover the most relevant ones. In dynamic networks without congestion, Kuhn et al. [20] show that the *simultaneous consensus* problem has a lower bound of $\Omega(d + \text{poly}(m))$ round. In this problem, the nodes need to output their consensus decisions simultaneously. Their knowledge-based proof exploits the need for simultaneous actions, and does not apply to our setting. Some other researchers (e.g., [3, 4]) have studied CONSENSUS and LEADERELECT in a dynamic network model where the set of nodes can change and where the topology is an *expander*. Their techniques (e.g., using random walks) critically reply on the expander property of the topology, and hence do not apply to our setting. Augustine et al. [2] have proved an upper bound of $O(d \log m)$ for LEADERELECT in dynamic networks while assuming d is known to all nodes. This does not contradict with our lower bound, since we do not assume the knowledge of d. Certain CONSENSUS and LEADERELECT protocols (e.g., [15]) assume that the network's topology eventually stops changing, which is different from our setting where the change does not stop. CONSENSUS and LEADERELECT have also been studied in *directed* dynamic networks (e.g., [11, 23]), which are quite different from our undirected version. In particular, lower bounds there are mostly obtained by exploiting the lack of guaranteed bidirectional communication in directed graphs. Our AGGREGATION problem considers the two aggregation functions SUM and MAX. Cornejo et al. [10] considers a different aggregation problem where the goal is to collect distributed tokens (without combining them) to a small number of nodes. Some other research (e.g., [6]) on AGGREGATION assumes that the topology is each round is a (perfect) matching, which is different from our setting where the topology must be connected.

Related work on reductions from CC. Reducing from two-party CC problems to obtain lower bounds for distributed computing problem has been a popular approach in recent years. For example, Kuhn et al. [21] and Das Sarma et al.[12] have obtained lower bounds on the *hear-from* problem and the *spanning tree verification* problem, respectively, by reducing from DISJOINTNESS. In particular, Kuhn et al.'s results suggest that the *hear-from* problem has a lower bound of $\Omega(d + \sqrt{m}/\log m)$ in *directed static networks*. Chen et al.'s work [9] on computing SUM in *static networks with node failures* has used a reduction from the $\text{GDC}_n^{1,q}$ problem. Our reduction in this paper is unique, in the sense that none of these previous reductions use the two key novel techniques in this work, namely CC with our leaker and sanitized adaptive adversaries.

Related work on CC. To the best of our knowledge, we are the first to exploit the CC with a leaker in reductions to distributed computing problems such as Consensus. Our leaker serves to allow oblivious adversaries. Quite interestingly, for completely different purposes, the notions of leakable pairs and a leaker have been extensively (but implicitly) used in proofs for obtaining direct sum results on the information complexity (IC) (e.g., [5, 7, 24]) of various communication problems: First, leakable pairs have been used to construct a *collapsing input*, for the purpose of ensuring that the answer to the problem Π is entirely determined by (x_i, y_i) at some index i. Second, an (implicit) leaker has often been used (e.g., in [7, 24]) to enable Alice and Bob to draw $(\mathbf{X}, \mathbf{Y})$ from a non-product distribution.

Because of the fundamentally different purposes of leaking, our leaker differs from those (implicit) leakers used in works on IC, in various specific aspects. For example in our work, all leakable pairs are subject to leaking, while in the works on IC, there is some index i that is never subject to leaking. Also, when our leaker leaks index j, it discloses both $\mathbf{x}_j$ and $\mathbf{y}_j$ to both Alice and Bob. In comparison, in works on IC, the (implicit) leaking is usually done differently: For example, Alice and Bob may use public coins to draw $\mathbf{x}_j$ and Bob may use his private coins to draw $\mathbf{y}_j$. Doing so (implicitly) discloses $\mathbf{x}_j$ to both Alice and Bob and (implicitly) discloses $\mathbf{y}_j$ *only* to Bob.

A key technical step in our work is to prove a lower bound on the CC of $\textsc{Gdc}_n^{g,q}$ with our leaker. For simpler problems such as Disjointness (which is effectively $\textsc{Gdc}_n^{1,2}$), we believe that such a lower bound could alternatively be obtained by studying its IC with our leaker. But the gap promise and the cycle promise in $\textsc{Gdc}_n^{g,q}$ make IC arguments rather tricky. Hence we will (in Section 8) obtain our intended lower bound by doing a direct reduction from the CC of $\textsc{Gdc}_{n'}^{g,q}$ without the leaker to the CC of $\textsc{Gdc}_n^{g,q}$ with the leaker.

3 Model and Definitions

Conventions. All protocols in this paper refer to Monte Carlo randomized algorithms. We always consider public coin protocols, which makes our lower bounds stronger. All log is base 2, while ln is base e. Upper case fonts (e.g., X) denote strings, vectors, sets, etc. Lower case fonts (e.g., x) denote scalar values. In particular, if X is a string, then x_i means the i-th element in X. Bold fonts (e.g., $\mathbf{X}$ and $\mathbf{x}$) refer to random variables. Blackboard bold fonts (e.g., $\mathbb{D}$) denote distributions. We write $\mathbf{x} \sim \mathbb{D}$ if $\mathbf{x}$ follows the distribution $\mathbb{D}$. Script fonts (e.g., $\mathcal{P}$ and $\mathcal{Q}$) denote either protocols or adversaries.

Dynamic networks. We consider a synchronous dynamic network with m fixed nodes, each with a unique id of $\Theta(\log m)$ bits. A protocol in such a network proceeds in synchronous rounds, and starts executing on all nodes in round 1. (Clearly such simultaneous start makes our lower bound stronger.) In each round, each node v first does some local computation, and then chooses to either send a single message of $O(\log m)$ size or receive. All nodes who are v's neighbors in that round and are receiving in that round will receive v's message at the end of the round. A node with multiple neighbors may receive multiple messages.

The topology of the network may change arbitrarily from round to round, as determined by some *adversary*, except that the topology in each round must be a connected undirected graph. (This is the same as the 1-interval model [19].) A node does not know the topology in a round. It does not know its neighbors either, unless it receives messages from them in that round. Section 1 already defined *oblivious adversaries* and *adaptive adversaries*. In particular in each round, an adaptive adversary sees all $\mathcal{P}$'s coin flip outcomes up to and including the current round, and manipulates the topology accordingly, before $\mathcal{P}$ uses the current round's coin flip outcomes.

We use the standard definition for the *(dynamic) diameter* [22] of a dynamic network: Intuitively, the diameter of a dynamic network is the minimum number of rounds needed for every node to influence all other nodes. Formally, we say that $(\omega, r) \to (v, r+1)$ if either ω is v's neighbor in round r or $\omega = v$. The *diameter* d of a dynamic network is the smallest d such that $(\omega, r) \rightsquigarrow (v, r+d)$ for all ω, v, and r, where "$\rightsquigarrow$" is the transitive closure of "$\to$". Since the topology is controlled by an adversary, a protocol never knows d beforehand.

Communication complexity. In a two-party communication complexity (CC) problem Π_n, Alice and Bob each hold input strings X and Y respectively, where each string has n *characters*. A character here is q-ary (i.e., an integer in $[0, q-1]$) for some given integer $q \geq 2$. For any given i, we sometimes call (x_i, y_i) as a *pair*. Alice and Bob aim to compute the value of the binary function $\Pi_n(X, Y)$. Given a protocol $\mathcal{P}$ for solving Π (without a leaker), we define $\text{cc}(\mathcal{P}, X, Y, \mathbf{C}_\mathcal{P})$ to be the communication incurred (in terms of number of bits) by $\mathcal{P}$, under the input (X, Y) and $\mathcal{P}$'s coin flip outcomes $\mathbf{C}_\mathcal{P}$. Note that $\mathbf{C}_\mathcal{P}$ is a random variable while cc() is a deterministic function. We similarly define $\text{err}(\mathcal{P}, X, Y, \mathbf{C}_\mathcal{P})$, which is 1 if $\mathcal{P}$'s output is wrong, and 0 otherwise. We define the *communication complexity* of $\mathcal{P}$ to be $\text{cc}(\mathcal{P}) = \max_X \max_Y E_{\mathbf{C}_\mathcal{P}}[\text{cc}(\mathcal{P}, X, Y, \mathbf{C}_\mathcal{P})]$, and the *error* of $\mathcal{P}$ to be $\text{err}(\mathcal{P}) = \max_X \max_Y E_{\mathbf{C}_\mathcal{P}}[\text{err}(\mathcal{P}, X, Y, \mathbf{C}_\mathcal{P})]$. We define the δ-error $(0 < \delta < \frac{1}{2})$ *communication complexity* of Π_n to be $\mathfrak{R}_\delta(\Pi_n) = \min \text{cc}(\mathcal{P})$, with the minimum taken over all $\mathcal{P}$ where $\text{err}(\mathcal{P}) \leq \delta$. For convenience, we define $\mathfrak{R}_\delta(\Pi_0) = 0$ and $\mathfrak{R}_\delta(\Pi_a) = \mathfrak{R}_\delta(\Pi_{\lfloor a \rfloor})$ for non-integer a.

We define similar concepts for CC with our leaker. Section 1 already defined leakable pairs and how our leaker works. Given $\mathcal{P}$ for solving Π with our leaker, $\text{cc}(\mathcal{P}, X, Y, \mathbf{C}_\mathcal{P}, \mathbf{C}_\mathcal{L})$ is the communication incurred by $\mathcal{P}$, under the input (X, Y), $\mathcal{P}$'s coin flip outcomes $\mathbf{C}_\mathcal{P}$, and the leaker's coin flip outcomes $\mathbf{C}_\mathcal{L}$. Here (X, Y) and $\mathbf{C}_\mathcal{L}$ uniquely determine which indices get leaked. We define $\text{cc}(\mathcal{P}) = \max_X \max_Y E_{\mathbf{C}_\mathcal{L}} E_{\mathbf{C}_\mathcal{P}}[\text{cc}(\mathcal{P}, X, Y, \mathbf{C}_\mathcal{P}, \mathbf{C}_\mathcal{L})]$. We similarly define $\text{err}(\mathcal{P}, X, Y, \mathbf{C}_\mathcal{P}, \mathbf{C}_\mathcal{L})$ and $\text{err}(\mathcal{P})$. Finally, we define the δ-error $(0 < \delta < \frac{1}{2})$ *communication complexity* of Π_n with our leaker, denoted as $\mathfrak{L}_\delta(\Pi_n)$, to be $\mathfrak{L}_\delta(\Pi_n) = \min \text{cc}(\mathcal{P})$, with the minimum taken over all $\mathcal{P}$ such that $\mathcal{P}$ solves Π_n with our leaker and $\text{err}(\mathcal{P}) \leq \delta$. Note that we always have $\mathfrak{L}_\delta(\Pi_n) \leq \mathfrak{R}_\delta(\Pi_n)$.

4 Preliminaries on Gap Disjointness with Cycle Promise

The section defines the two-party GDC problem and describes some basic properties of GDC.

▶ **Definition 1** (Gap Disjointness with Cycle Promise)**.** In *Gap Disjointness with Cycle Promise*, denoted as $\text{GDC}_n^{g,q}$, Alice and Bob have input strings X and Y, respectively. X and Y each have n characters, and each character is an integer in $[0, q-1]$. Alice and Bob aim to compute $\text{GDC}_n^{g,q}(X, Y)$, defined to be 1 if (X, Y) contains no $(0, 0)$ pair, and 0 otherwise. The problem comes with the following two promises:

- **Gap promise:** (X, Y) contains either no $(0, 0)$ pair or at least g such pairs.
- **Cycle promise [9]:** For each index i, x_i and y_i satisfy exactly one of the following four conditions: i) $x_i = y_i = 0$, ii) $x_i = y_i = q-1$, iii) $x_i = y_i + 1$, or iv) $x_i = y_i - 1$.

One can easily verify that the cycle promise is trivially satisfied when $q = 2$. It is also easy to see $\text{GDC}_n^{1,2}$ degenerates to the classic DISJOINTNESS problem. The gap promise and the cycle promise start to impose material restrictions when $g \geq 2$ and $q \geq 3$, respectively. For example for $g = 2$ and $q = 4$, $X = 02103$ and $Y = 03003$ satisfy both the two promises, where (X, Y) contains 2 pairs of $(0, 0)$, at indices 1 and 4. For GDC, all $(0, 0)$ pairs are

non-leakable, while all other pairs are leakable. For example for $X = 02103$ and $Y = 03003$, those 3 pairs at index 2, 3, and 5 are leakable. The following result (proven in the full version [16] of this paper) on the CC of GDC is a simple adaption from the result in [9]:

▶ **Theorem 2.** *For any δ where $0 < \delta < 0.5$, there exist constants $c_1 > 0$ and $c_2 > 0$ such that for all n, g, and q, $\mathfrak{R}_\delta(\text{GDC}_n^{g,q}) \geq \frac{c_1 n}{gq^2} - c_2 \log \frac{n}{g}$.*

The proof of Theorem 2 also showed that $\mathfrak{R}_\delta(\text{GDC}_n^{g,q}) \geq \mathfrak{R}_\delta(\text{GDC}_{n/g}^{1,q})$. It is important to note that $\mathfrak{L}_\delta(\text{GDC}_n^{g,q}) \geq \mathfrak{L}_\delta(\text{GDC}_{n/g}^{1,q})$ does *not* hold in general (see [16] for more discussion). Hence when later proving the lower bound on $\mathfrak{L}_\delta(\text{GDC}_n^{g,q})$, we will have to work with the gap promise directly, instead of obtaining the lower bound via $\mathfrak{L}_\delta(\text{GDC}_{n/g}^{1,q})$.

5 Review of Existing Proof under Adaptive Adversaries

This section gives an overview of the recent CONSENSUS lower bound proof [25] under adaptive adversaries. That proof is quite lengthy and involved, hence we will stay at the high-level, while focusing on aspects that are more relevant to this paper.

Overview. Consider any oracle CONSENSUS protocol $\mathcal{P}$ with $\frac{1}{10}$ error. Let $\text{tc}(d, m)$ be $\mathcal{P}$'s time complexity, when running over dynamic network controlled by adaptive adversaries and with d diameter and m nodes. The proof in [25] is mainly for proving $\text{tc}(8, m) = \Omega(\text{poly}(m))$. The proof trivially extends to $\text{tc}(d, m)$ for all $d \geq 8$. Combining with the trivial $\Omega(d)$ lower bound will lead to the final lower bound of $\Omega(d + \text{poly}(m))$.

To prove $\text{tc}(8, m) = \Omega(\text{poly}(m))$, [25] uses a reduction from $\text{GDC}_n^{g,q}$ to CONSENSUS. To solve $\text{GDC}_n^{g,q}(X, Y)$, Alice knowing X and Bob knowing Y simulate the CONSENSUS protocol $\mathcal{P}$ in the following way: In the simulation, the input (X, Y) is mapped to a dynamic network. Roughly speaking, if $\text{GDC}_n^{g,q}(X, Y) = 1$, the resulting dynamic network will have a diameter of 8. Hence $\mathcal{P}$ should decide within $r_1 = \text{tc}(8, m)$ rounds on expectation. If $\text{GDC}_n^{g,q}(X, Y) = 0$, then the resulting dynamic network will have a diameter of roughly $\frac{q}{2}$. It is then shown [25] that $\mathcal{P}$ must take $r_2 = \Omega(q)$ rounds to decide in dynamic networks with such a diameter. The value of q is chosen, as a function of $\text{tc}(8, m)$, such that $r_2 > 10r_1$. Alice and Bob determine the answer to GDC based on when $\mathcal{P}$ decides: If $\mathcal{P}$ decides within $10r_1$ rounds, they claim that $\text{GDC}_n^{g,q}(X, Y) = 1$. Otherwise they claim $\text{GDC}_n^{g,q}(X, Y) = 0$.

To solve GDC using the above simulation, Alice and Bob need to simulate $\mathcal{P}$ for $10r_1 = 10\text{tc}(8, m)$ rounds. In each round, to enable the simulation to continue, Alice and Bob will need to incur $O(\log m)$ bits of communication. Hence altogether, they incur $10\text{tc}(8, m) \cdot O(\log m)$ bits for solving $\text{GDC}_n^{g,q}$. The lower bound on the CC of $\text{GDC}_n^{g,q}$ then immediately translates to a lower bound on $\text{tc}(8, m)$.

Crux of the proof. When solving GDC, Alice only knows X and not Y. This means that Alice does *not* actually have the full knowledge of the dynamic network, which is a function of (X, Y). Hence the proof's central difficulty is to design the dynamic network in such a way that Alice can nevertheless still properly simulate $\mathcal{P}$ over that dynamic network. The proof in [25] overcomes this key difficulty by i) leveraging the cycle promise in GDC, and ii) using an *adaptive* adversary — in particularly, using an adaptive adversary is highlighted [25] as a key technique. We give a concise review below.

Given (X, Y), the dynamic network constructed in [25] has one *chain* for each index $i \in [1, n]$. Each chain has 3 node in a line (Figure 1). Consider as an example the i-th chain where $x_i = 0$. Since $x_i = 0$, y_i must be either 0 or 1 (by the cycle promise). The set of edges

(a) ν is sending in round $t_i + 1$ **(b)** ν is receiving in round $t_i + 1$

Figure 1 The adaptive decisions of the adversary in [25].

on this chain will be different depending on whether y_i is 0 or 1 — this serves to make the diameter of the dynamic network different when $\text{GDC} = 1$ and when $\text{GDC} = 0$, as discussed earlier. The difficulty for Alice, is that she does not know y_i, and hence does not know the exact set of edges on this chain. This prevents her from properly simulating those nodes that she need to simulate for this chain. Similar difficulty applies to Bob.

To overcome this difficulty, if a pair (x_i, y_i) is not $(0, 0)$, the adversary in [25] will make an adaptive decision for manipulating the edges on the i-th chain,[5] to help enable Alice (and also Bob) to simulate. The cycle promise already tells us that for given x_i (e.g., 0), there are two possibilities for y_i (e.g., 0 and 1). The adaptive decisions of the adversary will have the following end effects: Under the topology resulted from such adaptive decisions, the behavior of those nodes that Alice needs to simulate will depend only on x_i and no longer depend on y_i. A similar property holds for Bob.

The details on why those adaptive decisions can achieve such end effects are complex, and are related to the fundamental fact that a node does not know its neighbors in a round until it receives messages from them. At the same time, those details are entirely orthogonal to this work. Hence due to space limitations, we refer interested readers to [25] for such details. Here we will only describe the specifics of all the adaptive decisions made by the adversary, which is needed for our later discussion: Consider any i where (x_i, y_i) is not $(0, 0)$. At the beginning of round $t_i + 1$ where t_i is some function of x_i and y_i, the adversary examines the coin flip outcomes of $\mathcal{P}$ and determines whether the middle node ν on the i-th chain is sending or receiving in round $t_i + 1$ (see Figure 1). If ν is sending, the adversary removes a certain edge e that is incidental to ν, immediately in round $t_i + 1$. Otherwise the adversary will remove the edge e in round $t_i + 2$. Except these adaptive decisions, the adversary does not make any other adaptive decisions. In particular, the adversary does not need to make adaptive decisions for chains corresponding to $(0, 0)$.

6 Roadmap for Lower Bound Proof under Oblivious Adversaries

This section provides the intuition behind, and the roadmap for, our novel proof of Consensus lower bound under oblivious adversaries. To facilitate discussion, we define a few simple concepts. Consider the i-th chain in the previous section where (x_i, y_i) is not $(0, 0)$, and the middle node ν on the chain. We define binary random variable $\mathbf{z} = 0$ if ν is sending in round $t_i + 1$, and define $\mathbf{z} = 1$ otherwise. We use $\mathcal{A}'$ to denote the adaptive adversary described in the previous section. We define $\lambda_{\mathcal{A}'}$ to be the adaptive decision made by $\mathcal{A}'$, where $\mathcal{A}'$ removes the edge e in round $t_i + 1 + \lambda_{\mathcal{A}'}$. With these concepts, $\mathcal{A}'$ essentially sets its decision $\lambda_{\mathcal{A}'}$ to be $\lambda_{\mathcal{A}'} = \mathbf{z}$.

[5] In the actual proof, the adversary only needs to make adaptive decisions for a subset (usually a constant fraction) of such chains. But it is much easier to understand if we simply let the adversary make an adaptive decision on all of them. Doing so has no impact on the asymptotic results.

Making guesses. $\mathcal{A}'$ is adaptive since $\lambda_{\mathcal{A}'}$ depends on $\mathbf{z}$, and $\mathbf{z}$ in turn is a function of $\mathcal{P}$'s coin flips. An oblivious adversary $\mathcal{A}$ cannot have its decision $\lambda_{\mathcal{A}}$ depend on $\mathbf{z}$. At the highest level, our idea of allowing $\mathcal{A}$ in the reduction is simple: We let $\mathcal{A}$ make a blind guess on whether ν is sending. Specifically, imagine that $\mathcal{A}$ itself flips a fair coin $\mathbf{c}$, and then directly set its decision to be $\lambda_{\mathcal{A}} = \mathbf{c}$. Same as $\mathcal{A}'$, $\mathcal{A}$ still removes the edge e in round $t_i + 1 + \lambda_{\mathcal{A}}$, except that now $\lambda_{\mathcal{A}} = \mathbf{c}$. Some quick clarifications will help to avoid confusion here. First, such a guess $\mathbf{c}$ may be either correct (i.e., $\mathbf{c} = \mathbf{z}$) or wrong (i.e., $\mathbf{c} = \bar{\mathbf{z}}$). $\mathcal{A}$ itself cannot tell whether the guess is correct, since $\mathcal{A}$ (being oblivious) does not know $\mathbf{z}$. Alice and Bob, however, can tell if the guess is correct, because they are simulating both the protocol $\mathcal{P}$ and the adversary $\mathcal{A}$, and hence know both $\mathbf{z}$ and $\mathbf{c}$. But they cannot interfere with the guess even if they know it is wrong.

Now if the guess is correct, then the decision of $\mathcal{A}$ will be exactly the same as $\mathcal{A}'$, and everything will work out as before. But if the guess is wrong, then $\mathcal{A}$ can no longer enable Alice to simulate without knowing Y. More specifically, if the guess is wrong, then for the i-th chain, the behavior of those nodes that Alice needs to simulate will depend on the value of y_i, and Alice does not know y_i. To overcome this main obstacle, our key idea is to add a special *leaker* entity in the two-party CC problem, which should be viewed as an oracle that is separate from Alice and Bob. If the guess is wrong for the i-th chain, the leaker will disclose for free to Alice and Bob the pair (x_i, y_i). The knowledge of y_i then immediately enables Alice to infer the exact behavior of the nodes that she needs to simulate. Similar arguments apply to Bob.

Roadmap. There are two non-trivial technical issues remaining in the above approach: i) when to make guesses, and ii) how the leaker impacts the CC of GDC. Overcoming them will be the main tasks of Section 7 and 8, respectively. Section 9 will present our final CONSENSUS lower bound, whose lengthy and somewhat tedious proof is deferred to the full version [16] of this paper.

7 Sanitized Adaptive Adversaries

The difficulty. It turns out that it does not quite work for Alice and Bob to approach the leaker for help when they feel needed. Consider the following example $\text{GDC}_6^{2,4}$ instance with $X = 000000$ and $Y = 111100$. As explained in Section 5, the dynamic network corresponding to this instance has six chains. For all i, we say that the i-th chain is an "$|_b^a$ chain" if $x_i = a$ and $y_i = b$. The first four chains in the dynamic network are thus all $|_1^0$ chains, while the remaining two are $|_0^0$ chains. The adaptive adversary $\mathcal{A}'$ in [25] (see Section 5) will make adaptive decisions for all $|_1^0$ chains, but does not need to do so for $|_0^0$ chains. Applying the idea from Section 6, the oblivious adversary $\mathcal{A}$ should thus make guesses for those four $|_1^0$ chains. Note that $\mathcal{A}$ needs to be simulated by Alice and Bob. The difficulty is that Alice does not know for which chains a guess should be made, since she does not know which chains are $|_1^0$ chains. In fact if she knew, she would have already solved GDC in this instance. Similar arguments apply to Bob.

A naive fix is to simply make a guess for each of the six chains. Imagine now that the guess turns out to be wrong for the last chain, which is a $|_0^0$ chain. Alice and Bob will then ask the leaker to disclose (x_6, y_6). Such disclosure unfortunately directly reveals the answer to the GDC instance. This in turn, reduces the CC of GDC to 0, rendering the reduction meaningless. (Refusing to disclose (x_6, y_6) obviously does not work either, since the refusal itself reveals the answer.)

Our idea. To overcome this, we do not let Alice and Bob decide for which chains the adversary $\mathcal{A}$ should make a guess. Instead, we directly let our leaker decide which indices should be leaked: For every i where $(x_i, y_i) \neq (0, 0)$, the leaker leaks the pair (x_i, y_i) with half probability, to both Alice and Bob. In the earlier example, the leaker will leak each of the indices 1 through 4 independently with half probability.

For any given i, define binary random variable $\mathbf{s} = 1$ iff the leaker leaks index i. If $\mathbf{s} = 1$, then Alice and Bob will "fabricate" a wrong guess for the adversary $\mathcal{A}$ that they are simulating, so that the guess of $\mathcal{A}$ is wrong (and hence index i needs to be leaked). Specifically, Alice and Bob examine the coin flip outcomes of the protocol $\mathcal{P}$ to determine the value of $\mathbf{z}$, and then set the guess $\mathbf{c}$ of $\mathcal{A}$ to be $\mathbf{c} = \bar{\mathbf{z}}$. (Recall that $\mathbf{z}$ indicates whether the middle node is sending in round $t_i + 1$.) In such a case, the decision $\lambda_{\mathcal{A}}$ of $\mathcal{A}$ will be $\lambda_{\mathcal{A}} = \mathbf{c} = \bar{\mathbf{z}}$. On the other hand, if $\mathbf{s} = 0$ (meaning that index i is not leaked), then Alice and Bob let $\mathcal{A}$ behave exactly the same as the adaptive adversary $\mathcal{A}'$ in Section 5. In particular, if $\mathcal{A}'$ makes an adaptive decision $\lambda_{\mathcal{A}'} = \mathbf{z}$ for this chain, then the decision $\lambda_{\mathcal{A}}$ of $\mathcal{A}$ will also be $\lambda_{\mathcal{A}} = \mathbf{z}$ (i.e., as if $\mathcal{A}$ guessed correctly). Combining the two cases gives $\lambda_{\mathcal{A}} = \mathbf{z} \oplus \mathbf{s}$.

Obviously $\mathcal{A}$ here is no longer oblivious (since $\lambda_{\mathcal{A}}$ now depends on $\mathbf{z}$), which seems to defeat the whole purpose. Fortunately, this adaptive adversary $\mathcal{A}$ is special in the sense that all the adaptivity (i.e., $\mathbf{z}$) has been "sanitized" by taking XOR with the independent coin of $\mathbf{s}$. Intuitively, this prevents $\mathcal{A}$ from effectively adapting. The following discussion will formalize and prove that such an $\mathcal{A}$ is no more powerful than an oblivious adversary, in terms of incurring the cost of a protocol.

Formal framework and results. Without loss of generality, we assume that an adversary makes *binary decisions* that fully describe the behavior of the adversary. An adversary is *deterministic* if its decisions are fixed given the protocol's coin flip outcomes, otherwise it is *randomized*. Consider any deterministic adaptive adversary $\mathcal{A}'$. A decision $\lambda_{\mathcal{A}'}$ made by $\mathcal{A}'$ is called *adaptive* if $\lambda_{\mathcal{A}'}$ can be different under different coin flip outcomes of the protocol. A randomized adaptive adversary $\mathcal{A}$ is called a *sanitized version* of $\mathcal{A}'$, if $\mathcal{A}$ behaves the same as $\mathcal{A}'$ except that $\mathcal{A}$ *sanitizes* all adaptive decisions made by $\mathcal{A}'$ and also an arbitrary (possibly empty) subset of the non-adaptive decisions made by $\mathcal{A}'$. Here $\mathcal{A}$ *sanitizes* a decision $\lambda_{\mathcal{A}'}$ made by $\mathcal{A}'$ by setting its own decision $\lambda_{\mathcal{A}}$ to be $\lambda_{\mathcal{A}} = \lambda_{\mathcal{A}'} \oplus \mathbf{s}$, where $\mathbf{s}$ is a separate fair coin and is independent of all other coins. We also call the above $\mathcal{A}$ as a *sanitized adaptive adversary*. In our discussion above, $\lambda_{\mathcal{A}'} = \mathbf{z}$, while $\lambda_{\mathcal{A}} = \mathbf{z} \oplus \mathbf{s} = \lambda_{\mathcal{A}'} \oplus \mathbf{s}$. The following simple theorem, proven in the full version [16] of this paper, confirms that $\mathcal{A}$ is no more powerful than an oblivious adversary:

▶ **Theorem 3.** *Let* $\mathrm{cost}(\mathcal{P}, \mathcal{A}, C_{\mathcal{P}}, C_{\mathcal{A}})$ *be any deterministic function (which the adversary aims to maximize) of the protocol* $\mathcal{P}$*, the adversary* $\mathcal{A}$*, the coin flip outcomes* $C_{\mathcal{P}}$ *of* $\mathcal{P}$*, and the coin flip outcomes* $C_{\mathcal{A}}$ *(if any) that may also influence the behavior of* $\mathcal{A}$*. For any protocol* $\mathcal{P}$*, any deterministic adaptive adversary* $\mathcal{A}'$*, and its sanitized version* $\mathcal{A}$*, there exists a deterministic oblivious adversary* $\mathcal{B}$ *such that* $E_{\mathbf{C}_{\mathcal{P}}}[\mathrm{cost}(\mathcal{P}, \mathcal{B}, \mathbf{C}_{\mathcal{P}}, -)] \geq E_{\mathbf{C}_{\mathcal{P}}, \mathbf{C}_{\mathcal{A}}}[\mathrm{cost}(\mathcal{P}, \mathcal{A}, \mathbf{C}_{\mathcal{P}}, \mathbf{C}_{\mathcal{A}})]$*. Furthermore, for every* $C_{\mathcal{P}}$ *in the support of* $\mathbf{C}_{\mathcal{P}}$*, there exists* $C_{\mathcal{A}}$ *in the support of* $\mathbf{C}_{\mathcal{A}}$*, such that* $\mathcal{B}$*'s decisions are exactly the same as the decisions made by* $\mathcal{A}$ *under* $C_{\mathcal{P}}$ *and* $C_{\mathcal{A}}$*.*

Summary of this section. Recall that $\mathcal{A}'$ denotes the adaptive adversary used in [25] and reviewed in Section 5. Based on the discussion in this section, our reduction from GDC (with a leaker) to CONSENSUS will use a sanitized adaptive adversary $\mathcal{A}$ for the dynamic network. $\mathcal{A}$ behaves exactly the same as $\mathcal{A}'$ except: For each i-th chain where $\mathcal{A}'$ makes an adaptive decision $\lambda_{\mathcal{A}'}$ for that chain, $\mathcal{A}$ sets its own decision $\lambda_{\mathcal{A}}$ for that chain to be

$\lambda_{\mathcal{A}} = \lambda_{\mathcal{A}'} \oplus \mathbf{s}$. Here $\mathbf{s}$ denotes whether index i is leaked by the leaker. Theorem 3 confirms that the consensus protocol $\mathcal{P}$'s end guarantees, even though $\mathcal{P}$ was designed to work against oblivious adversaries instead of adaptive adversaries, will continue to hold under $\mathcal{A}$.

8 Communication Complexity with The Leaker

To get our final Consensus lower bound , the next key step is to prove a lower bound on the CC of Gdc with the leaker. At first thought, one may think that having the leaker will not affect the CC of Gdc much, since i) the leakable pairs have no impact on the answer to the problem and are hence "dummy" parts, and ii) the leaker only leaks about half of such "dummy" parts. As a perhaps surprising example, Lemma 1 in the full version [16] of this paper shows that having the leaker reduces the CC of $\text{Gdc}_n^{16\sqrt{n}\ln\frac{1}{\delta},2}$ from $\Omega(\sqrt{n})$ to 0. This implies that the impact of the leaker is more subtle than expected. In particular, without a careful investigation, it is not even clear whether the CC of Gdc with our leaker is large enough to translate to our intended $\Omega(d + \text{poly}(m))$ lower bound on Consensus.

This section will thus do a careful investigation and eventually establish a formal connection between the CC with the leaker ($\mathfrak{L}_\delta$) and the CC without the leaker ($\mathfrak{R}_\delta$):

▶ **Theorem 4.** *For any constant $\delta \in (0, \frac{1}{2})$, there exist constants $c_1 > 0$ and $c_2 > 0$ such that for all n, g, q, and $n' = c_2\sqrt{n}/(q^{1.5}\log q)$, $\mathfrak{L}_\delta(\text{Gdc}_n^{g,q}) \geq c_1\mathfrak{R}_\delta(\text{Gdc}_{n'}^{g,q})$.*

Later we will see that the lower bound on Gdc with our leaker as obtained in the above theorem (combined with Theorem 2) is sufficient for us to get a final $\Omega(d + \text{poly}(m))$ lower bound on Consensus. The theorem actually also holds for many other problems beyond Gdc, though we do not present the general form here due to space limitations.

8.1 Our Approach and Key Ideas

While we will only need to prove Theorem 4 for Gdc, we will consider general two-party problem Π, since the specifics of Gdc are not needed here. We will prove Theorem 4 via a reduction: We will construct a protocol $\mathcal{Q}$ for solving $\Pi_{n'}$ without the leaker, by using an oracle protocol $\mathcal{P}$ for solving Π_n with the leaker, where n' is some value that is smaller than n. Such a reduction will then lead to $\mathfrak{R}_{\delta'}(\Pi_{n'}) = O(\mathfrak{L}_\delta(\Pi_n))$.

We will call each kind of leakable pairs as a *leakable pattern*. For example, $\text{Gdc}_n^{1,2}$ has leakable patterns of $(1,1)$, $(0,1)$, and $(1,0)$. Note that leakable patterns are determined by the problem Π and not by an instance of the problem. We use $k \in [0, q^2]$ to denote the total number of leakable patterns for Π whose inputs are q-ary strings. For $\text{Gdc}_n^{g,q}$, $k = 2q - 1$.

Simulating the leaker via padded pairs. The central difficulty in the reduction is that Alice and Bob running $\mathcal{Q}$ need to simulate the leaker, in order to invoke the oracle protocol $\mathcal{P}$. (Note that $\mathcal{P}$ here is the two-party protocol, and has nothing to do with the Consensus protocol.) This is difficult because each party only knows her/his own input. Our first step to overcome this difficulty is to pad known characters to the inputs and then leak *only* those padded characters, as explained next.

Let (X', Y') be the given input to $\mathcal{Q}$. Assume for simplicity that $(2,1)$ is the only leakable pattern in Π, and consider the problem instance in Figure 2 where $X' = 02$ and $Y' = 01$. Alice and Bob will append/pad a certain number of occurrences of each leakable pattern to (X', Y'). Let (X, Y) denote the resulting strings after the padding. In the example in Figure 2, Alice and Bob append 1 occurrence of $(2,1)$ to (X', Y') — or more specifically,

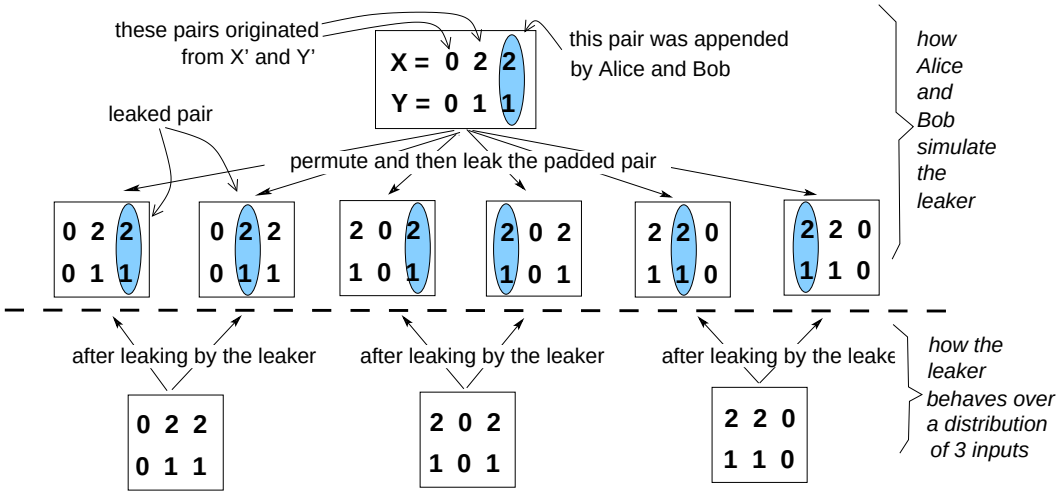


Figure 2 How padding and permutation enable Alice and Bob to simulate the leaker. In this example $X' = 02$, $Y' = 01$, $X = 022$, and $Y = 011$. Here to help understanding, we assume that the leaker leaks *exactly* half of all the leakable pairs.

Alice appends 2 to X' and Bob appends 1 to Y'. Doing so gives $X = 022$ and $Y = 011$. Note that doing so does not involve any communication, since the leakable patterns are publicly known. Imagine that Alice and Bob now invoke $\mathcal{P}$ using (X, Y), where $X = 022$ and $Y = 011$. Note that the two-party protocol $\mathcal{P}$ assumes the help from our leaker. Alice and Bob can easily simulate the leaking of (x_3, y_3), since (x_3, y_3) is the padded pair and they both know that the pair is exactly $(2, 1)$. However, (x_2, y_2) is also a leakable pair. Alice and Bob still cannot simulate the leaking of this pair, since this pair originated from (X', Y') and they do not know the value of this pair.

To overcome this, Alice and Bob use public coins to generate a random permutation, and then use the permutation to permute X and Y, respectively (Figure 2). This step does not involve communication. For certain problems Π (e.g., for GDC), one can easily verify that such permutation will not affect the answer to Π. Such permutation produces an interesting effect, as illustrated in Figure 2. The upper part of Figure 2 plots the 6 possible outcomes after the permutation, for our earlier example of $X = 022$ and $Y = 011$. Before the permutation, the last pair in (X, Y) is a padded pair. Imagine that Alice and Bob leak this pair. Now after the permutation, this leaked pair will occupy different indices in the 6 outcomes of the permutation.

The bottom part of Figure 2 illustrates the (real) leaker's behavior over certain inputs. To help understanding, assume here for simplicity that the leaker leaks *exactly* half of all the leakable pairs. Now consider 3 different inputs $(022, 011)$, $(202, 101)$, and $(220, 110)$. One can see that the behavior of the leaker over these 3 inputs (see Figure 2) exactly matches the result of permutation as done by Alice and Bob. Hence when Alice and Bob feed the result of the permutation into $\mathcal{P}$ while leaking the padded pair, it is as if $\mathcal{P}$ were invoked over the previous 3 inputs (each chosen with $1/3$ probability) together with the real leaker. This means that $\mathcal{P}$'s correctness and CC guarantees should continue to hold, when Alice and Bob invoke $\mathcal{P}$ while leaking only the padded pair.

How many pairs to leak. Imagine that (X', Y') contain o pairs of $(2, 1)$, and Alice and Bob pad p pairs of $(2, 1)$ to (X', Y'). The result of the padding, (X, Y), will contain $o + p$ pairs of $(2, 1)$. Let $\mathbf{f}$ be the number of $(2, 1)$ pairs in (X, Y) that should be leaked, which

Protocol 1: Our δ'-error protocol $\mathcal{Q}$ for solving $\Pi_{n'}$ without our leaker. $\mathcal{Q}$ invokes the δ-error oracle two-party protocol $\mathcal{P}$ that solves Π_n with our leaker. The following only shows Alice's part of $\mathcal{Q}$. Bob's part of $\mathcal{Q}$ can be obtained similarly.

Input : X', n, n', δ, δ', where $\delta < \delta'$

1 $s \leftarrow \frac{\mathfrak{R}_{\delta'}(\Pi_{n'})}{4 \log q}$; **foreach** $j = 1, \ldots, k$ **do** $\mathbf{v}_j \leftarrow 0$;

2 **repeat** s **times do**

3 draw a uniformly random integer $i \in [1, n']$ using public coins;

4 send x'_i to Bob and receive y'_i from Bob ;

5 **foreach** $j = 1, \ldots, k$ **do if** (x'_i, y'_i) *equals the* j*-th leakable pattern* **then** $\mathbf{v}_j \leftarrow \mathbf{v}_j + \frac{n'}{s}$;

6 **end**

/*** Here h_j is the number of times that the j-th leakable pattern is padded to (X', Y'). ***/

7 $h \leftarrow 2n' + \frac{500}{(\delta'-\delta)^2}(k^2 + \frac{kn'^2}{2s} \ln \frac{24k}{\delta'-\delta})$;

8 **foreach** $j = 1, \ldots, k-1$ **do** $h_j \leftarrow h$;

9 $h_k \leftarrow n - n' - (k-1)h$; **if** $h_k < h$ **then** generate an arbitrary output and exit;

10 **foreach** $j = 1, \ldots, k$ **do**

11 draw an integer $\mathbf{b}_j$ from the binomial distribution $\mathbb{B}(\frac{h_j + \mathbf{v}_j}{2})$ using public coins ;
// $\mathbb{B}(\mu)$ is the distribution for the number of heads obtained when flipping 2μ fair coins.

12 **if** $\mathbf{b}_j > h_j$ **then** $\mathbf{b}_j \leftarrow h_j$;

13 let (a, b) be the j-th leakable pattern ;

14 append h_j copies of a to X', and flag the first $\mathbf{b}_j$ indices of these h_j indices as "to be leaked";

15 **end**

16 generate a uniformly random permutation $\mathbf{M}$ using public coins;

17 $\mathbf{X} \leftarrow \mathbf{M}(X')$ /* the flags in X' will be treated as part of X' and be permuted as well. */;

18 invoke $\mathcal{P}$ (together with the other party) using $\mathbf{X}$ as input, while leaking all those indices that are flagged, until either $\mathcal{P}$ outputs or $\mathcal{P}$ has incurred $(\frac{6}{\delta'-\delta})cc(\mathcal{P})$ bits of communication ;
/* when leaking index i, both x'_i and y'_i will be given to $\mathcal{P}$ — this can be done since a leaked index here must correspond to a padded pair at Line 14 */;

19 **if** $\mathcal{P}$ *has incurred* $(\frac{6}{\delta'-\delta})cc(\mathcal{P})$ *bits of communication* **then** exit with an arbitrary output ;

20 **else** output $\mathcal{P}$'s output and exit ;

obviously follows a binomial distribution with a mean of $\frac{o+p}{2}$. Ideally, Alice and Bob should draw $\mathbf{f}$ from the above binomial distribution, and then simulate the leaking of $\mathbf{f}$ pairs of $(2, 1)$. (They can do so as long as $\mathbf{f} \leq p$ — with proper p, we easily throw $\Pr[\mathbf{f} > p]$ into the error.) The difficulty, however, is that Alice and Bob do not know o, and hence cannot draw $\mathbf{f}$ with the correct mean of $\frac{o+p}{2}$.

To overcome this, Alice and Bob will estimate the value of o by sampling: For each sample, they use public coin to choose a uniformly random $i \in [1, n']$, and then send each other the values of x'_i and y'_i. They will spend total $\frac{\mathfrak{R}_{\delta'}(\Pi_{n'})}{2}$ bits for doing this, so that such sampling is effectively "free" and does not impact the asymptotic quality of the reduction. Alice and Bob will nevertheless still not obtain the exact value of o. This means that the distribution they use to draw $\mathbf{f}$ will be different from the distribution that the (real) leaker uses. Our formal proof will carefully take into account such discrepancy.

8.2 Formal Reduction and Final Guarantees

Pseudo-code. Protocol 1 presents the protocol $\mathcal{Q}$ for solving $\Pi_{n'}$ without our leaker, as run by Alice. $\mathcal{Q}$ internally invokes the oracle two-party protocol $\mathcal{P}$, where $\mathcal{P}$ solves Π_n with our leaker. At Line 1–6, Alice and Bob first exchange sampled indices to estimate the occurrences of each leakable pattern. Next Line 7–9 calculate the amount of padding needed. Line 10–15 do the actual padding, and then for each leakable pattern, flag a certain number of padded pairs as "to be leaked". At Line 16–20, Alice and Bob do a random permutation to obtain $(\mathbf{X}, \mathbf{Y})$, and then invoke $\mathcal{P}$ on $(\mathbf{X}, \mathbf{Y})$ while leaking all those flagged pairs.

Final properties of $\mathcal{Q}$. The full version [16] of this paper will prove that $\mathcal{Q}$ solves Π without our leaker, with an error of $\delta + \frac{11}{12}(\delta' - \delta)$, while incurring $\frac{\mathfrak{R}_{\delta'}(\Pi_{n'})}{2} + 5.5\mathrm{cc}(\mathcal{P})$ bits of communication. This will eventually lead to the proof of Theorem 4 (see the full version [16] of this paper).

9 Consensus Lower Bound under Oblivious Adversaries

Following is our final theorem on Consensus under oblivious adversaries:

▶ **Theorem 5.** *If the nodes only know a poor estimate m' for m such that $|\frac{m'-m}{m}|$ reaches $\frac{1}{3}$ or above, then a $\frac{1}{10}$-error* Consensus *protocol for dynamic networks with oblivious adversaries must have a time complexity of $\Omega(d + m^{\frac{1}{12}})$ rounds.*

Our proof under oblivious adversaries partly builds upon the previous proof under adaptive adversaries [25], as reviewed in Section 5. The key difference is that we reduce from Gdc *with our leaker* to Consensus. The complete proof is lengthy and tedious as it needs to build upon the lengthy proof in [25]. Since Section 7 and 8 already discussed the key differences between our proof and [25], we leave the full proof to the full version [16] of this paper, and only provide an overview here on how to put the pieces from Section 7 and 8 together.

Consider any oracle Consensus protocol $\mathcal{P}$ with $\frac{1}{10}$ error. Let $\mathrm{tc}(d, m)$ denote $\mathcal{P}$'s time complexity when running over dynamic networks controlled by oblivious adversaries and with d diameter and m nodes. As explained in Section 5, the crux will be to prove $\mathrm{tc}(8, m) \geq m^{\frac{1}{12}}$. To do so, we consider $\mathrm{Gdc}_n^{g,q}$ with $n = \frac{m-4}{3}$, $q = 20\mathrm{tc}(8, m) + 20$, and $g = 15q \ln q$. To solve $\mathrm{Gdc}_n^{g,q}(X, Y)$, Alice and Bob simulate $\mathcal{P}$ in the following way: In the simulation, the input (X, Y) is mapped to a *sanitized* adaptive adversary $\mathcal{A}$ that determines the topology of the dynamic network. Roughly speaking, if $\mathrm{Gdc}_n^{g,q}(X, Y) = 1$, the resulting dynamic network will have a diameter of 8. Even though $\mathcal{A}$ is an adaptive adversary, by Theorem 3 in Section 7, $\mathcal{P}$'s time complexity should remain $\mathrm{tc}(d, m)$ under $\mathcal{A}$. Hence $\mathcal{P}$ should decide within $\mathrm{tc}(8, m)$ rounds on expectation. If $\mathrm{Gdc}_n^{g,q}(X, Y) = 0$, then the resulting dynamic network will have a diameter of $\Theta(q)$. For $\mathcal{P}$ to decide in this dynamic network, we prove that it takes at least roughly $\frac{q}{2}$ rounds. Note that $\frac{q}{2} > 10\mathrm{tc}(8, m)$ — in other words, it takes longer for $\mathcal{P}$ to decide if $\mathrm{Gdc}_n^{g,q}(X, Y) = 0$. Alice and Bob do not know the other party's input, and hence does not have full knowledge of the dynamic network. But techniques from [25], together with the help from our leaker, enable them to still properly simulate $\mathcal{P}$'s execution. Finally, if $\mathcal{P}$ decides within $10\mathrm{tc}(8, m)$ rounds, Alice and Bob claim that $\mathrm{Gdc}_n^{g,q}(X, Y) = 1$. Otherwise they claim $\mathrm{Gdc}_n^{g,q}(X, Y) = 0$. Our proof will show that to solve $\mathrm{Gdc}_n^{g,q}$ with our leaker, using the above simulation, Alice and Bob incur $\Theta(\mathrm{tc}(8, m) \cdot \log n)$ bits of communication. We thus have $\Theta(\mathrm{tc}(8, m) \log n) \geq \mathfrak{L}_\delta(\mathrm{Gdc}_n^{g,q})$. Together with the lower bound on $\mathfrak{L}_\delta(\mathrm{Gdc}_n^{g,q})$ from Theorem 4 in Section 8 (and Theorem 2 in Section 4), this will lead to a lower bound on $\mathrm{tc}(8, m)$.

References

1 J. Augustine, C. Avin, M. Liaee, G. Pandurangan, and R. Rajaraman. Information spreading in dynamic networks under oblivious adversaries. In *DISC*, 2016.

2 J. Augustine, T. Kulkarni, P. Nakhe, and P. Robinson. Robust leader election in a fast-changing world. In *Workshop on Foundations of Mobile Computing*, 2013.

3 J. Augustine, G. Pandurangan, and P. Robinson. Fast byzantine leader election in dynamic networks. In *DISC*, October 2015.

4 John Augustine, Gopal Pandurangan, and Peter Robinson. Fast byzantine agreement in dynamic networks. In *PODC*, July 2013.

5 Z. Bar-Yossef, T. S. Jayram, R. Kumar, and D. Sivakumar. An information statistics approach to data stream and communication complexity. *Journal of Computer and System Sciences*, 68(4):702–732, June 2004.

6 Q. Bramas, T. Masuzawa, and S. Tixeuil. Distributed online data aggregation in dynamic graphs. In *ICDCS*, 2016.

7 M. Braverman. Interactive information complexity. *SIAM Journal on Computing*, 44(6):1698–1739, 2015.

8 K. Censor-Hillel, E. Haramaty, and Z. Karnin. Optimal dynamic distributed MIS. In *PODC*, 2016.

9 Binbin Chen, Haifeng Yu, Yuda Zhao, and Phillip B. Gibbons. The cost of fault tolerance in multi-party communication complexity. *Journal of the ACM*, 61(3), May 2014.

10 Alejandro Cornejo, Seth Gilbert, and Calvin Newport. Aggregation in dynamic networks. In *PODC*, July 2012.

11 E. Coulouma, E. Godard, and J. Peters. A characterization of oblivious message adversaries for which consensus is solvable. *Theoretical Computer Science*, 584:80–90, June 2015.

12 A. Das Sarma, S. Holzer, L. Kor, A. Korman, D. Nanongkai, G. Pandurangan, D. Peleg, and R. Wattenhofer. Distributed verification and hardness of distributed approximation. In *STOC*, 2011.

13 C. Dutta, G. Pandurangan, R. Rajaraman, Z. Sun, and E. Viola. On the complexity of information spreading in dynamic networks. In *SODA*, January 2013.

14 Mohsen Ghaffari, Nancy Lynch, and Calvin Newport. The cost of radio network broadcast for different models of unreliable links. In *PODC*, July 2013.

15 R. Ingram, P. Shields, and J. Walter. An asynchronous leader election algorithm for dynamic networks. In *IPDPS*, 2009.

16 I. Jahja, H. Yu, and Y. Zhao. Some lower bounds in dynamic networks with oblivious adversaries. Technical Report TRA7/17, School of Computing, National University of Singapore, July 2017. Also available at `http://www.comp.nus.edu.sg/~yuhf/oblivious-disc17-technicalreport.pdf`.

17 M. Konig and R. Wattenhofer. On local fixing. In *OPODIS*, 2013.

18 Fabian Kuhn, Nancy Lynch, Calvin Newport, Rotem Oshman, and Andrea Richa. Broadcasting in unreliable radio networks. In *PODC*, July 2010.

19 Fabian Kuhn, Nancy Lynch, and Rotem Oshman. Distributed computation in dynamic networks. In *STOC*, June 2010.

20 Fabian Kuhn, Yoram Moses, and Rotem Oshman. Coordinated consensus in dynamic networks. In *PODC*, June 2011.

21 Fabian Kuhn and Rotem Oshman. The complexity of data aggregation in directed networks. In *DISC*, September 2011.

22 Fabian Kuhn and Rotem Oshman. Dynamic networks: Models and algorithms. *SIGACT News*, 42(1):82–96, March 2011.

23 U. Schmid, B. Weiss, and I. Keidar. Impossibility results and lower bounds for consensus under link failures. *SIAM Journal on Computing*, 38(5):1912–1951, 2009.

24 O. Weinstein. Information complexity and the quest for interactive compression. *SIGACT News*, 46(2):41–64, June 2015.

25 H. Yu, Y. Zhao, and I. Jahja. The cost of unknown diameter in dynamic networks. In *SPAA*, July 2016. (Journal version under submission).

Recoverable FCFS Mutual Exclusion with Wait-Free Recovery

Prasad Jayanti[*1] **and Anup Joshi**[†2]

1 **6211 Sudikoff Lab for Computer Science, Dartmouth College, Hanover, NH, USA**
prasad@cs.dartmouth.edu

2 **6211 Sudikoff Lab for Computer Science, Dartmouth College, Hanover, NH, USA**
anupj@cs.dartmouth.edu

Abstract

Traditional mutual exclusion locks are not resilient to failures: if there is a power outage, the memory is wiped out. Thus, when the system comes back on, the lock will have to be restored to the initial state, i.e., all processes are rolled back to the Remainder section and all variables are reset to their initial values. Recently, Golab and Ramaraju showed that we can improve this state of the art by exploiting the Non-Volatile RAM (NVRAM). They designed algorithms that, by maintaining shared variables in NVRAM, allow processes to recover from crashes on their own without a need for a global reset, even though a crash can wipe out the local memory of a process.

We present a Recoverable Mutual Exclusion algorithm using the commonly supported CAS primitive. The main features of our algorithm are that it satisfies FCFS, it ensures that each process recovers in a wait-free manner, and in the absence of failures, it guarantees a worst-case Remote Memory Reference (RMR) complexity of $O(\lg n)$ on both Cache Coherent (CC) and Distributed Shared Memory (DSM) machines, where n is the number of processes for which the algorithm is designed. This bound matches the $\Omega(\lg n)$ RMR lower bound by Attiya, Hendler, and Woelfel for Mutual Exclusion algorithms that use comparison primitives.

1998 ACM Subject Classification D.1.3 Concurrent Programming

Keywords and phrases concurrent algorithm, synchronization, mutual exclusion, recovery, fault tolerance, non-volatile main memory, shared memory, multi-core algorithms

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.30

1 Introduction

Recent research has focused on exploiting non-volatile main memory to design algorithms that can tolerate process crashes. The underlying idea of these algorithms is that upon a crash a process executes a recovery code that consults the shared state stored in the non-volatile main memory and brings the system back to a usable state. Due to an explicit recovery procedure to emerge out of a crash, these algorithms are called *recoverable* algorithms. We present a recoverable algorithm for the *FCFS Mutual Exclusion* problem.

* The first author is grateful for the generous support of James Frank Family Professorship.
† The second author is grateful for the support of Dartmouth Fellowship.

© Prasad Jayanti and Anup Joshi;
licensed under Creative Commons License CC-BY

Let us recall the traditional (non-recoverable) mutual exclusion problem [2], which captures the requirements of a system of processes that share an exclusive resource – a resource that can be accessed by only one process at a time. Specifically, we consider a system consisting of $n > 1$ asynchronous processes, where each process cycles through four sections of code – Remainder, Try, Critical Section (CS), and Exit. Initially, all processes are in the Remainder section. When a process gets interested in accessing the resource, it executes the Try section, where it competes with other processes for exclusive access to the resource. When the Try section terminates, the process moves to CS, where it actually accesses the resource. When the process no longer needs the resource, it executes the Exit section to relinquish its access to the resource. A standard version of the *mutual exclusion problem* is to come up with code for the Try and Exit sections so that the following properties are ensured:

(i) *mutual exclusion:* at most one process is in the CS at any time;

(ii) *bounded exit:* every process completes the exit section in a bounded number of its steps; and

(iii) *starvation freedom:* if a process is in the Try section, it eventually enters the CS (under the assumptions that every process that enters the CS eventually leaves the CS, and no process permanently stops taking steps while in the Try or Exit sections).

We could additionally require the fairness property FCFS [8] which, informally, states that processes enter the CS in the order in which they request the CS. A *mutual exclusion lock* is a solution to the mutual exclusion problem.

A traditional mutual exclusion lock cannot gracefully tolerate failures. In particular, if a process crashes, its memory – including its Program Counter – can lose their contents. As a result, when a process restarts after a crash, it cannot tell where in the algorithm it had failed and from where and how to resume its execution. Thus, process crashes can render a lock permanently unusable. One solution would be a global reset: to recover from a crash, roll back all processes to the Remainder section and set all variables to their initial values. There are two disadvantages to such a global reset. First, if the crash occurs while a process is in the CS and while the data structures manipulated in the CS are in an inconsistent state, the inconsistency will persist even after the reset. Second, a local failure (e.g., the crash of a single process) causes the reset of the entire system, which can be unacceptable in very large systems where the probability that some process or the other crashes is non-negligible.

It would be ideal if the lock can be designed so that the failure of some processes do not affect other processes and even the failed processes, when they eventually restart, can proceed as if they had never failed. Golab and Ramaraju's recoverable mutual exclusion algorithm [4] shed some light on how one might realize this goal by cleverly exploiting hybrid architectures [10] that use NVRAM (nonvolatile random access memory) technology for shared memory. When a process crashes, all its local variables are wiped out. So, when a process restarts after a crash, the general idea for an algorithm is that the process could consult the non-volatile shared memory to reconstruct its lost state and then resume its execution of the algorithm from the point where it crashed, making it seem like the crash never occurred. For instance, if a process crashes while in the CS, after restart a recovery procedure could put the process right back in the CS, making the crash appear simply like a long delay.

Ensuring such seamless recovery while also guaranteeing efficiency is where the intellectual challenge lies. At first glance it might seem that if each process stores its state in the non-volatile memory after every step, then when restarting after a crash, the process should be able to easily restore its state, thereby rendering the design of a recoverable algorithm easy. This strategy however falls short because, even with such a laborious storing of its state, when a process attempts to restore its state after a crash, if the stored value for its

program counter is k, it cannot distinguish whether it had crashed before or after executing the instruction at k. This seemingly small ambiguity makes the design of an algorithm challenging. To appreciate this fact, a reader familiar with the MCS lock should imagine the plight of a recovering process that crashed right after the initial swap operation.

We present an implementation of a recoverable lock using CAS that has the following merits:

(i) it satisfies FCFS,

(ii) its recovery is wait-free, and

(iii) it has a Remote Memory Reference (RMR) complexity of $O(f + \lg n)$ on both Cache Coherent (CC) and Distributed Shared Memory (DSM) machines, where n is the number of processes for which the lock is designed and f is the number of times a process crashes between the time it invokes and exits the lock.

The wait-free recovery of our algorithm ensures that when a process that crashes in the CS subsequently restarts, it gets back into the CS in a bounded number of its own steps, i.e., without any waiting whatsoever, regardless of however many other processes have failed or are slow. Attiya, Hendler, and Woelfel proved a lower bound that the RMR complexity is $\Omega(\lg n)$ for even a non-recoverable CAS-based lock [1]. Thus, our algorithm adds only $O(1)$ RMRs per crash. It would be interesting to resolve if the $O(f + \lg n)$ complexity of our algorithm is optimal.

In comparison, Golab and Ramaraju's locks [4] do not satisfy FCFS and, except for their tournament based algorithm, they do not have wait-free recovery. Ramaraju's lock [11] satisfies FCFS, but it works only on CC machines and uses the memory-to-memory swap primitive, which is not supported on any of the current multiprocessors. Golab and Hendler adapt the MCS lock in three different ways to implement recoverable locks. Two of their locks, although not claimed in their paper, appear to satisfy FCFS. One of these two works only on CC machines; further, even a process that crashes just once can incur $\Omega(n)$ RMRs in the worst case (but on the positive side a process that does not crash incurs at most $O(1)$ RMRs). The other algorithm of theirs that satisfies FCFS uses a double word primitive, which is not supported on any of the current multiprocessors.

Recoverable algorithms are typically designed by adapting existing (non-recoverable) mutual exclusion algorithms. Golab and Ramaraju's recoverable algorithms [4] adapt Yang and Anderson's tournament algorithm [12]. Ramaraju's [11] and Golab and Hendler's recoverable algorithms [3] adapt Mellor-Crummey and Scott's algorithm [9]. Our algorithm in this paper adapts Jayanti's mutual exclusion algorithm [7].

2 Model and Specification

The system consists of n processes named $1, 2, \ldots, n$ and atomic shared variables that support *read*, *write*, and *compare&swap* operations. Each process has five sections of code – Remainder, Recover, Try, CS, and Exit. A *recoverable mutual exclusion algorithm* specifies the code for Recover, Try, and Exit sections of all processes, and the initial values for all local and shared variables. We make no assumptions about the Remainder section and CS other than that none of the shared and local variables of the mutual exclusion algorithm are modified by these sections. All processes are initially in the Remainder section.

2.1 Configuration and step

A *configuration* of the system is specified by the values of all shared variables and the states of the n processes, where the state of a process p is in turn specified by the value of PC_p

(p's program counter) and the values of p's local variables. The configuration changes when a process takes a step. There are two types of steps that a process can take – *normal step* or *crash step* – explained as follows.

- A *normal step by a process p from a configuration C* causes p to perform the instruction that PC_p points to in C. We assume that in a normal step p performs a single operation on a single shared variable, and p's state changes based on the value returned by the shared variable. A special case occurs if p is in the Remainder section: if p is in the Remainder section in a configuration C, a normal step by p from C causes p to transfer control to the Recover section and execute the first instruction of the Recover section. The flow of control from other sections is modeled as usual. If p is in the Try section, after a normal step p either remains in Try or moves to CS. If p is in the CS, after a normal step p moves to the Exit section. If p is in the Exit section, after a normal step p either remains in Exit or moves to the Remainder section. If p is in the Recover section, after a normal step p can be in any of the sections, and the exact rules are specified in Section 2.4.
- A crash step models a process crash. We only model process crashes that occur outside the Remainder section. Specifically, if a process p is in the Recover, Try, CS, or Exit sections in a configuration C, then a *crash step by p from C* sets PC_p to the Remainder section and sets all the local variables of p to arbitrary values. (The crash step does not affect any shared variables since they are assumed to reside in the non-volatile memory, which is unaffected by the crash failures of processes.)

2.2 Execution and Attempt

From the above, we see that a step is determined by which process takes the step and whether the step is normal or crash. Thus, a *step* is an element of $\{1, 2, \ldots, n\} \times \{\text{normal}, \text{crash}\}$. A *schedule* is any finite or infinite sequence of steps. An *execution* corresponding to a schedule $\sigma = s_1, s_2, \ldots$ is $C_0, s_1, C_1, s_2, C_2, \ldots$, where C_0 is the initial configuration specified by the mutual exclusion algorithm, C_1 is the configuration after step s_1, C_2 is the configuration after steps s_1 and s_2, and so on.

Let E be an execution and s be a step by a process p from a configuration C in E. We say *p initiates an attempt in step s* if p is in the Remainder section in C and p's latest step before s is a normal step. We say *p completes an attempt in step s'* if s' is a normal step by p that moves p to the Remainder section. An *attempt* by p in E is a fragment of E that starts with an attempt initiation step s by p and ends with p's earliest attempt completion step s' that follows s. We say *p is active in a configuration C* if C occurs in an attempt by p. It is important to note that p might visit the Remainder section multiple times during an attempt because of its crash steps; thus, p can be active even when it is in the Remainder section.

2.3 Basic properties

Mutual exclusion, bounded exit, and starvation-freedom are properties normally required of any algorithm. The last two properties require suitable adaptation from how they are normally stated for the failure-free setting.

Mutual Exclusion: *An algorithm satisfies Mutual Exclusion* if at most one process is in the CS in every configuration of every execution.

Bounded Exit: The bounded exit property stipulates that a process be able to relinquish its access to the CS without being obstructed by other processes. Formally, *an algorithm satisfies Bounded Exit* if there is an integer b such that, for all executions E and for all

processes p, if p is in the Exit section in any configuration in E and the subsequent steps of p in E are normal steps, then p moves to the Remainder section in at most b of its own steps.

Starvation Freedom: Intuitively, starvation-freedom requires that every process that initiates an attempt eventually enters the CS. However, it is impossible to satisfy this condition unless each process that enters the CS eventually gives up the CS, no process fails infinitely many times in an attempt, and no active process permanently stops taking steps. The last condition means that if a process crashes, it must eventually restart and perform normal steps. We formalize starvation-freedom as follows.

An execution E is *fair* if, for all processes p, we have:

(i) if p is in the CS and does not crash while there, then p subsequently enters the Exit section;

(ii) p has only a finite number of crash steps in any one attempt; and

(iii) if p initiates an attempt then either p completes that attempt or p has an infinite number of normal steps.

An algorithm satisfies Starvation Freedom if in every fair execution every process that initiates an attempt enters the CS in that attempt.

2.4 Well-formedness: how control transfers upon crash and restart

We know from the definition of a normal step that control moves from the Remainder section to the Recover section when a process initiates a new attempt or when a process restarts after a crash during an ongoing attempt. Control moves out of the Recover section either because the process crashes (and the crash step moves the control to Remainder) or because the process eventually completes the Recover section at some normal step. In the latter case, where control moves to is governed by the following expectations:

(i) a process must enter the CS in each of its attempts,

(ii) if a process enters the Recover section because of a crash in the CS, then the Recover section will put the process right back in the CS, and

(iii) if a process enters the Recover section because of a crash in the Exit section, the Recover section can put the process back in the Exit section, bypassing the Try and CS.

More specifically, let s be a normal step by p in which p completes the Recover section, and s' be the latest step by p before s in which p initiates an attempt or p crashes outside of the Recover section (i.e., in Try, CS, or Exit). Then, the rules for where the control moves to after step s are as follows:

- If s' is an attempt initiation step, then s moves control to Try section or CS.
  Informally, if p enters the Recover section from the Remainder section due to an attempt initiation step, when control transfers out of the Recover section eventually due to a normal step, it transfers to Try section or CS.
- If s' is a crash step while p is in Try section, then s moves control to Try section or CS.
  Informally, if p crashes in the Try section and subsequently enters the Recover section, when control transfers out of the Recover section eventually due to a normal step, it transfers to Try section or CS.
- If s' is a crash step while p is in CS, then s moves control to CS.
  Informally, if p crashes in the CS and subsequently enters the Recover section, when control transfers out of the Recover section eventually due to a normal step, it transfers to CS.
- If s' is a crash step while p is in Exit, then s moves control to CS, Exit section, or Remainder section.

Informally, if p crashes in the Exit section and subsequently enters the Recover section, when control transfers out of the Recover section eventually due to a normal step, it transfers to CS, Exit section, or Remainder section.

An execution E is *well-formed* if all of the above rules are met in E. An algorithm satisfies *Well-formedness* if every execution of the algorithm is well-formed.

2.5 Critical Section Reentry

Suppose that a process p crashes while in the CS. Well-formedness guarantees that, once p restarts, Recover section puts p back in the CS. Since the data structures that p was manipulating in the CS could be in an inconsistent state at the time of p's crash, it is important that no other process visits the CS between p's crash in the CS and its subsequent reentry into the CS. Golab and Ramaraju captured this requirement through a property that they called *Critical Section Reentry (CSR)*, stated as follows.

An algorithm satisfies Critical Section Reentry if, for all executions E, if a process p crashes inside the CS, then no other process enters the CS before p reenters the CS.

2.6 Wait-free Recovery

Intuitively, the purpose of the Recover section is to enable a process that is restarting after a crash to repair its state and resume from where it was before the crash. It is desirable if a process can do this recovery without being obstructed by other processes, neither by their relative speeds nor by their crashes. Accordingly, we define:

An algorithm satisfies Wait-free Recovery if there is a bound b such that, for all executions E and for all processes p, if p is in the Recover section in a configuration C of E and the subsequent steps of p in E are normal steps, then p moves out of the Recover section in at most b of its own steps.

Wait-free recovery, together with well-formedness, yields two significant benefits:

1. When a process that crashed in the CS restarts, it gets back into the CS in a bounded number of its own steps, regardless of whether other processes are slow, fast, or have crashed.
2. Similarly, when a process that crashed in the Exit restarts, it gets back into the CS or Exit in a bounded number of its own steps, which means that it can complete the protocol without any obstruction from others.

Furthermore, Wait-free Recovery implies Critical Section Reentry, as shown below.

▶ **Lemma 1.** *Wait-free Recovery, together with Well-Formedness and Mutual Exclusion, implies Critical Section Reentry.*

Proof. Suppose that p crashes inside the CS at some time t. When p subsequently restarts, Well-Formedness and Wait-free Recovery ensure that p will reenter the CS at some time t'. Assume, for a contradiction, that Critical Section Reentry is violated because some process q gets into the CS at some time τ such that $t < \tau < t'$. Let C be the configuration at time τ. In C, q is in the CS and p is in the Remainder or Recover sections. If p alone takes steps from C, then Well-Formedness and Wait-free Recovery ensure that p will enter the CS, thereby violating Mutual Exclusion. ◀

2.7 FCFS

Intuitively, FCFS (First Come First Served) requires that processes enter the CS in the order in which they request the CS [8]. For the traditional model where processes are assumed not to crash, Lamport [8] formalized this intuition by (i) stipulating that the Try section be structured as a bounded section of code, called the *doorway*, followed by the rest of the Try section code that he called the *waiting room*, and (ii) requiring that if a process p is in the waiting room in an attempt A before a process p' initiates an attempt A', then p' does not enter the CS in attempt A' before p enters the CS in attempt A. For the traditional model where processes are assumed not to crash, an equivalent formulation that avoids this syntactic separation of Try into doorway and waiting room would be as follows:

> An algorithm satisfies *FCFS* if there is a bound b such that, for all executions E and for all attempts A and A' in E, if A is an attempt by p, A' is an attempt by p', and p executes b steps in attempt A before p' initiates attempt A', then p' does not enter the CS in attempt A' before p enters the CS in attempt A.

In the current setting, in order to get to the CS, a process needs to execute not only the Try section, but also the Recover section where the attempt is initiated. Therefore, formulating FCFS via doorway and waiting room is cumbersome. Consequently, we have chosen to adapt the above stated alternative specification to the current setting. Two issues arise in this exercise, as we discuss below.

- A process that initiates an attempt might crash before completing the b steps required to establish its "priority". Therefore, to prohibit p' from entering the CS before p, the specification should require that p completes b *consecutive normal steps* before p' initiates its attempt.
- A process might enter and leave the CS multiple times within the same attempt if it crashes repeatedly. In particular, if a process leaves the CS and crashes in the Exit section, subsequent execution of the Recover section can put the process back in the CS. Therefore, the final phrase in the FCFS specification that states "p' does not enter the CS in attempt A' before p enters the CS in attempt A" is revised to "p' does not enter the CS in attempt A' before p *first* enters the CS in attempt A".

Putting these elements together, the final specification of FCFS is as follows. Below, when we say "p performs b consecutive normal steps in attempt A before p' initiates attempt A'" we mean that the sequence of steps that p performs in attempt A (before p' initiates A') includes b consecutive steps all of which are normal.

An algorithm satisfies FCFS if there is a bound b such that, for all executions E and for all attempts A and A' in E, if A is an attempt by p, A' is an attempt by p', and p performs b consecutive normal steps in attempt A before p' initiates attempt A', then p' does not enter the CS in attempt A' before p first enters the CS in attempt A.

2.8 RMR complexity

An operation by a process p on a shared variable X is considered a Remote Memory Reference (RMR) if it involves traversing the processor-memory interconnect. On a Cache-Coherent (CC) machine, a read of X by p counts as an RMR if X was not in p's cache (in which case the read brings X into p's cache), and a non-read operation on X by p always counts as an RMR and removes X from all caches. On a Distributed Shared Memory (DSM) machine, shared memory is partitioned and each process hosts a partition. An operation by p on X, whether a read or a non-read, is counted as remote if and only if X is not in p's partition.

For mutual exclusion algorithms that are designed to run on a CC or a DSM machine, the standard performance metric is the *RMR complexity*, which is the worst case number of RMRs that a process performs in a single attempt. Unlike in a non-recoverable algorithm where a process executes the Try and Exit sections exactly once in an attempt, a process in the current setting can execute the Recover, Try, and Exit sections an unbounded number of times within the same attempt because of its repeated crashes during the attempt. Therefore, we express the RMR complexity of an attempt in terms of n and f, where n is the number of processes for which the algorithm is designed and f is the number of times the process crashes during the attempt.

2.9 Specification of Recoverable Mutual Exclusion

The *recoverable mutual exclusion problem* is to design an algorithm that satisfies Mutual Exclusion, Well-Formedness, Starvation Freedom, Bounded Exit, and Critical Section Reentry. We solve this problem with an algorithm that additionally satisfies FCFS and Wait-free Recovery.

3 Important differences with prior works

Golab and Ramaraju's work [4] is the first to explore the use of non-volatile memory to help processes recover from crash when solving mutual exclusion. The subsequent work by Golab and Hendler [3] uses the same model. Our model and specification differs from these in two significant ways.

- In Golab and Ramaraju's model, the Recover section always puts a process p in Try, thereby causing p to execute Try, CS, and Exit sections in that order each time it restarts after a crash. Consequently, even if the crash occurs after p has completed the CS, when p restarts, it is forced to compete once more for the CS (in the Try section). In contrast, in our model the Recover section can put p in any of the sections. In particular, if p crashes while in the Exit section, when p later restarts, the Recover section will put p directly in CS, Exit, or Remainder, bypassing the needless repetition of Try.
- The properties of wait-free recovery and well-formedness that we have defined differentiate our two works, and have significant implication to how quickly a process can recover after a crash. For instance, suppose that a process p crashes while in the CS. When p subsequently restarts, our well-formedness and wait-free recovery properties together ensure that p will get back into CS in a bounded number of its own steps, i.e., without any waiting whatsoever regardless of however many other processes crashed or are slow. In contrast, Golab and Ramaraju's specification does not have such a guarantee.

We note that, given an algorithm A designed for our model, it is straightforward to transform it into an algorithm A' that conforms with Golab and Ramaraju's model: if the Recover section in A puts a process in Exit section, then the Recover section in A' will make p complete the Exit code (within the Recover section itself) and then send it to Try section.

4 An auxiliary min-array object

Our recoverable FCFS mutual exclusion algorithm, presented in the next section, relies on a special object O that we call a *min-array*. A min-array O has n locations, one per process, and supports two types of operations:

(i) *write*(v), which when executed by process p sets $O[p]$ to v, and
(ii) *findmin*$()$, which returns the minimum value in the array.

No hardware directly supports a min-array, so we implement it using read, write, and CAS operations. Such an implementation of a min-array O specifies code for two procedures, namely, $O.write(p, v)$ that p can execute to set $O[p]$ to v and $O.findmin()$ that a process can execute to get the minimum element in O. For the implementation to be useful in the design of our algorithm, we require it to satisfy wait-freedom and idempotence, which are explained below.

- As always, wait-freedom means that if p invokes $O.write(p, v)$ or $O.findmin()$ and executes the steps of that procedure without crashing, then p will complete and return from that procedure in a bounded number of its own steps [5].
- To explain what we mean by idempotence, let us begin with the notion of a partial execution by p of a procedure Π, where Π is either $O.write(p, v)$ (for a fixed v) or $O.findmin()$. An execution of Π is partial if p invokes Π and performs an arbitrary number of steps of Π, but does not run Π to completion. With most implementations, if p executes Π partially and then reexecutes Π from the start, the implementation can go completely haywire and return arbitrary responses. We however require the implementation to behave gracefully in such a scenario because, if p crashes in the middle of Π, by our model it goes to the Remainder section and subsequently when it starts running again, it might reexecute Π. Motivated by this requirement, we define an *epoch of* Π *by* p as consisting of one or more partial/complete executions of Π. We say the epoch has length f if it consists of f partial/complete executions of Π. By *idempotence* we mean that (i) if the epoch contains at least one complete execution of Π, then the epoch linearizes to some point within the epoch; (ii) if the epoch does not contain a complete execution of Π, then either the epoch never takes effect or it linearizes to some point after the start of the epoch; and (iii) if the same process executes an epoch e of Π and later an epoch e' of a different procedure Π', e' will not linearize before e. (Reader unfamiliar with linearizability is referred to [6].)

Jayanti designed a wait-free, linearizable implementation of a min-array [7] which, although not claimed in his paper, is idempotent. That implementation uses LL/SC operations and has an RMR complexity of $O(\lg n)$ for $O.write(p, v)$ and $O(1)$ for $O.findmin()$. We make a simple modification so that the implementation uses CAS instead of LL/SC and has an RMR complexity of $O(f + \lg n)$ for an epoch of $O.write(p, v)$ of length f. We briefly describe this implementation in the appendix and summarize the result below.

▶ **Lemma 2.** *The algorithm in Algorithm 2 and 3 in the appendix presents a wait-free, idempotent implementation of a min-array O using read, write, and CAS primitives. The RMR complexity of an epoch of O.*write(p, v) *of length f is $O(f + \lg n)$ and the RMR complexity of an epoch of O.*findmin$()$ *of length f is $O(f)$.*

5 The Algorithm

Our Recoverable Mutual Exclusion algorithm is presented in Algorithm 1. We assume that all the shared variables are stored in the Non-volatile Main Memory, and the local (or private) variables are stored in the respective processor registers.

5.1 Shared variables and their purpose

We describe below the role played by each shared variable used in the algorithm.

- Go$[p]$: This is a boolean flag that process p busywaits on, before entering the CS. This variable is set to *true* by p at the start of its Try Section. When a process q makes p the

Algorithm 1 Recoverable Mutual Exclusion Algorithm with FCFS and Wait-free Recovery. Algorithm for process p.

Constants
$\mathcal{P} = \{1, 2, \ldots, n\}$ // Set of process names.
Shared variables (stored in NVMM)
REGISTRY : A min-array, initially empty.
CSOWNER $\in (\mathcal{P} \times \mathcal{P}) \cup (\perp \times \mathcal{P})$, initially $(\perp, 1)$; supports *read*, *write*, and *CAS* operations.
$\forall p \in \mathcal{P}$, GO$[p]$ is a boolean initialized to *true*; supports *read* and *write* operations.
$\forall p \in \mathcal{P}$, REM$[p]$ is a boolean initialized to *true*; supports *read* and *write* operations.
$\forall p \in \mathcal{P}$, MYTOKEN$[p]$ is an integer initialized to ∞; supports *read* and *write* operations.
TOKEN is an integer initialized to 0; supports *read* and *CAS* operations.
Private variables
$\forall p, g_p$ is a boolean arbitrarily initialized, local to p.
$\forall p, t_p$ is an integer arbitrarily initialized, local to p.
$\forall p, i_p$, j_p, s_p, α_p contain a process name from $\mathcal{P}$, all initialized arbitrarily, and local to p.

```
    Recover Section
1.  if REM[p] == true goto Line 7
2.  g_p ← GO[p]
3.  if ((s_p, −) ← CSOWNER) ∧ (s_p ≠⊥)
4.      if ¬GO[s_p]
5.          if (s_p, p) == CSOWNER
                if s_p == p goto Line 16
                else α_p ← s_p; goto Line 23
6.  t_p ← MYTOKEN[p]
    if ¬g_p
        if t_p == ∞ goto Line 9
        else goto Line 12
    else if s_p == p
        if t_p == ∞ goto Line 19
        else goto CRITICAL SECTION
    else goto Line 21

    Try Section
7.  GO[p] ← false
8.  REM[p] ← false
9.  t_p ← TOKEN
10. CAS(TOKEN, t_p, t_p + 1)
11. MYTOKEN[p] ← t_p
12. REGISTRY.write(p, (t_p, p))
13. if ((i_p, j_p) ← CSOWNER) ∧ (i_p ==⊥)
14.     if REGISTRY.findmin() == (t_p, p)
15.         if CAS(CSOWNER, (⊥, j_p), (p, p))
16.             GO[p] ← true
17. wait till GO[p] == true

    Exit Section
18. MYTOKEN[p] ← ∞
19. REGISTRY.write(p, (∞, ∞))
20. CSOWNER ← (⊥, p)
21. if ((−, α_p) ← REGISTRY.findmin()) ∧ (α_p ≠ ∞)
22.     if CAS(CSOWNER, (⊥, p), (α_p, p))
23.         GO[α_p] ← true
24. REM[p] ← true
```

owner of the CS, q releases p from its busywait loop by assigning *false* to GO$[p]$. The operations supported by GO$[p]$ are read and write. To achieve the local-spin property, GO$[p]$ is allocated to p's memory module.

- REM$[p]$: This is a boolean flag that process p uses during recovery to distinguish whether p is active in an attempt or if p entered the Recover section from the Remainder section to initiate a new attempt. This variable is set to *false* by p during its Try Section indicating that it has initiated an attempt. p completes its attempt by setting REM$[p]$ to *true* in the Exit Section. The operations supported by REM$[p]$ are read and write.
- TOKEN is an integer variable supporting read and CAS operations. TOKEN is used to implement a counter so that its values can be used to assign token numbers to processes requesting the CS: in the Try section, a process reads TOKEN to get its token number and then increments TOKEN. As a result, if p executes up to Line-10 and obtains a token during an attempt A before q begins its attempt A', p will get a smaller token number than q (this fact helps achieve the FCFS and starvation-freedom properties).
- REGISTRY is a min-array that supports `write` and `findmin` operations. We require that the two operations satisfy wait freedom and idempotence as mentioned in Section 4. In our algorithm, REGISTRY acts like a queue by holding the names of processes waiting to enter the CS, and orders them according to their token numbers. Specifically, in the Try section, a process p inserts in REGISTRY an element (t, p) (Line 12), where t is p's token

number. When exiting, p deletes this element (Line 19). The elements in REGISTRY are ordered according to their token numbers: $(t, p) < (t', q)$ if $t < t'$ or $t = t' \wedge p < q$. Thus, the `findmin` operation returns (t, p), where p is the process in REGISTRY with the smallest token. If REGISTRY is empty, `findmin` returns the special value (∞, ∞).

- MYTOKEN$[p]$ is an integer variable supporting read and write operations. It is used by processes to remember the token numbers they use while registering their attempt. A process may acquire a token, attempt to insert its entry into the REGISTRY, and crash while completing the insertion. So that it does not acquire a different token and attempt to insert the new token into REGISTRY after recovery from that crash, the process remembers the token that it acquired by recording the token into MYTOKEN$[p]$.
- CSOWNER: This variable holds a tuple both components of which hold process names. The first component holds the name of the process that currently owns the CS. If no process currently owns the CS, the first component holds the value $\perp$. The second component holds the name of the process that gave the ownership of the CS to a certain process. A process may crash just after granting the ownership of CS to some process but right before letting that process into CS, hence not setting the GO flag of that process to *true*. We use the second component for recovery purposes in such a case so that the process can easily remember that it needs to let the waiting process into the CS which is made owner of the CS but not yet let into the CS. The operations supported by CSOWNER are read, write, and CAS.

5.2 Informal description

We now describe informally the algorithm in Algorithm 1. When a process p wants to enter the CS from the Remainder section, it executes an attempt initiation step. In this step it executes the first instruction of Recover and finds that REM$[p]$ is *true*. Therefore, the process proceeds to Line 7 in the Try section. It sets its GO flag (Line 7), and then signals that it is now active in an attempt by setting REM$[p]$ to *false* (Line 8). It then proceeds to obtain a token for itself and increments the global counter (Line 9-10). It also saves the token it obtained for itself (Line 11), so that in the event of a crash it does not obtain a different token and try registering its attempt with the new token. Then, it inserts its name, tagged with its token, into the REGISTRY (Line 12). If a process p executes normal steps upto Line 12 in attempt A before q initiates an attempt A', then q does not enter the CS in A' before p first enters the CS in A. After executing the Line 12, p tries to capture the CS on its own by first confirming that the CS is free to be occupied (Line 13), the process with the smallest token in the REGISTRY is p itself (Line 14), and then attempting to capture the CS for itself (Line 15). If p succeeds in capturing the CS, it sets its own GO flag to *true* (Line 16), informing itself that it no longer needs to wait. Following this, p busy-waits until it is informed that it no longer needs to wait (Line 17). It then enters the CS. When p leaves the CS, it first wipes out the token that it obtained for the attempt (Line 18). It then removes its own name from the REGISTRY (Line 19), and marks the CS as available (Line 20). Following this, p tries to capture the CS for a waiting process (Line 21-22) and informs the waiting process that it no longer needs to wait (Line 23), if successful in capturing the CS. Whether p lets another process into the CS or not, it completes the attempt by setting REM$[p]$ to *true* (Line 24) to indicate that it is no longer active in an attempt.

Next we describe the Recover section. A process p may go to the Recover section due to one of two reasons:

(i) p may crash,

(ii) p may initiate a new attempt due to an attempt initiation step.

Therefore, p first reads Rem[p] to determine if it wants to initiate an attempt (Line 1). If so, p proceeds to the Try Section. Otherwise, p reads its own Go flag so that it can decide later where it crashed during the attempt (Line 2). p then checks if any process is currently the owner of CS. The section of code on Lines 4-5 is executed when p learns that some process owns the CS. Suppose p learns that some process owns the CS. Then p checks the Go flag of that process to determine if that process was informed to no longer wait. If that process is still waiting without any information, p checks if the CS is still owned by the same process, and whether p installed that process as the owner. If that is so, p was either installing itself as the owner, in which case it has to go back to the Try section (i.e. Line 16), or while exiting p was installing some other process as the owner of the CS, in which case it has to go to the Exit section again. If any of the checks at Lines 4-5 fail, p is not responsible in informing any waiting process. If no process is the owner, then p definitely did not crash while letting some process into the CS. Therefore, p proceeds to read its own token (Line 6). Depending on p's token value, Go flag, and the value of s_p (used only to identify if p itself is the owner of CS), p decides whether to go back to Try, CS, or Exit section.

5.3 The subtle features of the algorithm

From the above description it might not be clear why we perform certain operations in the algorithm, namely,

(A) the need to set Rem[p] after setting Go[p] (Line 7-8),

(B) the need to read CSOwner twice in the Recover section (Lines 3, 5).

We demonstrate below that the first feature is necessary to preserve Well-formedness. The second feature is necessary to preserve Mutual Exclusion.

The need for feature A: In order to distinguish between a fresh attempt and a recovery to an active attempt, we have to maintain the boolean variable Rem[p]. The placement of the lines that modify Rem[p] is also important. Suppose, instead, we set Rem[p] to *false* in the first line of the Try Section. Then it is possible for a process to re-initiate an already completed attempt as we demonstrate in the following scenario. Process p initiates a new attempt, obtains the lock and executes the CS, and continues to the Exit section. p executes all the steps upto Line 24. Right when $PC_p = 24$, p executes a crash step. The next steps that p executes are from the Recover section. p notices that Rem[p] = *true*, MyToken[p] = ∞, and Go[p] = *true*. Since p didn't crash at Line 23 (which it knows after reading CSOwner), it incorrectly concludes that it crashed somewhere in the Try section before obtaining a token for itself. Therefore, p goes back to the Try section instead of the Exit section. This violates Well-formedness. This problem would not arise with p if Rem[p] is set to *false* in Line 8, because by reading Go[p] it can infer that the value of Go[p] is *true* due to its own completed attempt. If p had not initiated an attempt, Go[p] would still be *true*, but Rem[p] would also be *true*. Therefore, p can infer correctly that it needs to go to the Exit instead of Try section.

The need for feature B: Suppose p crashes either at Line 16 or Line 23, i.e., while setting the Go flag of some process. Then, during recovery, in order to conclude that p crashed just before writing to the Go flag, p has to read CSOwner and suppose the second component contains p's name, then it has to read the Go flag of the process whose name appears in the first component of CSOwner. Let s_1 be the step when p reads CSOwner the first time for recovery. Let q be the process whose name p read in the first component of CSOwner

and p read its own name in the second component. Let s_2 be the step when p reads GO$[q]$. Suppose p finds that GO$[q]$ is *false* during s_2. There are two cases applicable here:
(1) During s_1 GO$[q]$ was *true*,
(2) During s_1 GO$[q]$ was *false*.
In the first case, p had already let q into the CS. Between steps s_1 and s_2, q completed that attempt (this would change the value of CSOWNER to something other than (q, p)), and later came back for another attempt setting GO$[q]$ back to *false*. Hence, p found GO$[q]$ as *false* by executing s_2 in the first case. In the second case, q was actually somewhere in the Try during s_1 and s_2 and it continues to be so, therefore, q has to be let into the CS by p. In order to distinguish between the two cases, we read CSOWNER again at Line 5. If p finds that CSOWNER is still (q, p), it concludes that it crashed just before setting GO$[q]$ to *true*. Hence, it goes back to the appropriate line. Otherwise, p hasn't obstructed any process from going into the CS, therefore, it resumes with the rest of the recovery. If p didn't read CSOWNER for the second time in Line 5, p would incorrectly inform q to proceed to CS in the first case discussed above. Hence, q would continue to CS when some other process already occupies it, violating Mutual Exclusion.

5.4 Main theorem

The theorem below presents our main result. We will soon release a Dartmouth College Technical report that includes a detailed proof of correctness based on an invariant satisfied by the algorithm.

▶ **Theorem 3.** *The algorithm in Algorithm 1 solves the recoverable mutual exclusion problem for n processes and additionally satisfies FCFS and Wait-free Recovery. The algorithm uses read, write and CAS operations. On both CC and DSM machines, a process that fails at most f times in an attempt performs at most $O(f + \lg n)$ RMRs.*

Acknowledgment. We are grateful to the anonymous reviewers for their careful and detailed reviews and Wojciech Golab for helpful discussions.

References

1 Hagit Attiya, Danny Hendler, and Philipp Woelfel. Tight RMR Lower Bounds for Mutual Exclusion and Other Problems. In *Proc. of the Fortieth ACM Symposium on Theory of Computing*, STOC'08, pages 217–226, New York, NY, USA, 2008. ACM.
2 E. W. Dijkstra. Solution of a Problem in Concurrent Programming Control. *Commun. ACM*, 8(9):569–, September 1965.
3 Wojciech Golab and Danny Hendler. Recoverable mutual exclusion in sub-logarithmic time. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC'17, pages 211–220, New York, NY, USA, 2017. ACM.
4 Wojciech Golab and Aditya Ramaraju. Recoverable Mutual Exclusion: [Extended Abstract]. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC'16, pages 65–74, New York, NY, USA, 2016. ACM.
5 Maurice Herlihy. Wait-free Synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991.
6 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.
7 Prasad Jayanti. f-arrays: Implementation and Applications. In *Proceedings of the Twenty-first Symposium on Principles of Distributed Computing*, PODC'02, pages 270–279, New York, NY, USA, 2002. ACM.

8 Leslie Lamport. A New Solution of Dijkstra's Concurrent Programming Problem. *Commun. ACM*, 17(8):453–455, August 1974.

9 John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, February 1991.

10 Dushyanth Narayanan and Orion Hodson. Whole-system persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 401–410, New York, NY, USA, 2012. ACM.

11 Aditya Ramaraju. RGLock: Recoverable mutual exclusion for non-volatile main memory systems. Master's thesis, University of Waterloo, 2015. URL: `https://uwspace.uwaterloo.ca/handle/10012/9473`.

12 Jae-Heon Yang and Jams H. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9(1):51–60, 1995.

A Min-Array implementation with f-arrays

In this section we describe Jayanti's f-arrays implementation [7]. Detailed explanation and a proof of correctness are presented in Jayanti's paper. Here we present a modified version of the implementation so that it incurs only a constant extra RMR in the event of a failure during any procedure call. Since the original implementation uses LL/SC, we give an equivalent implementation that uses CAS and unbounded counters.

Algorithm 2 gives the tree-based implementation of the f-arrays using CAS. In the implementation there is one node for every process, for a total of n nodes, representing the array cells. These n nodes are the leaves of a binary tree such that the height of the tree is $\lg n$.

In computing the min function, the idea is that a process enters an element in its own cell first. It then traces a leaf to root path so that at each node in the path it takes the values of that node's children and writes the minimum among these values into the node. This continues until the process reaches the root, where it stores the minimum value it encountered in the path.

The shared variable SavedState is an additional component that stores the address of a node in the tree. During the execution of the `write` operation a process may abort (in our case crash) to resume the execution at a later point of time. When a process invokes a procedure call to `write`, we first check if the call was to write a different value from the previous call (Line 26). If so, then start percolating the value into the f-array from the leaf nodes of the tree as explained above. Otherwise, the current call is an invocation to write the same value as before, hence, to maintain idempotence, start percolating the value up the tree from the point where the process stopped in the last invocation before the abort.

As described in [7], after the second call to `refresh` we are sure that the value in *val* is percolated upto the node in the tree pointed to by *currNode*. Therefore, we save this progress as a checkpoint along with the value in the shared variable SavedState, and continue percolating the value up the tree.

In each call the function min() (or $f()$ in case of an f-array) in Line 42 takes the minimum value stored in the child nodes of the node pointed by *currNode* and writes the minimum value into *currNode*. The call to refresh happens repeatedly so that a value that is first written to a leaf node in the tree (Line 25) gets percolated up and ultimately once the process reaches the root of the tree, it writes in the root the minimum value in the array. To read the minimum value, the process simply reads the root node and returns the value (Line 37, 38).

Algorithm 2 Tree algorithm to implement f-arrays object with CAS. Algorithm for process p. Adaptation of algorithm is from Figure 4 in [7]. Note, here we explicitly substitute the min() function in Line 42 instead of the function $f()$ since we know we will use this as a min-array. Algorithm 1 does not constrain the structure of the object tree τ'. Therefore, we assume that τ' is a binary tree of height $O(\lg n)$.

Shared objects (stored in NVMM)
τ': Object tree corresponding to a type tree τ as described in Section 4.2 of [7].
$\forall p, L_p$ is the pth leaf of τ'.
Shared variables (stored in NVMM)
$\forall p$, SavedState$[p]$: An array storing the node in the tree to start update from, and the value on which $\texttt{write}_p$ was called. Initialized to $(\&L_p, \infty)$.
Private variables
val, val' are integers, initialized arbitrarily.
$currNode$ is a reference to a node in τ', initialized arbitrarily.
$res, seq, v, v_1, v_2, \ldots, v_k$ are integers, initialized arbitrarily.

procedure $\texttt{write}_p(val)$
25. $L_p \leftarrow val$; $res \leftarrow val$
26. **if** $(((currNode, val') \leftarrow$ SavedState$[p]) \wedge$ $val \neq val')$
27. $currNode \leftarrow L_p$
28. **if** $currNode == root(\tau')$
29. **return** res
30. **repeat**
31. $currNode \leftarrow parent(currNode)$
32. **if** $\neg\texttt{refresh}(currNode)$
33. $\texttt{refresh}(currNode)$
34. SavedState$[p] \leftarrow (currNode, val)$
35. **until** $currNode == root(\tau')$
36. **return** res

procedure $\texttt{read}()$
37. $(v, -) \leftarrow \text{read}(root(\tau'))$
38. **return** v

procedure $\texttt{refresh}(currNode)$
Let $C_1, \ldots, C_k$ be $currNode$'s children
39. $(val, seq) \leftarrow \text{read}(currNode)$
40. **for** $i \leftarrow 1$ **to** k
41. $v_i \leftarrow \text{read}(C_i)$
42. **return** CAS$(currNode, (val, seq),$ $(f(v_1, v_2, \ldots, v_k), seq + 1))$

Algorithm 3 Implementation of min-array using f-arrays (see Algorithm 2; substitute min() function for the function $f()$ in Line 42) adapted from algorithm of Figure 8 in [7]. Algorithm for process p.

Shared variables and objects (stored in NVMM)
$\mathcal{A}$: An f-array of n components as described in Algorithm 2.
Private variables
val, v are integers, initialized arbitrarily.

procedure $O.\texttt{write}(p, val)$
43. $\texttt{write}_p(\mathcal{A}, val)$

procedure $O.\texttt{findmin}()$
44. $v \leftarrow \texttt{read}(\mathcal{A})$
45. **return** v

In Algorithm 3 we give an f-array implementation of Registry used in our algorithm. We adapt this from Jayanti's f-arrays paper [7] except that we do not utilize adaptivity as done in the paper.

Interactive Compression for Multi-Party Protocols

Gillat Kol[1], Rotem Oshman[*2], and Dafna Sadeh[†3]

1 **Princeton University, USA**
gillat.kol@gmail.com
2 **Tel Aviv University, Israel**
rotem.oshman@gmail.com
3 **Tel Aviv University, Israel**
dafnasadeh@mail.tau.ac.il

Abstract

The field of compression studies the question of how many bits of communication are necessary to convey a given piece of data. For one-way communication between a sender and a receiver, the seminal work of Shannon and Huffman showed that the communication required is characterized by the *entropy* of the data; in recent years, there has been a great amount of interest in extending this line of research to *interactive communication*, where instead of a sender and a receiver we have two parties communication back-and-forth. In this paper we initiate the study of interactive compression for distributed multi-player protocols. We consider the classical *shared blackboard model*, where players take turns speaking, and each player's message is immediately seen by all the other players. We show that in the shared blackboard model with k players, one can compress protocols down to $\tilde{O}(I \cdot k)$, where I is the information content of the protocol and k is the number of players. We complement this result with an almost matching lower bound of $\tilde{\Omega}(I \cdot k)$, which shows that a nearly-linear dependence on the number of players cannot be avoided.

1998 ACM Subject Classification E.4 Coding and Information Theory

Keywords and phrases interactive compression, multi-party communication

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.31

1 Introduction

In their seminal work, Shannon, Fano and Huffman considered the *data compression problem*: a sender wants to send a message x to a receiver. We think of x as a random variable generated from some distribution μ. How many bits does the sender need to send, so that the receiver will be able to recover x with high probability? The answer given in [15, 9, 11] is that he needs to send only $\lceil \mathrm{H}(x) \rceil$ bits, in expectation, where H denotes Shannon's entropy function. Roughly speaking, this means that every message can be compressed to its information content.

While classical information theory studied the case of one-way transmission, over the last decades, *interactive* communication protocols were also studied extensively. The *interactive compression problem* [2] is the analog of the data compression problem in the interactive setting: it asks whether the transcript of any interactive protocol can be *compressed* to its information content. Roughly speaking, compressing a protocol Π means constructing

* Rotem Oshman is supported by the Israeli Centers of Research Excellence (I-CORE) program (Center No.4/11) and by BSF Grant No. 2014256.
† Dafna Sadeh is supported by the Israeli Centers of Research Excellence (I-CORE) program (Center No.4/11) and by BSF Grant No. 2014256.

© Gillat Kol, Rotem Oshman, and Dafna Sadeh;
licensed under Creative Commons License CC-BY

a different protocol Π', hopefully with smaller communication, which computes the same function. The interactive compression problem for the two players setting attracted a lot of attention in recent years, resulting in several compression protocols for different settings [10, 2, 6, 4, 7, 3, 14, 12, 16].

Interactive compression can be viewed as a tool for protocol design: one first designs a communication-inefficient protocol, making sure only that it does not reveal a lot of information about the inputs; interactive compression then allows us to convert the protocol into a communication-efficient one.

In this work we initiate the study of *distributed interactive compression*. We show how to *compress* a given k-party communication protocol, in order to reduce its communication, and we also show the limitation of such compression schemes.

We study the *shared blackboard* model of multi-party communication. In this classical model, the players communicate over a "shared blackboard", taking turns to write messages on the board. All players can see the contents of the board, and the player whose turn it is to write next is determined by what was written so far. The model can be viewed as a single-hop radio network: when a player sends a message, the message is immediately received by all the other players.

Measuring information content. As discussed above, in the case of one-way communication, the information content of the data $\boldsymbol{D}$ we wish to compress is measured by its Shannon entropy, $\mathrm{H}(\boldsymbol{D})$. The analog of entropy for interactive communication is *information cost* [8, 1, 2], which measures how much information the players reveal about their inputs.

The precise notion we work with here is called *external information cost*: it measures, in mutual information, the amount of information an external observer learns about the players' inputs from observing the transcript of the protocol. Formally, if $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_k$ are random variables denoting the inputs to the k players, sampled according to the joint distribution μ, and $\boldsymbol{\Pi}(\boldsymbol{X}_1, \ldots, \boldsymbol{X}_k)$ is a random variable denoting the transcript of the protocol Π when it is run with the inputs $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_k$, then the external information cost of the protocol Π with respect to the distribution μ is defined as

$$\underset{\mu}{\mathrm{IC}}(\Pi) = \mathrm{I}(\boldsymbol{\Pi}(\boldsymbol{X}_1, \ldots, \boldsymbol{X}_k); \boldsymbol{X}_1, \ldots, \boldsymbol{X}_k),$$

where I denotes the mutual information, $\mathrm{I}(\boldsymbol{A}; \boldsymbol{B}) = \mathrm{H}(\boldsymbol{A}) - \mathrm{H}(\boldsymbol{A}|\boldsymbol{B}) = \mathrm{H}(\boldsymbol{B}) - \mathrm{H}(\boldsymbol{B}|\boldsymbol{A})$.

We mention that in the two-player case, another notion of information cost, called *internal* information cost, was studies extensively. This notions measures the amount of information that the players learn about *each other's* inputs from their interaction. Internal information cost differs from external information cost when the players' inputs are not independent, because in this case the players potentially know something about each other's inputs just by looking at their own inputs, while an external observer has no such prior information. However, it is unclear how to adapt the definition of internal information to the multi-player setting in a meaningful way (and indeed, this is an interesting open problem). In the sequel, when we say "information cost", we mean external information cost.

1.1 Our Results

1.1.1 A compression protocol in the shared blackboard model

We show that a protocol with information cost I can be compressed down to $\tilde{O}(I \cdot k)$ bits of communication, where the $\tilde{O}$-notation hides polylogarithmic factors in I, k, and the original communication cost of the protocol.

What does it mean to "compress" a protocol? As in the case of non-interactive data compression, we give a *compression scheme* and a *decoding function* dec. Given any protocol Π, we can apply the compression scheme to obtain a compressed protocol Π' (hopefully with less communication), and we can apply the decoder dec to Π''s transcripts (and its public randomness) to extract from them transcripts of Π. Our compression has some error: for any input X, the transcript we extract from Π' on X is *close in distribution* to the transcript of Π on X.

More formally, let π_X be the distribution of the transcript of Π on a specific input X, and let $\mathsf{dec}(\pi'_X)$ be the distribution of the extracted transcript obtained by running Π' on X and then applying the decoding function dec. Then our compression result can be stated (slightly informally) as follows:

▶ **Theorem 1** (Compression in the Shared Blackboard Model, Informal)**.** *Let $\rho > 0$ and $k \in \mathbb{N}$. Let Π be a randomized shared blackboard protocol between k players, and let μ be a joint distribution over the inputs for the players in Π. Then there exists a randomized protocol Π' in the* shared blackboard model *satisfying the following properties:*

1. *The worst case communication complexity of Π' is $\tilde{O}(k \cdot \mathrm{IC}_\mu(\Pi)/\operatorname{poly}(\rho))$.*
2. *There exists a deterministic function* dec *that given a transcript of Π', outputs a corresponding transcript of Π, such that for any global input X we have $\mathrm{SD}(\pi_X, \mathsf{dec}(\pi'_X)) \le \rho$.*

Here, $\mathrm{SD}(\mu, \eta) = \sup_{A \subseteq \Omega} |\mu(A) - \eta(A)|$ denotes the statistical distance between the distributions μ, η over the universe Ω, and $\mu(A), \eta(A)$ denote the probability of event A under μ and η respectively.

Our compression scheme is based on the beautiful two-player compression scheme of [2] for external information, but we face several non-trivial challenges in adapting the scheme to work with multi-player protocols.

1.1.2 Compression lower bound

Our compression scheme achieves communication $\tilde{O}(I \cdot k)$. For any compression scheme, the information cost $\mathrm{IC}(\Pi)$ is a lower bound on the communication of the compressed protocol, as any bit communicated by the protocol can give at most one bit of information about the inputs. However, it is natural to ask whether the blowup by a factor of k in the communication complexity of our above compression result is necessary, and we show that indeed it is: there is a communication protocol with information cost I, which cannot be compressed to a protocol that uses less than $\tilde{\Omega}(k \cdot I)$ bits of communication. This rules out the existence of a better compression scheme than the one suggested by Theorem 1, up to logarithmic factors.

To show this lower bound, we construct a function f on k inputs, and show that f can be computed by a protocol Π with information cost I. In contrast, we prove an $\tilde{\Omega}(k \cdot I)$ lower bound on the distributional communication complexity of f, that is, we show that no protocol with communication cost $\tilde{O}(k \cdot I)$ can compute f. This gives a *separation* between information and communication in the distributed multi-player setting. Observe that this also means that the protocol Π, which has information cost I, cannot be simulated by a protocol with communication complexity less than $\tilde{\Omega}(k \cdot I)$, because this would give us a low-communication protocol for solving f.

▶ **Theorem 2.** *Let $I, k \in \mathbb{N}$. There exists a function $f(X_1, \ldots, X_k)$ on k inputs, and a joint distribution μ over the inputs $X = (X_1, \ldots, X_k)$ for f, such that the following hold:*

1. *There exists a deterministic communication protocol Π with $\mathrm{IC}_\mu(\Pi) = I$ for which the output of $\Pi(X)$ is $f(X)$ for every $X \in \operatorname{supp}(\mu)$.*

2. *Any randomized (public coin) communication protocol* Π' *satisfying*

$$\Pr[\Pi'(X) \text{ outputs } f(X)] \geq 0.99,$$

must communicate $\tilde{\Omega}(k \cdot I)$ *bits on average. Here the error probability and the average communication are over inputs* X *drawn from* μ *and the randomness used by* Π'*,*

We sketch the proof in Section 4.

A gap of $\tilde{\Omega}(k)$ between information and communication in the shared blackboard model was first shown in [5], where it is shown that the AND function on k input bits has a protocol with information cost $O(\log k)$, but any protocol that computes AND with high probability communicates at least $\Omega(k)$ bits. However, this leaves open the possibility that the difference is *additive* in k: that is, it could conceivably be that every protocol Π can be compressed to a protocol with communication complexity $\tilde{O}(I + k)$. Indeed, consider the Disjointness problem, $\text{DISJ}_{n,k}$, where each player i gets a set $X_i \subseteq [n]$, and we want to determine if $\bigcap_i X_i = \emptyset$. We can view $\text{DISJ}_{m,k}$ as the OR of n instances of AND: $\text{DISJ}_{n,k}(X_1, \ldots, X_k) = \bigwedge_{j=0}^{n-1} \bigvee_{i=1}^{k} \neg X_j^i$. It is shown in [5] that $\text{DISJ}_{n,k}$ has information $\Theta(n \log k)$ and communication cost $\tilde{\Theta}(n \log k + k)$ in the shared blackboard model, so even though AND exhibits a gap of $k/\log k$ between communication and information, somehow "many instances of AND" no longer exhibit the same gap. Nevertheless, Theorem 2 above shows that compression to $\tilde{O}(I + k)$ is impossible in general, so $\text{DISJ}_{n,k}$ is the exception and not the rule.

1.2 Organization of the Paper

The remainder of the paper is organized as follows. In Section 2, we review the basic notions of communication complexity and information theory required to state and prove our compression results. In Section 3 we sketch the compression scheme; for lack of space, some technical details are omitted here, and will appear in the full version of the paper. Finally, in Section 4 we describe our compression lower bound.

2 Preliminaries

Notation. We use bold-face letters to denote random variables. For variables $\boldsymbol{A}_1, \ldots, \boldsymbol{A}_\ell$ with joint distribution μ, we let $\mu(\boldsymbol{A}_i)$ denote the marginal distribution of $\boldsymbol{A}_i$, and $\mu(\boldsymbol{A}_i \mid \boldsymbol{A}_j = a_j)$ denote the distribution of $\boldsymbol{A}_i$ conditioned on $\boldsymbol{A}_j = a_j$ (and similarly for more variables). For a string S, we let $|S|$ denote the length of S.

Communication complexity. For a protocol Π in the shared blackboard model, we define the *communication complexity of* Π, denoted $\text{CC}(\Pi)$, as the worst-case number of bits that are written on the board in any execution of Π. We say that Π *solves* a problem $P : \mathcal{X}^k \to \mathcal{Y}$ if for any input $X = (X_1, \ldots, X_k) \in \mathcal{X}^k$, the probability that Π's output on X is $P(X)$ is at least $2/3$. The *communication complexity* of a problem P, denoted $\text{CC}(P)$, is the minimum communication complexity of a protocol that solves P. We also study the *distributional communication complexity* of P, denoted $\text{CC}_\mu(P)$, where now the minimum is taken over protocols that only need to succeed with high probability over inputs drawn from the distribution μ.

Information theory. We require the following notions.

For a pair of random variables $\boldsymbol{X}, \boldsymbol{Y}$ with joint distribution μ, we denote by $\mathrm{I}_\mu(\boldsymbol{X};\boldsymbol{Y}) = \mathrm{H}(\boldsymbol{X}) - \mathrm{H}(\boldsymbol{X}|\boldsymbol{Y})$ the *mutual information* between $\boldsymbol{X}$ and $\boldsymbol{Y}$ (here H is the Shannon entropy). We omit the distribution μ when clear from the context.

For a pair of distributions p, q over the same domain, we denote by $\mathsf{D}(p \parallel q)$ the *KL divergence* between p and q, given by $\mathsf{D}(p \parallel q) = \mathrm{E}_{x\sim p}\left[\log\left(\frac{p(x)}{q(x)}\right)\right]$. Mutual information and KL divergence satisfy: $\mathrm{I}(\boldsymbol{X};\boldsymbol{Y}) = \mathrm{E}_{x\sim\mu(\boldsymbol{X})}\left[\mathsf{D}\left(\mu(\boldsymbol{Y}|\boldsymbol{X}=x) \parallel \mu(\boldsymbol{Y})\right)\right]$, where μ is the joint distribution of $\boldsymbol{X}, \boldsymbol{Y}$, $\mu(\boldsymbol{X})$ (resp. $\mu(\boldsymbol{Y})$) is the marginal distribution of $\boldsymbol{X}$ (resp. $\boldsymbol{Y}$), and $\mu(\boldsymbol{Y}|\boldsymbol{X}=x)$ is the distribution of $\boldsymbol{Y}$ conditioned on $\boldsymbol{X}=x$. Intuitively, the mutual information measures the differences between the distribution of $\boldsymbol{Y}$ when we know $\boldsymbol{X}$, and the prior distribution of $\boldsymbol{Y}$.

Information cost. We will use the following measure of the information revealed by an interactive protocol.

▶ **Definition 3.** The *information cost* of a (private coin) protocol Π over random inputs $\boldsymbol{X} = \boldsymbol{X}_1, \ldots, \boldsymbol{X}_k$ drawn according to a joint distribution μ, is defined as

$$\underset{\mu}{\mathrm{IC}}(\Pi) = \underset{\mu}{\mathrm{I}}(\boldsymbol{\Pi}; \boldsymbol{X}_1, \ldots, \boldsymbol{X}_k),$$

where $\boldsymbol{\Pi}$ is a random variable indicating the transcript of Π on inputs $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_k$.

3 Compression for Multi-Party Protocols: a Proof Sketch

Suppose we are given a protocol Π, with communication $\mathrm{CC} = \mathrm{CC}(\Pi)$ and information cost $I = \mathrm{IC}_\mu(\Pi)$ on some input distribution μ. We want to construct another protocol, Π', which on a given input *generates a transcript of* Π, but with communication cost that depends only polylogarithmically on $\mathrm{CC}(\Pi)$, and mainly depends on the information cost I of Π and on the number of players, k. We follow the framework introduced in [2] for two-party protocols.

All compression schemes rely on the intuition that if $\mathrm{IC}_\mu(\Pi) = \mathrm{I}_\mu(\boldsymbol{\Pi}; \boldsymbol{X})$ is small, then someone who does not know the input $\boldsymbol{X}$ can sample "close to" the correct distribution $\boldsymbol{\Pi}$ even without knowing the input $\boldsymbol{X}$. For example, in the extreme case where the information cost is 0, transcript is *independent* of the inputs and we can sample it from its correct distribution without knowing the inputs. More generally, if the information cost is very small, $O(1)$ bits, then we can sample the transcript without looking at the inputs, and have the players look at their inputs and "correct the mistakes" afterwards without using a lot of communication (the manner in which we do this is described in Sections 3.3 and 3.4). When the protocol has high information cost, $\mathrm{IC}(\Pi) = \omega(1)$, we reduce to the case of constant information cost by chopping the transcript up into pieces that each reveals a constant amount of information: this way, we get $O(\mathrm{IC}(\Pi))$ "pieces" that each reveal $\Theta(1)$ information, and we can compress each "piece" separately.

It is not trivial to "chop up the transcript" into pieces with $\Theta(1)$ information cost, because we may not *know* a-priori how much information has been revealed at each point. For example, suppose that player 1 gets two bits of input, $a, b \in \{0,1\}$, which are uniform and independent. If $a = 0$, then player 1 sends a uniformly random bit, and in this case it reveals zero information about its input; if $a = 1$, player 1 sends the bit b, revealing one bit of information about its input. Only player 1 knows how much information it has revealed, and in general all the players could behave this way; therefore the players need to cooperate to cut up the transcript into pieces that each reveal $\Theta(1)$ information.

Next we describe more formally what we *mean* by "the information revealed" up to some point in the protocol, and how this notion relates to our ability to sample transcripts without looking at the input.

3.1 The divergence tree

We view the run of Π on an input $x = (x_1, \ldots, x_k)$ as a *binary tree* representing all possible transcripts of Π. Since Π is randomized, it induces a *distribution* π_x on the leafs of the tree; our goal is to sample a *leaf* of the tree (that is, a transcript of Π) from a distribution that is close to π_x, but using as little communication as possible. In the sequel we freely interchange *transcripts* with *nodes* of the transcript tree. For convenience we introduce the following short-hand notation: $\pi_{\leq r}, \pi_{<r}$ denote the distribution of the first r or the first $(r-1)$ bits of Π, respectively. To denote a distribution η conditioned on an event of the form $\boldsymbol{A} = a$, we write $\eta|a$, and for a specific value y, we write its probability under $\eta|a$ as $\eta(y|a)$. (When we condition on multiple values we write, e.g., $\eta|a, b$).

At each node v of the tree, there is some player $O(v)$ whose turn it is to speak when we reach node v. We call this player the *owner* of node v. The two children of node v correspond to the case where player $O(v)$ writes 0 and 1 on the board, respectively.

The owner $O(v)$ of v knows the correct distribution over children of v induced by π_x, because this distribution depends only on its input (the player determines which child we will go to by speaking). We denote by c_{vx} this "correct" distribution; formally, for each $b \in \{0,1\}$, if v is a node at depth r, then

$$c_{vx}(b) = \Pr\left[\boldsymbol{\Pi}_{\leq r+1} = v \cdot b \mid \boldsymbol{\Pi}_{\leq r} = v, \boldsymbol{X} = x\right].$$

(Note that actually this probability only depends on the input $x_{O(v)}$ of the player that owns v, because what a player decides to write on the board depends only on its input and what was written so far.)

The other players and the observer do not know c_{vx}, because they do not know x (only their own private input). But they know the *prior* distribution c_v on the children of node v, which is simply the probability over the protocol's randomness *and the input*, that player $O(v)$ will write 0 (resp. 1), given that we reached node v. Formally, for $b \in \{0,1\}$,

$$c_v(b) = \Pr\left[\boldsymbol{\Pi}_{\leq r+1} = v \cdot b \mid \boldsymbol{\Pi}_{\leq r} = v\right] = \sum_x \left(\Pr_\mu\left[\boldsymbol{X} = x \mid \boldsymbol{\Pi}_{\leq r} = v\right] \cdot c_{vx}(b)\right).$$

(Here again r is the depth of node v.)

Re-written in short-hand notation, we have

$$c_{vx}(b) = \pi_{\leq r+1}(vb|v,x), \qquad c_v(b) = \pi_{\leq r+1}(vb|v) = \sum_x \left(\mu(x|v) c_{vx}(b)\right).$$

3.2 Relating the information cost to the tree

Intuitively, if Π has low information cost, then the true distribution c_{vx} and the prior c_v should be "close" for most nodes v in the tree corresponding to input x, because the message each player decides to write on the board does not depend strongly on its input (otherwise it would reveal a lot of information about the input). And this is made formal by recalling that $\mathrm{IC}_\mu(\Pi) = \mathrm{I}_\mu(\boldsymbol{\Pi}; \boldsymbol{X})$, and using the chain rule and the relationship between mutual

information and divergence to see that

$$\begin{aligned}\underset{\mu}{\mathrm{IC}}(\Pi) &= \sum_{r=1}^{\mathrm{CC}(\Pi)} \underset{\mu}{\mathrm{I}}(\boldsymbol{\Pi}_r; \boldsymbol{X} \mid \boldsymbol{\Pi}_{<r}) = \sum_{r=1}^{\mathrm{CC}(\Pi)} \mathop{\mathrm{E}}_{v\sim\pi_{<r}} \left[\mathop{\mathrm{E}}_{x\sim\mu|v} \left[\mathsf{D}\left(\pi_r|v, x \parallel \pi_r|v\right)\right]\right] \\ &= \sum_{r=1}^{\mathrm{CC}(\Pi)} \mathop{\mathrm{E}}_{v\sim\pi_{<r}} \left[\mathop{\mathrm{E}}_{x\sim\mu|v} \left[\mathsf{D}\left(c_{vx} \parallel c_v\right)\right]\right]. \end{aligned} \tag{1}$$

So, what we "pay" at each round is the expected divergence between the true distribution c_{vx} and the prior c_v. Call this quantity the *divergence cost of* v on input x: $\mathrm{D}_x(v) = \mathsf{D}\left(c_{vx} \parallel c_v\right)$. We extend this to paths $p = v_0, \ldots, v_s$ in the natural way: the cost of the path is the sum of the costs of the nodes on the path, that is, $\mathrm{D}_x(p) = \sum_{i=0}^{s} \mathsf{D}\left(c_{v_i x} \parallel c_{v_i}\right)$. By re-arranging the order of the expectations in (1) we get that $\mathrm{IC}_\mu(\Pi) = \mathrm{E}_{x\sim\mu, t\sim\pi|x}\left[\mathrm{D}_x(t)\right]$, where here t is a complete transcript, viewed as path from the root of the tree to a leaf. Thus, protocols with low information cost have on average a low divergence cost on paths from the root to a leaf, so at most nodes on the path, the true distribution c_{vx} is close to the prior c_v in divergence. We rely on this characterization in our compression scheme.

A key technical ingredient in the compression algorithm is *rejection sampling*, which we review below.

3.3 Rejection sampling

Suppose we want to sample from some distribution p, but we only have access to samples generated from another distribution q. If we know an upper bound $M \geq \max_w p(w)/q(w)$ on the ratio between p and q, we can use *rejection sampling*, which works as follows:

1. Generate a candidate sample $w \sim q$.
2. *Accept* w with probability $p(w)/(M \cdot q(w))$, and otherwise *reject* w and try again.

It is not hard to show that the probability that for any w, the probability that w is generated by the procedure above is exactly $p(w)$: informally, we sample from q, but then "self-correct" by accepting the sample only with probability $p(w)/(M \cdot q(w))$, so the probability that the candidate is w and we accept it is $q(w) \cdot (p(w)/(Mq(w)) = p(w)/M$. Also, at each attempt, the probability that we accept the candidate is

$$\sum_w \left(q(w) \cdot \frac{p(w)}{Mq(w)} \right) = \frac{1}{M} \sum_w p(w) = \frac{1}{M},$$

so the distribution generated *given* that we accepted is exactly p. Moreover, the number of attempts required until we accept a candidate is $O(M)$ in expectation.

For our purposes we use rejection sampling as follows:

- The distribution q that we know how to sample from is the *prior* distribution π on leafs (or later on, internal nodes) of the tree, which all players know, because it does not depend on the input. (This is the distribution where at each node v we sample a child from the prior c_v.) We can sample from this distribution simply using the public randomness.
- The distribution p that we *want* to sample from is the true distribution π_x on leafs (obtained by taking c_{vx} at each node v), which is not known to any player.

We will show that the players can *approximate* the ratio $\pi_x(t)/\pi(t)$ for any leaf (or internal node) t. This allows us to first sample $t \sim \pi$ from the prior, and then reject it with *approximately* the right probability $\pi_x(t)/(M\pi(t))$, where M is an appropriately chosen

normalization factor. In this way we generate leafs from a distribution that is *close* to the true distribution $\pi_x(t)$.

(A major difference between our work and [2] is that in [2], where there are only two players, they do not need to explicitly compute the ratio $\pi_x(t)/\pi(t)$; they use a clever trick that rejects with the right probability. For a general number of players $k \geq 2$, we can no longer do this, and we must compute an approximation of the ratio and reject with that probability.)

To instantiate this outline, we must answer the following questions:

1. What is a good upper bound $M \geq \max_t \pi_x(t)/\pi(t)$? Actually, we will not be able to find a perfect bound, only a *high probability* bound which holds for most t.
2. How do we compute or approximate the ratio $\pi_x(t)/(M\pi(t))$?

3.4 Compressing protocols with constant information cost

To start with, suppose we have a protocol Π with $\mathrm{IC}_\mu(\Pi) = I = O(1)$, that is, a typical path from the root to a leaf of Π incurs only constant divergence cost. In this case we can compress Π to a protocol Π' with communication $2^{O(I)}$. This initially looks very bad, but since $I = O(1)$, in fact the communication cost of Π' is also $2^{O(1)} = O(1)$, regardless of the communication cost of the original protocol Π.

Recall that the divergence between two distributions p, q is defined as

$$\mathsf{D}\left(p \parallel q\right) = \mathop{\mathrm{E}}_{w\sim p}\left[\log \frac{p(w)}{q(w)}\right],$$

that is, $\mathsf{D}\left(p \parallel q\right)$ is the *expected log-ratio* between p and q when we sample from p. This means that if $\mathsf{D}\left(p \parallel q\right) = d$ and we sample $w \sim p$, then with good probability we will have $p(w)/q(w) \leq 2^{O(d)}$ [4]. Therefore we can use $2^{O(d)} = 2^{O(I)}$ as our (high-probability) upper bound M in the rejection sampling scheme, which leads to an expected communication cost of $2^{O(I)}$, times the cost of a single attempt of rejection sampling (that is, sampling a candidate using public randomness, and then deciding whether to accept it).

It remains to describe how we approximate the acceptance probability of a leaf t, which should be $\pi_x(t)/(M\pi(t))$. Here we use the following observation about k-party protocols, which generalizes the corresponding observation for two players from [2]: for each player i, let π_x^i be the distribution where at each node v we select a child with probability c_{vx} if player i owns node v, and with probability c_v otherwise. Each player i can compute $\pi_x^i(t)$ for any leaf t. And if we take the *product* of the $\pi_x^i(t)$'s, we get:

$$\prod_{i=1}^{k} \pi_x^i(t) = \prod_{r=0}^{\mathrm{CC}(\Pi)-1} \left(c_{t_r x}(t_{r+1}) \cdot c_{t_r}(t_{r+1})^{k-1}\right),$$

where t_r is the r-th node on the path from the root to t; this is because exactly one player, the owner of t_r, uses $c_{t_r x}$ to select a child at t_r, and the other players use c_{t_r}. Now let $f_i(t) = \pi_x^i(t)/\pi(t)$ for each $i = 1, \ldots, k$. Then the product of the f_i's is

$$\prod_{i=1}^{k} f_i(t) = \frac{\prod_{r=0}^{\mathrm{CC}(\Pi)-1} c_{t_r x}(t_{r+1}) \cdot c_{t_r}(t_{r+1})^{k-1}}{\prod_{r=0}^{\mathrm{CC}(\Pi)-1} c_{t_r}(t_{r+1})^{k}} = \prod_{r=0}^{\mathrm{CC}(\Pi)-1} \frac{c_{t_r x}(t_{r+1})}{c_{t_r}(t_{r+1})} = \frac{\pi_x(t)}{\pi(t)},$$

giving us exactly the ratio we want.

Thus, in order to implement the rejection sampling, we need to approximate the product $\prod_{i=1}^{k} f_i(t)$, and use it to get an approximate acceptance probability. We require *very* good precision: the approximation needs to be to within a factor $(1 \pm \epsilon)$, where $\epsilon = O(1/I)$ (for

now, I is constant, but later it will not be). This is not hard: we can estimate the product to within $(1 \pm \epsilon)$ by estimating $\sum_{i=1}^{k} \log f_i(t)$ to within $(1 \pm O(\epsilon))$, but first discarding from the sum terms that have very small absolute value, $|\log f_i(t)| < O(\epsilon/k)$. Because $2^x = 1 + O(x)$ for very small x, even if we ignore all the small terms, they only add up to $O(\epsilon)$ in absolute value, and we still get a $(1 \pm \epsilon)$-multiplicative approximation.

For the larger terms, $\log f_i$ such that $|\log f_i| \geq \epsilon/k$, we estimate each one of them to within $(1 \pm \epsilon)$ by dividing the range of possibilities for their absolute value, $[\epsilon/k, M]$, into intervals of exponentially-increasing width, where each interval is $(1 + \epsilon)$ the size of the preceding interval. We then have each player i tell us which interval their contribution $\log f_i(t)$ lies in, and its sign. Combining all the estimates of the individual contributions, we come up with a $(1 \pm \epsilon)$-approximation to the product $\prod_{i=1}^{k} f_i(t)$.

Using an approximation instead of the exact value of the acceptance probability means that the rejection sampling does not generate the correct distribution π_x. However, we prove that if we use a $(1 \pm \epsilon)$-approximation for the acceptance probability, then the distribution generated is $O(\epsilon)$-close to π_x in statistical distance, so that our compression scheme generates a distribution that is very close to the correct one. In the shared blackboard model, the cost of the approximation is $O(k \log 1/\epsilon)$ bits of communication.

3.5 Compression for protocols with large information cost

Following [2], compression for protocols with constant information cost can be extended to protocols with higher information cost as follows: we "chop up" each path in the protocol tree into segments with divergence cost $\Theta(1)$ each. This induces a set of *frontiers* in the tree, where each frontier intersects each path from the root at exactly one node. Instead of directly sampling a complete transcript, which corresponds to directly sampling leaf of the tree, we sample the transcript in segments: first we sample a node from the first frontier, then a node from the second frontier, and so on, until we reach a leaf. Because the divergence cost of each segment is constant, we can use rejection sampling as outlined above for constant divergence cost. And since each frontier "consumes" $\Omega(1)$ of the total divergence cost of the path, and we know that the total divergence cost is $O(I)$ with high probability, the total number of steps is $O(I)$.

Our overall compression scheme is as follows: we start from the root of the tree, and while we have not yet reached a leaf, we sample a node from the next frontier, using rejection sampling to sample very close to the correct distribution induced by π_x. The cost of each such step is $\tilde{O}(k)$, so the total cost of sampling a leaf of the tree is $\tilde{O}(I \cdot k)$.

Next we give a more precise definition of the frontiers, and show how we can sample a node from the next frontier.

3.5.1 Frontiers

The definition of a frontier is subtle (and differs from [2] significantly; having only two players makes life much easier). The main question is: how can we define the frontier in a way that is precise enough so that each path segment between two frontiers has divergence cost $\Theta(1)$, but on the other hand allows us the players to *find* the frontier using only $\tilde{O}(k)$ communication? In particular, we cannot afford to have the players send real numbers with high precision. So how can we identify the point where *together* the players' divergence cost has reached $\Theta(1)$?

Fix a parameter β, which is the target divergence cost we want each path segment between two frontiers to have. Denote by $\mathrm{D}_i(v,w)$ the *individual divergence cost* of player i, defined as the sum of the divergence costs on the path from v to u, but only at nodes owned by player i. Then by definition, $\mathrm{D}(v,w) = \sum_{i=1}^{k} \mathrm{D}_i(v,w)$. Moreover, each player can compute $\mathrm{D}_i(v,w)$, because it knows the divergence cost at nodes it owns: it knows both the true distribution and the prior, and can compute the divergence between them. Finally, let $\tilde{\mathrm{D}}_i(v,w) = \lfloor \mathrm{D}_i(v,w)/(\beta/k) \rfloor$ denote the player i's divergence cost $\mathrm{D}_i(v,w)$ divided by β/k and rounded down to the nearest integer.

Formally, we define the *frontier* of a node v in the tree to be the $\mathcal{F}_{vx}$ set of nodes w such that

1. For each strict prefix w' of the path from v to w we have $\sum_{i=1}^{k} (\beta/k)\tilde{\mathrm{D}}_i(v,w') < \beta$, but
2. For the full path we have $\sum_{i=1}^{k} (\beta/k)\tilde{\mathrm{D}}_i(v,w) \geq \beta$.

That is, the frontier is the first point on the path where the sum of the divergence costs of the players, each rounded down to the nearest β/k, first exceeds β.

Each player can compute its own contribution $\tilde{\mathrm{D}}_i(v,w)$ given v and w and announce a constant multiplicative approximation of it. This is good enough to ensure that for any frontier node $w \in \mathcal{F}_{vx}$ we have $\mathrm{D}(v,w) = \Theta(\beta)$.

(In [2], the frontier is defined as the point where *one* of the two players first reaches divergence cost $\mathrm{D}_i(v,w) \geq \beta$. Since there are only two players, this also means that the *total* divergence cost is $\Theta(\beta)$. This has the advantage that in order to check if a node is on the frontier, all we need to do is ask the two players whether they have individual divergence cost at most β or not, costing a single bit per player. In our case, if we tried to take this approach, the total divergence cost could be as high as $\Theta(k\beta)$, which we cannot afford, as our probability of accepting in the rejection sampling would then be only $2^{-\Theta(k\beta)}$. Thus, we must define the frontier by the *total* divergence cost directly, and this leads to technical complications.)

We define two distributions on the frontier $\mathcal{F}_{vx}$: the first, F_{vx}, is the "correct" distribution induced by π_x, where we sample a path from v to a leaf, $t \sim \pi_x|v$, and cut the path at the (unique) point where it intersects $\mathcal{F}_{vx}$. The second distribution, denoted F_v, is the *prior* distribution, induced by the prior π in the same way.

3.5.2 Sampling from the frontier

Suppose we are currently at node v in the tree, and we want to sample from the correct distribution F_{vx} on the frontier at node v. Let us recall the ingredients we need to sample from the frontier using rejection sampling:

1. We need to sample a frontier node $w \sim F_v$,
2. We need to compute or approximate the acceptance probability, $F_{vx}(w)/(MF_v(w))$.

We already saw how to approximate the acceptance probability in Section 3.4. Here it becomes important to set the precision parameter ϵ to $O(1/I)$, because we will be passing through $O(I)$ frontiers, and the error adds up; we can only afford an error of $O(1/I)$ per frontier to get constant error in the end.

It remains to describe how we can sample a frontier node from the prior, $w \sim F_v$. To do this, we first sample a leaf $t \sim \pi|v$, using public randomness (since $\pi|v$ is known to all players). Next, we find the point where the frontier $\mathcal{F}_{vx}$ intersects the path from v to t, and return this node. In order to find the cut-point of the frontier we use binary search (as in [2]), to find the first node w on the path from v to t where we have $\sum_{i=1}^{k} (\beta/k)\tilde{\mathrm{D}}_i(v,w) \geq \beta$.

The total cost of sampling from $w \sim F_v$ is bounded by $\log \mathrm{CC}$ times the cost of approximating $\sum_{i=1}^{k} (\beta/k)\tilde{\mathrm{D}}_i(v,w) \geq \beta$ to within a constant multiplicative factor ϵ, which is $O(k \log(k/\epsilon))$ bits.

4 Compression Lower Bound

In this section we present the $\texttt{AndTree}_h$ function, parameterized by h, with low information cost, $O(h \log k)$, but high communication cost, $\tilde{\Omega}(k \cdot h)$. This function serves to *separate* information from communication in the shared blackboard model, ruling out the existence of a compression scheme whose cost does not depend nearly-linearly on k.

4.1 The AndTree problem

The input to $\texttt{AndTree}_h$ is represented by a complete binary tree of depth h; at each node u of the tree, we embed an instance of AND, with each player i receiving a bit $\boldsymbol{X}_u^i$. We represent each node of the tree by the path from the root, with the root denoted λ, a left child appending zero to its parent, and a right child appending one to its parent.

For a tree T, let $\mathsf{leaf}(T)$ be the leaf obtained by starting at the root, turning left at each node u such that $\bigwedge_i \boldsymbol{X}_u^i = 0$, and turning right at each node u such that $\bigwedge_i \boldsymbol{X}_u^i = 1$. We define the output of the $\texttt{AndTree}_h$ problem to be the *parity* of this leaf: $\texttt{AndTree}_h(T) = \mathrm{parity}(\mathsf{leaf}(T))$.

4.2 The information cost of AndTree

A single instance of AND can be solved with $O(\log k)$ bits of information under any input distribution: simply have the players announce their inputs one-by-one, until we either find a player with input 0, in which case we halt and output 0, or all players have announced that their input is 1, in which case we output 1. This protocol has only $k+1$ possible transcripts: if all players got 1, the transcript is 1^k, and otherwise the transcript is $1^i 0$, where i is the index of the first player that got 0. Therefore the *entropy* of the transcript is $O(\log k)$, meaning that the information cost of the protocol is also $O(\log k)$.

To solve a full instance of $\texttt{AndTree}$, we start at the root, and use the protocol above to solve each node, moving down to the correct child. When we arrive at a leaf, we output its parity. The information cost is $O(h \log k)$ under *any* input distribution.

4.3 The communication complexity of AndTree

Next we must show that no randomized protocol with $o(k \cdot I)$ communication can solve $\texttt{AndTree}$ with small error in the worst-case. We show a slightly stronger lower bound: we give an input distribution μ, and show that no randomized protocol with $o(k \cdot I)$ communication can solve $\texttt{AndTree}$ with small *average* error when inputs are drawn from μ.

For the analysis, we divide the tree into R layers, each of depth $d = \Theta(\log k + \log h)$. We define the input distribution μ_R on an R-layer tree as follows: for each node u of the tree, we draw a random player $\boldsymbol{P}(u)$ uniformly and independently. The input of player $\boldsymbol{P}(u)$ is drawn uniformly, $\boldsymbol{B}_u \sim \mathsf{U}(\{0,1\})$; the other players receive 1. Thus, under μ_R, for each node u we have $\bigwedge_i \boldsymbol{X}_u^i = \boldsymbol{X}_u^{\boldsymbol{P}(u)} = \boldsymbol{B}_u$.

It is convenient to represent an r-layer tree as $T = (S, R_1, \ldots, R_{2^d})$, where S is the top layer (i.e., the first d levels of the tree), and $R_1, \ldots, R_{2^d}$ are the subtrees starting at the leafs of S.

▶ **Theorem 4.** *For sufficiently small constant ϵ, the randomized public-coin communication complexity of $\texttt{AndTree}_h$ is $\mathrm{CC}(\texttt{AndTree}_h, \epsilon) = \Omega(R \cdot k)$, where $R = \Theta(h/(\log k + \log h))$.*

We prove the theorem by induction on the number of layers: for each $r \le R$, given an protocol with communication C and error ϵ for the r-layer tree, we "peel off" a layer, and

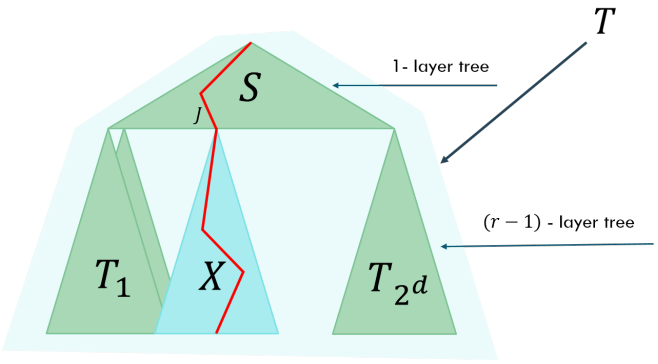


Figure 1 Embedding X in an r-layer tree

construct a protocol with communication $C - \Theta(k)$ and error $\epsilon + \Theta(1/R)$ for the $(r-1)$-layer tree. If we started out with communication less than $c \cdot R \cdot k$ for some sufficiently small constant c, then eventually we obtain a protocol with zero communication and constant error for the one-layer tree, which is impossible.

The induction step. Let $d = \beta(\log k + \log h)$ be the height of a layer, where β is a constant whose value will be fixed later.

Given a protocol Π for the r-layer tree $\texttt{AndTree}_{r \cdot d}$, with communication cost $\mathrm{CC}(\Pi) = C$ and distributional error ϵ on μ_r, we construct a new protocol, $\tilde{\Pi}$, for $(r-1)$-layer trees.

Embedding in an r-layer tree

The input to $\tilde{\Pi}$ is an $(r-1)$-layer tree $\boldsymbol{X}$, but the protocol Π that we are given takes as input r-*layer* trees. How can we use Π to solve our input, which has one fewer layer? The answer is that we *embed* our $(r-1)$-layer input in a larger r-layer tree $\boldsymbol{T}$, the rest of which we generate using public randomness. We design the embedding so that the answer on the input $\boldsymbol{X}$ can be extracted from the answer on $\boldsymbol{T}$.

More formally, we embed $\boldsymbol{X}$ in an r-layer tree by publicly generating a random "top layer", $\boldsymbol{S} \sim \mu_1$, and placing $\boldsymbol{X}$ as the $\boldsymbol{J}$-th subtree of $\boldsymbol{S}$, where $\boldsymbol{J}$ is the "correct" leaf of $\boldsymbol{S}$, obtained by solving the AND at each node and turning left when the answer is 0 and right when the answer is 1. Note that by definition, $\mathrm{parity}(\boldsymbol{J}) = \texttt{AndTree}(\boldsymbol{S})$.

The other $2^d - 1$ subtrees, placed under the leafs of $\boldsymbol{S}$ that are not the "correct" leaf $\boldsymbol{J}$, are generated publicly and independently from μ_{r-1}. Let $\boldsymbol{T}_1, \ldots, \boldsymbol{T}_{2^d}$ denote these subtrees. Then formally, what we said above is that we set $\boldsymbol{T}_{\boldsymbol{J}} = \boldsymbol{X}$, and we sample $(\boldsymbol{T}_1, \ldots, \boldsymbol{T}_{\boldsymbol{J}-1}, \boldsymbol{T}_{\boldsymbol{J}+1}, \ldots, \boldsymbol{T}_{2^d}) \sim (\mu_{r-1})^{2^d-1}$.

Let $\boldsymbol{T} = (\boldsymbol{S}, \boldsymbol{T}_1, \ldots, \boldsymbol{T}_{2^d})$ be the resulting r-layer tree.

▶ **Property 5.** *Observe that since* $\texttt{AndTree}_d(\boldsymbol{S}) = \mathrm{parity}(\boldsymbol{J})$*, and we set* $\boldsymbol{T}_{\boldsymbol{J}} = \boldsymbol{X}$*, we have* $\texttt{AndTree}_{(r-1)d}(\boldsymbol{X}) = \texttt{AndTree}_{(r-1)d}(\boldsymbol{T}_{\boldsymbol{J}}) = \texttt{AndTree}_{rd}(\boldsymbol{T}) \oplus \mathrm{parity}(\boldsymbol{J})$.

We can now solve our $(r-1)$-layer input $\boldsymbol{X}$ by calling Π on the r-layer tree $\boldsymbol{T}$ that we constructed, but this is not enough: for the induction step we need to construct a protocol with *less communication* than Π. Our goal is therefore to simulate the execution of Π on $\boldsymbol{T}$, but using less communication than Π requires. If we can do this, then we can solve $\texttt{AndTree}_{(r-1)d}$ using less communication.

Saving $\Theta(k)$ bits of communication

We split the transcript of the original protocol Π into two parts, $\boldsymbol{\Pi} = \boldsymbol{\Pi}_1\boldsymbol{\Pi}_2$, where the prefix $\boldsymbol{\Pi}_1$ is of length $|\boldsymbol{\Pi}_1| = \alpha k$ for a constant α whose value will be fixed later, and the suffix $\boldsymbol{\Pi}_2$ consists of the rest of the transcript.

In $\tilde{\Pi}$, instead of sampling $\boldsymbol{\Pi}_1$ "correctly" by having each player look at their input and send the messages indicated under Π, we *fix* a prefix m_1 of length αk, and have the players sample $\boldsymbol{\Pi}_2$ as though they had said m_1; that is, the transcript of $\tilde{\Pi}$ is the suffix $\Pi \sim \pi_2|\boldsymbol{\Pi}_1 = m_1, \boldsymbol{J} = j$. We need to show that there is a "good" choice for m_1 and j, under which the suffix has small error probability on our real input $\boldsymbol{X}$.

▶ **Definition 6.** We say that a pair (m_1, j) is *good*, where $m_1 \in \{0,1\}^{\alpha k}, j \in [2^d]$, if the following holds:

1. $\mathsf{D}\left(\mu_{r-1}(\boldsymbol{T}_j|\boldsymbol{\Pi}_1 = m_1, \boldsymbol{J} = j) \parallel \mu_{r-1}(\boldsymbol{T}_j)\right) \leq \frac{1}{100R^2}$, and,
2. $\Pr\left[\Pi \text{ errs} \mid \boldsymbol{\Pi}_1 = m_1, \boldsymbol{J} = j\right] \leq \left(1 + \frac{1}{R}\right) \cdot \epsilon$.

Note that since $\boldsymbol{T}_j$ is independent of $\boldsymbol{J}$, the first condition can also be written as:

$$\mathsf{D}\left(\mu_{r-1}(\boldsymbol{T}_j|\boldsymbol{\Pi}_1 = m_1, \boldsymbol{J} = j) \parallel \mu_{r-1}(\boldsymbol{T}_j|\boldsymbol{J} = j)\right) \leq \frac{1}{100R^2}.$$

▶ **Lemma 7.** *There exists a good pair (m_1, j).*

Before proving that there exists a good setting for (m_1, j), let us show that if (m_1, j) *is* good, then when we sample $\boldsymbol{\Pi}_2 \sim \pi_2|\boldsymbol{\Pi}_1 = m_1, \boldsymbol{J} = j$ we have small error on our $(r-1)$-layer input, $\boldsymbol{T}_j = \boldsymbol{X}$.

▶ **Lemma 8.** *If (m_1, j) is good, then* $\Pr_{\boldsymbol{X}\sim\mu_{r-1}}\left[\boldsymbol{\Pi} \text{ errs on } \boldsymbol{X} \mid \boldsymbol{J} = j, \boldsymbol{\Pi}_1 = m_1\right] \leq \left(1 + \frac{1}{R}\right) \cdot \epsilon + \frac{1}{10R}$.

Proof sketch. Let μ' be the distribution $\mu_{r-1}(\boldsymbol{T}_j|\boldsymbol{\Pi}_1 = m_1, \boldsymbol{J} = j)$ of the j-th subtree, given $\boldsymbol{\Pi}_1 = m_1$ and $\boldsymbol{J} = j$. Let $\boldsymbol{E} = \boldsymbol{E}(\boldsymbol{X})$ be an indicator for the event that the output produced by the suffix $\boldsymbol{\Pi}_2$ is incorrect on $\boldsymbol{X}$. Finally, let π_2 be the distribution of $\boldsymbol{\Pi}_2$ given $\boldsymbol{\Pi}_1 = m_1, \boldsymbol{J} = j$ when the input is $\boldsymbol{X} \sim \mu_{r-1}$, and let π_2' be the distribution of $\boldsymbol{\Pi}_2$ given $\boldsymbol{\Pi}_1 = m_1, \boldsymbol{J} = j$ when the input is $\boldsymbol{X} \sim \mu'$.

Re-stated in this notation, the lemma asserts that $\Pr_{\pi_2}[\boldsymbol{E}] \leq \left(1 + \frac{1}{R}\right) \cdot \epsilon + \frac{1}{10 \cdot R}$, and Condition 2 of Definition 6 says that $\Pr_{\pi_2'}[\boldsymbol{E}] \leq \left(1 + \frac{1}{R}\right) \cdot \epsilon$. Thus, to show the lemma, we show that $\pi_2(\boldsymbol{E})$ and $\pi_2'(\boldsymbol{E})$ are "close", and therefore the expectation of $\boldsymbol{E}$ cannot differ by much between them. We get the required "closeness" from the first condition of 6, which bounds the divergence between the two distributions. ◀

▶ **Corollary 9.** *Let $\tilde{\boldsymbol{\Pi}}$ be the protocol defined by taking a good pair (m_1, j), embedding the input in location $\boldsymbol{T}_j$, sampling the suffix $\boldsymbol{\Pi}_2$ from its distribution given $\boldsymbol{\Pi}_1 = m_1, \boldsymbol{J} = j$ (the missing parts of the input are sampled from public randomness according to m_1 and j) and returning ($\boldsymbol{\Pi}_2'$s output) $\oplus$ parity$(\boldsymbol{S})$. Then* $\Pr_{\boldsymbol{X}\sim\mu_{r-1}}\left[\tilde{\boldsymbol{\Pi}} \text{ errs on } \boldsymbol{X}\right] \leq \left(1 + \frac{1}{R}\right) \cdot \epsilon + \frac{1}{10R}$.

Now let us sketch the proof that with high probability there is a good pair (m_1, j).

Proof sketch of Lemma 7. We show that the probability over $\boldsymbol{M}_1$ and $\boldsymbol{J}$ that *either* the first or the second condition of Definition 6 fails to hold is smaller than 1, which means that there is a pair (m_1, j) satisfying both conditions.

First consider the first condition, which requires that m_1 does not reveal a lot of information about the subtree $\boldsymbol{T}_j$. Since m_1 is short, only αk bits, the total information it reveals about *all* the sub-trees $\boldsymbol{T}_1, \ldots, \boldsymbol{T}_{2^d}$ together is at most $O(\alpha k)$ with high probability. The sub-trees $\boldsymbol{T}_1, \ldots, \boldsymbol{T}_{2^d}$ are initially independent, and information has the *super-additivity property*:

if $\boldsymbol{A}_1, \ldots, \boldsymbol{A}_m$ are independent, then for any $\boldsymbol{B}$, $\mathrm{I}(\boldsymbol{B}; \boldsymbol{A}_1, \ldots, \boldsymbol{A}_m) \geq \sum_{i=1}^{m} \mathrm{I}(\boldsymbol{B}; \boldsymbol{A}_i)$. In our case this means that if we choose a *uniformly random* index $\boldsymbol{J} \in \{1, \ldots, 2^d\}$ which is independent of m_1, the information m_1 reveals about $\boldsymbol{T}_{\boldsymbol{J}}$ is at most $O(\alpha k)/2^d = O(\alpha k)/\operatorname{poly}(k)$, which is negligible.

Unfortunately, the choice of the index $\boldsymbol{J}$ where we embed our input is *not* independent of m_1: there is a hypothetical possibility that the players can discover where the input is embedded by computing the correct leaf of the top layer $\boldsymbol{S}$ (which is how we defined $\boldsymbol{J}$). If they can do this, then they can focus all their attention on the correct sub-tree $\boldsymbol{T}_{\boldsymbol{J}}$, and m_1 can reveal a lot of information about it. We need to show that since m_1 is short, the players *cannot* discover $\boldsymbol{J}$.

To capture the fact that the players cannot learn $\boldsymbol{J}$, except with small probability, we use the notion of *min-entropy*: for a random variable $\boldsymbol{A} \sim \eta$ over domain Ω, the min-entropy of $\boldsymbol{A}$ is defined as $\mathrm{H}_\infty(\boldsymbol{A}) = \min_{\omega \in \Omega} \log \frac{1}{\eta(\omega)}$. The min-entropy corresponds to our ability to *guess* the correct value of $\boldsymbol{A}$ (unlike Shannon entropy).

We use a lemma from [13] which generalizes the super-additivity of information to variables with high min-entropy, and asserts that

$$\begin{aligned} &\mathop{\mathrm{E}}_{\boldsymbol{J}|\boldsymbol{\Pi}_1 = m_1} \left[\mathsf{D}\left(\mu_{r-1}(\boldsymbol{T}_{\boldsymbol{J}}|\boldsymbol{\Pi}_1 = m_1, \boldsymbol{J}) \parallel \mu_{r-1}(\boldsymbol{T}_{\boldsymbol{J}}|\boldsymbol{J})\right)\right] \\ &\leq 2^{-\mathrm{H}_\infty(\boldsymbol{J}|m_1)} \cdot \mathsf{D}\left(\mu_{r-1}^{2^d}(\boldsymbol{T}_1, \ldots, \boldsymbol{T}_{2^d}|\boldsymbol{\Pi}_1 = m_1) \parallel \mu_{r-1}^{2^d}(\boldsymbol{T}_1, \ldots, \boldsymbol{T}_{2^d})\right). \end{aligned} \tag{2}$$

To use this lemma, we must show that with high probability the min-entropy $\mathrm{H}_\infty(\boldsymbol{J}|m_1)$ of the correct leaf given the message m_1 is high. This holds because m_1 consists of only αk bits of communication, which allows only an α-fraction of players to speak; for most nodes u in the top layer $\boldsymbol{S}$, the "influential" player $\boldsymbol{P}(u)$, whose input determines the correct child of u, does not get to say anything at all. Therefore the correct child at most nodes remains uniformly random even after seeing the message m_1, and the correct leaf $\boldsymbol{J}$ retains high min-entropy.

For the second condition of a good pair, we know that the overall error of Π is ϵ, and therefore, by Markov and the law of total expectation,

$$\Pr_{\boldsymbol{\Pi}_1, \mathcal{J}}\left[\Pr\left[\Pi \text{ errs} \mid \boldsymbol{J}, \boldsymbol{\Pi}_1\right] > \left(1 + \frac{1}{R}\right)\epsilon\right] < \frac{\epsilon}{\epsilon(1 + 1/R)} < 1 - \frac{1}{2R}.$$

A union bound over the probabilities that the first and second condition fail to hold yields the lemma. ◀

Proof of Theorem 4. Suppose Π is a protocol for $\texttt{AndTree}_h$ with error $1/100$ and communication C. Let $R = \lfloor h/d \rfloor - 1$. Using Corollary 9, we construct a series of protocols $\Pi_0, \ldots, \Pi_R$, where $\Pi_R = \Pi$ is the protocol we started with, and for each $r > 0$, the protocol Π_r solves $\texttt{AndTree}_{h-(R-r)d}$ with error at most ϵ_r, and its communication cost is $C - \alpha r k$. Applying Corollary 9 R times, we get that the final protocol Π_0, solves $\texttt{AndTree}_{h-Rd}$ with error

$$\epsilon_0 < \left(1 + \frac{1}{R}\right)^R \left(\epsilon_R + R \cdot \frac{1}{10R}\right) < e \cdot \left(\frac{1}{100} + \frac{1}{10}\right) < 1/3.$$

Since $h - Rd > 0$ by definition of R, the problem $\texttt{AndTree}_{h-Rd}$ cannot be solved with error $1/3$ with no communication (indeed, we know that the communication required is $\Omega(k)$ to solve even a single instance of AND), and therefore we must have $C > \alpha R k = \Omega\left(\frac{h}{\log k + \log h} k\right)$. ◀

We have now shown that for any sufficiently large h, the $\texttt{AndTree}_h$ problem can be solved using $O(h \log k)$ bits of information, but requires $\Omega(hk/(\log k + \log h))$ bits of communication. This shows that our compression scheme is optimal up to polylogarithmic factors in k and the input size.

References

1. Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, and D. Sivakumar. An information statistics approach to data stream and communication complexity. *J. Comput. Syst. Sci.*, 68(4):702–732, 2004.
2. Boaz Barak, Mark Braverman, Xi Chen, and Anup Rao. How to compress interactive communication. In *STOC*, pages 67–76, 2010.
3. Balthazar Bauer, Shay Moran, and Amir Yehudayoff. Internal compression of protocols to entropy. In *APPROX/RANDOM*, pages 481–496, 2015.
4. Mark Braverman. Interactive information complexity. In *STOC*, pages 505–524, 2012.
5. Mark Braverman and Rotem Oshman. On information complexity in the broadcast model. In *PODC*, pages 355–364, 2015.
6. Mark Braverman and Anup Rao. Information equals amortized communication. In *FOCS*, pages 748–757, 2011.
7. Joshua Brody, Harry Buhrman, Michal Koucký, Bruno Loff, Florian Speelman, and Nikolay K. Vereshchagin. Towards a reverse newman's theorem in interactive information complexity. In *CCC*, pages 24–33, 2013.
8. Amit Chakrabarti, Yaoyun Shi, Anthony Wirth, and Andrew Chi-Chih Yao. Informational complexity and the direct sum problem for simultaneous message complexity. In *FOCS*, pages 270–278, 2001.
9. Robert M Fano. *The transmission of information*. Massachusetts Institute of Technology, Research Laboratory of Electronics, 1949.
10. Prahladh Harsha, Rahul Jain, David A. McAllester, and Jaikumar Radhakrishnan. The communication complexity of correlation. *IEEE Transactions on Information Theory*, 56(1):438–449, 2010.
11. David A Huffman. A method for the construction of minimum redundancy codes. *proc. IRE*, 40(9):1098–1101, 1952.
12. Gillat Kol. Interactive compression for product distributions. In *STOC*, pages 987–998, 2016.
13. Gillat Kol and Ran Raz. Interactive channel capacity. In *STOC*, pages 715–724, 2013.
14. Sivaramakrishnan Natarajan Ramamoorthy and Anup Rao. How to compress asymmetric communication. In *CCC*, pages 102–123, 2015.
15. C. E. Shannon. A mathematical theory of communication. *The Bell Systems Technical Journal*, 27:July 379–423, October 623–656, 1948.
16. Alexander A. Sherstov. Compressing interactive communication under product distributions. In *FOCS*, pages 535–544, 2016.

Self-Stabilising Byzantine Clock Synchronisation is Almost as Easy as Consensus*

Christoph Lenzen[1] and Joel Rybicki†[2]

1 **Max Planck Institute for Informatics, Saarland Informatics Campus, Saarbrücken, Germany**
clenzen@mpi-inf.mpg.de

2 **Department of Biosciences, University of Helsinki, Finland**
joel.rybicki@helsinki.fi

Abstract

We give fault-tolerant algorithms for establishing synchrony in distributed systems in which each of the n nodes has its own clock. Our algorithms operate in a very strong fault model: we require self-stabilisation, i.e., the initial state of the system may be arbitrary, and there can be up to $f < n/3$ ongoing Byzantine faults, i.e., nodes that deviate from the protocol in an arbitrary manner. Furthermore, we assume that the local clocks of the nodes may progress at different speeds (clock drift) and communication has bounded delay. In this model, we study the pulse synchronisation problem, where the task is to guarantee that eventually all correct nodes generate well-separated local pulse events (i.e., unlabelled logical clock ticks) in a synchronised manner.

Compared to prior work, we achieve *exponential* improvements in stabilisation time and the number of communicated bits, and give the first sublinear-time algorithm for the problem:

- In the deterministic setting, the state-of-the-art solutions stabilise in time $\Theta(f)$ and have each node broadcast $\Theta(f \log f)$ bits per time unit. We exponentially reduce the number of bits broadcasted per time unit to $\Theta(\log f)$ while retaining the same stabilisation time.
- In the randomised setting, the state-of-the-art solutions stabilise in time $\Theta(f)$ and have each node broadcast $O(1)$ bits per time unit. We exponentially reduce the stabilisation time to $\operatorname{polylog} f$ while each node broadcasts $\operatorname{polylog} f$ bits per time unit.

These results are obtained by means of a recursive approach reducing the above task of *self-stabilising* pulse synchronisation in the *bounded-delay* model to *non-self-stabilising* binary consensus in the *synchronous* model. In general, our approach introduces at most logarithmic overheads in terms of stabilisation time and broadcasted bits over the underlying consensus routine.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Byzantine faults, self-stabilisation, clock synchronisation, consensus

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.32

1 Introduction

Many of the most fundamental problems in distributed computing relate to timing and fault tolerance. Even though most distributed systems are inherently asynchronous, it is often convenient to design such systems by assuming some degree of synchrony provided by reliable global or distributed clocks. For example, the vast majority of existing Very

* Full version available on arXiv [27], http://arxiv.org/abs/1705.06173.
† Part of this work was done while JR was affiliated with Helsinki Institute for Information Technology HIIT, Department of Computer Science, Aalto University.


© Christoph Lenzen and Joel Rybicki;
licensed under Creative Commons License CC-BY

Large Scale Integrated (VLSI) circuits operate according to the synchronous paradigm: an internal clock signal is distributed throughout the chip neatly controlling alternation between computation and communication steps. Of course, establishing the synchronous abstraction is of high interest in numerous other large-scale distributed systems, as it makes the design of algorithms considerably easier.

However, as the accuracy and availability of the clock signal is typically one of the most basic assumptions, clocking errors affect system behavior in unpredictable ways that are often hard – if not impossible – to tackle at higher system layers. Therefore, *reliably* generating and distributing a joint clock is an essential task in distributed systems. Unfortunately, the cost of providing fault-tolerant synchronisation and clocking is still poorly understood.

Pulse synchronisation. In this work, we study the *self-stabilising Byzantine pulse synchronisation* problem [16, 9], which requires the system to achieve synchronisation despite severe faults. We assume a fully connected message-passing system of n nodes, where

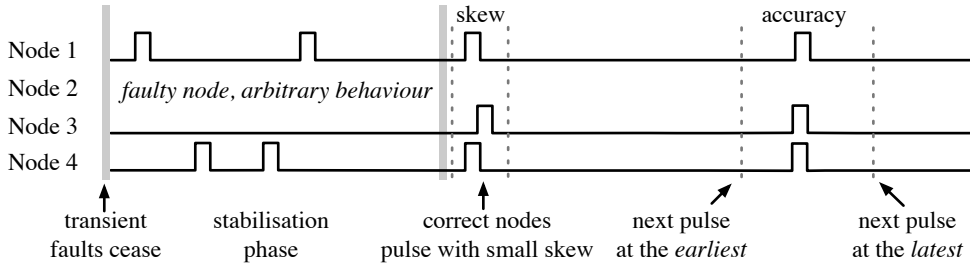
1. an unbounded number of transients faults may occur anywhere in the network, and
2. up to $f < n/3$ of the nodes can be faulty and exhibit *arbitrary* ongoing misbehaviour.

In particular, the transient faults may arbitrarily corrupt the state of the nodes and result in loss of synchrony. Moreover, the nodes that remain faulty may deviate from any given protocol, behave adversarially, and collude to disrupt the other nodes by sending them *different* misinformation even after transient faults have ceased. Note that this also covers faults of the communication network, as we may map faults of communication links to one of their respective endpoints. The goal is now to (re-)establish synchronisation once transient faults cease, despite up to $f < n/3$ Byzantine nodes. That is, we need to consider algorithms that are simultaneously (1) self-stabilising [7, 15] and (2) Byzantine fault-tolerant [23].

More specifically, the problem is as follows: after transient faults cease, no matter what is the initial state of the system, the choice of up to $f < n/3$ faulty nodes, and the behaviour of the faulty nodes, we require that after a bounded *stabilisation time* all the *non-faulty* nodes must generate pulses that

- occur almost simultaneously at each correctly operating node (i.e., have small *skew*), and
- satisfy specified minimum and maximum frequency bounds (*accuracy*).

While the system may have arbitrary behaviour during the initial stabilisation phase due to the effects of transient faults, eventually the above conditions provide synchronised unlabelled clock ticks for all non-faulty nodes:



In order to meet these requirements, it is necessary that nodes can estimate the progress of time. To this end, we assume that nodes are equipped with (continuous, real-valued) hardware clocks that run at speeds that may vary arbitrarily within 1 and ϑ, where $\vartheta \in O(1)$. That is, we normalize minimum clock speed to 1 and assume that the clocks have drift bounded by a constant. Observe that in an asynchronous system, i.e., one in which communication and/or computation may take unknown and unbounded time, even perfect clocks are insufficient to ensure any relative timing guarantees between the actions of different nodes. Therefore, we

additionally assume that the nodes can send messages to each other that are received and processed within at most $d \in \Theta(1)$ time. The clock speeds and message delays can behave adversarially within the respective bounds given by ϑ and d.

In summary, this yields a highly adversarial model of computing, where further restrictions would render the task infeasible: (1) transient faults are arbitrary and may involve the entire network; (2) ongoing faults are arbitrary, cover erroneous behavior of both the nodes and the communication links, and the problem is not solvable if $f \geq n/3$ [10]; and (3) the assumptions on the accuracy of local clocks and communication delay are minimal to guarantee solvability.

Background and related work. If one takes any one of the elements described above out of the picture, then this greatly simplifies the problem. Without permanent faults, the problem becomes trivial: it suffices to have all nodes follow a designated leader. Without transient faults [22], straightforward solutions are given by elegant classics [33, 34], where [34] also guarantees asymptotically optimal skew [29]. Taking the uncertainty of unknown message delays and drifting clocks out of the equation leads to the so-called digital clock synchronisation problem [3, 11, 25, 28, 26], where communication proceeds in synchronous rounds and the task is to agree on a consistent (bounded) round counter. While this abstraction is unrealistic as a basic system model, it yields conceptual insights into the pulse synchronisation problem in the bounded-delay model. Moreover, it is useful to assign numbers to pulses after pulse synchronisation is solved, in order to get a fully-fledged shared system-wide clock [24].

In contrast to these relaxed problem formulations, the pulse synchronisation problem was initially considered to be very challenging – if not impossible – to solve. In a seminal article, Dolev and Welch [16] proved otherwise, albeit with an algorithm having an impractical exponential stabilisation time. In a subsequent line of work, the stabilisation time was reduced to polynomial [6] and then linear in f [12]. However, the linear-time algorithm relies on simulating multiple instances of synchronous *consensus* algorithms [30] concurrently, which results in a high communication complexity.

The consensus problem [30, 23] is one of the fundamental primitives in fault-tolerant computing. Most relevant to this work is synchronous binary consensus with (up to f) Byzantine faults. Here, node v is given an input $x(v) \in \{0,1\}$, and it must output $y(v) \in \{0,1\}$ such that the following properties hold:

1. **Agreement:** There exists $y \in \{0,1\}$ such that $y(v) = y$ for all correct nodes v.
2. **Validity:** If for $x \in \{0,1\}$ it holds that $x(v) = x$ for all correct nodes v, then $y = x$.
3. **Termination:** All correct nodes eventually decide on $y(v)$ and terminate.

In this setting, two of the above main obstacles are not present: the system is properly initialised (no self-stabilisation required) and computation proceeds in synchronous rounds, i.e., well-ordered compute-send-receive cycles. This confines the task to understanding how to deal with the interference from Byzantine nodes. Synchronous consensus is extremely well-studied; see e.g. [32] for a survey. It is known that precisely $\lfloor (n-1)/3 \rfloor$ faults can be tolerated in a system of n nodes [30], $\Omega(nf)$ messages need to be sent in total [14], the connectivity of the communication network must be at least $2f+1$ [8], deterministic algorithms require $f+1$ rounds [19, 1], and randomised algorithms can solve the problem in constant expected time [18]. In constrast, no non-trivial lower bounds on the time or communication complexity of pulse synchronisation are known.

The linear-time pulse synchronisation algorithm in [12] relies on simulating (up to) one synchronous consensus instance for each node simultaneously. Accordingly, this protocol requires each node to broadcast $\Theta(f \log f)$ bits per time unit. Moreover, the use of *de-*

Table 1 Summary of pulse synchronisation algorithms for $f \in \Theta(n)$. For each respective algorithm, the first two columns give the stabilisation time and the number of bits broadcasted by a node per time unit. The third column denotes whether algorithm is deterministic or randomised. The fourth column indicates additional details or model assumptions. All algorithms tolerate $f < n/3$ faulty nodes except for (*), where we have $f < n/(3+\varepsilon)$ for any constant $\varepsilon > 0$.

time	bits	type	notes	reference
poly f	$O(\log f)$	det.		[6]
$O(f)$	$O(f \log f)$	det.		[12]
$O(f)$	$O(\log f)$	det.		this work and [4]
$2^{O(f)}$	$O(1)$	rand.	adversary cannot predict coin flips	[16]
$O(f)$	$O(1)$	rand.	adversary cannot predict coin flips	[9]
polylog f	polylog f	rand.	private channels, (*)	this work and [21]
$O(\log f)$	poly f	rand.	private channels	this work and [18]

terministic consensus is crucial, as failure of any consensus instance to generate correct output within a prespecified time bound may result in loss of synchrony, i.e., the algorithm would fail *after* apparent stabilisation. In [9], these obstacles were overcome by avoiding the use of consensus by reducing the pulse synchronisation problem to the easier task of generating at least one well-separated "resynchronisation point", which is roughly uniformly distributed within any period of $\Theta(f)$ time. This can be achieved by trying to initiate such a resynchronisation point at random times, in combination with threshold voting and locally checked timing constraints to rein in the influence of Byzantine nodes. In a way, this seems much simpler than solving consensus, but the randomisation used to obtain a suitable resynchronisation point strongly reminds of the power provided by shared coins [31, 2, 18, 3] – and this is exactly what the core routine of the expected constant-round consensus algorithm from [18] provides.

Contributions. Our main result is a framework that reduces pulse synchronisation to an arbitrary (non-self-stabilising) synchronous binary consensus routine at very small overheads. In other words, given *any* efficient algorithm that solves consensus in the standard synchronous model of computing, we show how to obtain an efficient algorithm that solves the pulse synchronisation problem in the bounded-delay model with clock drift.

While we build upon existing techniques, our approach has many key differences. First of all, while Dolev et al. [9] also utilise the concept of resynchronisation pulses, these are generated probabilistically. Moreover, their approach has an inherent time bound of $\Omega(f)$ for generating such pulses. In contrast, we devise a new recursive scheme that allows us to (1) *deterministically* generate resynchronisation pulses in $\Theta(f)$ time and (2) *probabilistically* generate resynchronisation pulses in $o(f)$ time. To construct algorithms that generate resynchronisation pulses, we employ resilience boosting and filtering techniques inspired by our recent line of work on digital clock synchronisation in the *synchronous* model [28, 25, 26]. One of its main motivations was to gain a better understanding of the linear time/communication complexity barrier that research on pulse synchronisation ran into, without being distracted by the additional timing uncertainties due to communication delay and clock drift. The challenge here is to port these newly developed tools from the synchronous model to the bounded-delay bounded-drift model in a way that keeps them in working condition.

The key to efficiency is a recursive approach, where each node participates in only $\lceil \log f \rceil$ consensus instances, one for each level of recursion. On each level, the overhead of the reduction over a call to the consensus routine is a constant multiplicative factor both in time and bit complexity; concretely, this means that both complexities increase by overall factors of $O(\log f)$. Applying suitable consensus routines yields *exponential improvements* in bit complexity of deterministic and time complexity of randomised solutions, respectively:

1. In the deterministic setting, we exponentially reduce the number of bits each node broadcasts per time unit to $\Theta(\log f)$, while retaining $\Theta(f)$ stabilisation time. This is achieved by employing the phase king algorithm [4] in our construction.
2. In the randomised setting, we exponentially reduce the stabilisation time to $\operatorname{polylog} f$, where each node broadcasts $\operatorname{polylog} f$ bits per time unit. This is achieved using the algorithm by King and Saia [21]. We note that this slightly reduces resilience to $f < n/(3+\varepsilon)$ for any fixed constant $\varepsilon > 0$ and requires private communication channels.
3. In the randomised setting, we can also obtain a stabilisation time of $O(\log f)$, polynomial communication complexity, and optimal resilience of $f < n/3$ by assuming private communication channels. This is achieved using the consensus routine of Feldman and Micali [18]. This almost settles the open question by Ben-Or et al. [3] whether pulse synchronisation can be solved in expected constant time.

The running times of the randomised algorithms (2) and (3) hold with high probability and the additional assumptions on resilience and private communication channels are inherited from the employed consensus routines. Here, private communication channels mean that Byzantine nodes must make their decision on which messages to sent in round r based on knowledge of the algorithm, inputs, and all messages faulty nodes receive up to and including round r. The probability distribution is then over the independent internal randomness of the correct nodes (which the adversary can only observe indirectly) and any possible randomness of the adversary. Our framework does not impose these additional assumptions: stabilisation is guaranteed for $f < n/3$ on each recursive level of our framework as soon as the underlying consensus routine succeeds (within prespecified time bounds) constantly many times in a row. Our results and prior work are summarised in Table 1.

Regardless of the employed consensus routine, we achieve a skew of $2d$, where d is the maximum message delay. This is optimal in our model, but overly pessimistic if the sum of communication and computation delay is not between 0 and d, but from (d^-, d^+), where $d^+ - d^- \ll d^+$. In terms of d^+ and d^-, a skew of $\Theta(d^+ - d^-)$ is asymptotically optimal [29, 34]. We remark that in [20], it is shown how to combine the algorithms from [9] and [34] to achieve this bound without affecting the other properties shown in [9]; we are confident that the same technique can be applied to the algorithm proposed in this work. Finally, all our algorithms work with any clock drift parameter $1 < \vartheta \leq 1.007$, that is, the nodes' clocks can have up to 0.7% drift. In comparison, cheap quartz oscillators achieve $\vartheta \approx 1 + 10^{-5}$.

We consider our results of interest beyond the immediate improvements in complexity of the best known algorithms for pulse synchronisation. Since our framework may employ any consensus algorithm, it proves that pulse synchronisation is, essentially, *as easy* as synchronous consensus – a problem without the requirement for self-stabilisation or any timing uncertainty. Apart from the possibility for future improvements in consensus algorithms carrying over, this accentuates the following open question:

Is pulse synchronisation *at least as hard* as synchronous consensus?

Due to the various lower bounds and impossibility results on consensus [30, 19, 8, 14] mentioned earlier, a positive answer would immediately imply that the presented techniques are near-optimal. However, one may speculate that pulse synchronisation may rather have

the character of (synchronous) approximate agreement [13, 17], as *precise* synchronisation of the pulse events at different nodes is not required. Considering that approximate agreement can be deterministically solved in $O(\log n)$ rounds, a negative answer is a clear possibility as well. Given that all currently known solutions either explicitly solve consensus, leverage techniques that are likely to be strong enough to solve consensus, or are very slow, this would suggest that new algorithmic techniques and insights into the problem are necessary.

2 Preliminaries

Let V denote the set of all n nodes, $F \subseteq V$ be the set of faulty nodes such that $|F| < n/3$, and $G = V \setminus F$ the set of correct nodes. The sets G and F are unknown to the correct nodes in the system. We assume a continous reference time $[0, \infty)$ that is *not* available to the nodes in the distributed system. The reference time is only used to reason about the behaviour of the system. The adversary can choose the initial state of the system (memory contents, initial clock values, any messages in transit), the set F of faulty nodes which it controls, how the correct nodes' clocks progress and what is the delay of each individual message within the respective maximum clock drift and message delay bounds of ϑ and d. We assume that ϑ and d are known constants. For the full formal description of the model we refer to [27].

Pulse synchronisation algorithms. In the pulse synchronisation problem, the task is to have all the correct nodes locally generate pulse events in an almost synchronised fashion, despite arbitrary initial states and the presence of Byzantine faulty nodes. In addition, these pulses have to be well-separated. Let $p(v,t) \in \{0,1\}$ indicate whether a correct node $v \in G$ generates a pulse at time t. Moreover, let $p_k(v,t) \in [t, \infty)$ denote the time when node v generates the kth pulse event at or after time t and $p_k(v,t) = \infty$ if no such time exists. We say that the system has stabilised from time t onwards if

1. $p_1(v,t) \leq t + \Phi^+$ for all $v \in G$,
2. $|p_k(v,t) - p_k(u,t)| < \sigma$ for all $u, v \in G$ and $k \geq 1$,
3. $\Phi^- \leq p_{k+1}(v,t) - \min\{p_k(u,t) : u \in G\} \leq \Phi^+$ for all $v \in G$ and $k \geq 1$,

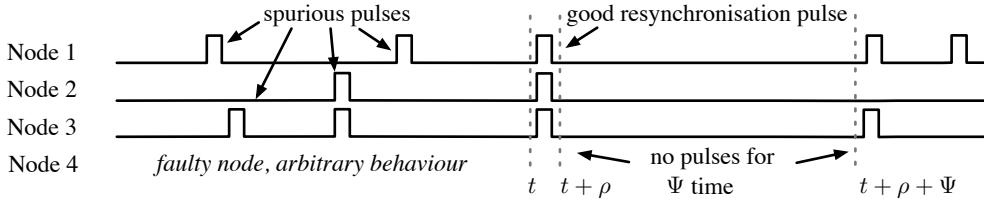
where Φ^- and Φ^+ are the accuracy bounds controlling the separation of the generated pulses. That is, (1) all correct nodes generate a pulse during the interval $[t, t + \Phi^+]$, (2) the kth pulse of any two correct nodes is less than σ time apart, and (3) for any pair of correct nodes their subsequent pulses are at least Φ^- but at most Φ^+ time apart.

We say that $\mathbf{A}$ is an f-resilient pulse synchronisation algorithm with *skew* σ and *accuracy* $\Phi = (\Phi^-, \Phi^+)$ with stabilisation time $T(\mathbf{A})$, if for any choices of the adversary such that $|F| \leq f$, there exists a time $t \leq T(\mathbf{A})$ such that the system stabilises from time t onwards. Moreover, a pulse synchronisation algorithm $\mathbf{A}$ is said to be a *T-pulser* if the accuracy bounds satisfy $\Phi^-, \Phi^+ \in \Theta(T)$. We use $M(\mathbf{A})$ to denote the maximum number of bits a correct node communicates per unit time when executing $\mathbf{A}$.

Resynchronisation algorithms. In our pulse synchronisation algorithm, we use so-called resynchronisation pulses to facilitate stabilisation. Essentially, the resynchronisation pulses are given by a weak variant of a pulse synchronisation algorithm, where the guarantee is that at some point all correct nodes generate a pulse almost synchronously, which is followed by a long period of silence. At all other times, the behaviour can be arbitrary.

Formally, we say that $\mathbf{B}$ is an f-resilient resynchronisation algorithm with skew ρ and separation window Ψ that stabilises in time $T(\mathbf{B})$ if the following holds: for any choices of the adversary such that $|F| \leq f$, there exists a time $t \leq T(\mathbf{B})$ such that every correct

node $v \in G$ locally generates a *resynchronisation pulse* at time $r(v) \in [t, t+\rho)$ and no other resynchronisation pulse before time $t+\rho+\Psi$. We call such a resynchronisation pulse *good*. In particular, we do not impose any restrictions on what the nodes do outside the interval $[t, t+\rho+\Psi)$, that is, there may be *spurious* resynchronisation pulses outside this interval:



3 The transformation framework

Our main contribution is a modular framework that allows us to turn any *non-self-stabilising* synchronous consensus algorithm into a self-stabilising pulse synchronisation algorithm in the bounded-delay model. In particular, this construction yields only a small overhead in time and communication complexity. This shows that efficient synchronous consensus algorithms imply efficient pulse synchronisation algorithms. As our construction is relatively involved, we opt to present it in a top-down fashion.

The main result. For notational convenience, we say that $\mathcal{C}$ is a *family of synchronous consensus routines* with running time $R(f)$ and message size $M(f)$, if for any $f \geq 0$ and $n \geq n(f)$, there exists a synchronous consensus algorithm $\mathbf{C} \in \mathcal{C}$ that runs correctly on n nodes given that there are at most f faulty nodes, terminates in $R(f)$ rounds, and uses messages of size $M(f)$. Here $n(f)$ gives the minimum number of nodes needed as a function of the resilience parameter f. Note that $R(f)$, $M(f)$, and $n(f)$ depend on $\mathcal{C}$; however, making this explicit would clutter notation. We emphasise that the algorithms in $\mathcal{C}$ are not assumed to be self-stabilising. Our main technical result states that given a family of consensus routines, we can obtain pulse synchronisation algorithms with only small additional overhead.

▶ **Theorem 1.** *Let $\mathcal{C}$ be a family of synchronous consensus routines that satisfy (i) for any $f_0, f_1 \in \mathbb{N}$, $n(f_0+f_1) \leq n(f_0)+n(f_1)$ and (ii) both $M(f)$ and $R(f)$ are increasing. Then, for any $f \geq 0$, $n \geq n(f)$, and $1 < \vartheta \leq 1.007$, there exists a $T_0(f) \in \Theta(R(f))$, such that for any $T \geq T_0(f)$ we can construct a T-pulser $\mathbf{A}$ with skew $2d$. The stabilisation time $T(\mathbf{A})$ and number of bits $M(\mathbf{A})$ broadcasted per time unit satisfy*

$$T(\mathbf{A}) \in O\left(d + \sum_{k=0}^{\lceil \log f \rceil} R(2^k)\right) \quad \text{and} \quad M(\mathbf{A}) \in O\left(1 + \sum_{k=0}^{\lceil \log f \rceil} M(2^k)\right),$$

where the sums are empty when $f = 0$.

In the deterministic case, the *phase king algorithm* [5] provides a family of synchronous consensus routines that satisfy the requirements. Moreover, it achieves optimal resilience (i.e., the minimal possible $n(f) = 3f+1$ [30]), constant message size, and asymptotically optimal [19] running time $R(f) \in O(f)$. Thus, this immediately yields the following result.

▶ **Corollary 2.** *For any $f \geq 0$ and $n > 3f$, there exists a deterministic f-resilient pulse synchronisation algorithm over n nodes with skew $2d$ and accuracy bounds $\Phi^-, \Phi^+ \in \Theta(f)$ that stabilises in $O(f)$ time and has correct nodes broadcast $O(\log f)$ bits per time unit.*

Randomised algorithms. Extending Theorem 1 for use with randomised consensus routines is straightforward; the reader is referred to the full paper [27] for details. By applying our construction to a fast and communication-efficient randomised consensus algorithm, e.g. the one by King and Saia [21], we get an efficient randomised pulse synchronisation algorithm.

▶ **Corollary 3.** *Suppose we have private channels. For any $f \geq 0$, constant $\varepsilon > 0$, and $n > (3+\varepsilon)f$, there exists a randomised f-resilient $\Theta(\text{polylog}\, f)$-pulser over n nodes that stabilises in* $\text{polylog}\, f$ *time w.h.p. and has nodes broadcast* $\text{polylog}\, f$ *bits per time unit.*

We can also utilise the constant expected time protocol by Feldman and Micali [18]. With some care, we can show that for $R(f) \in O(1)$, Chernoff's bound readily implies that the stabilisation time is not only in $O(\log n)$ in expectation, but also with high probability.

▶ **Corollary 4.** *Suppose we have private channels. For any $f \geq 0$ and $n > 3f$, there exists a randomised f-resilient $\Theta(\log f)$-pulser over n nodes that stabilises in $O(\log f)$ time w.h.p. and has nodes broadcast* $\text{poly}\, f$ *bits per time unit.*

Proof sketch for Theorem 1. The proof of the main result takes an inductive approach. In the inductive step, we assume two pulse synchronisation algorithms with small resilience. We then use these to construct (via some hoops we discuss later) a new pulse synchronisation algorithm with higher resilience. This step is formalised in the following lemma.

▶ **Lemma 5.** *Let $f, n_0, n_1 \in \mathbb{N}$, $n = n_0 + n_1$, $f_0 = \lfloor (f-1)/2 \rfloor$, and $f_1 = \lceil (f-1)/2 \rceil$. Suppose for $i \in \{0,1\}$ there exists an f_i-resilient $\Theta(R)$-pulser $\mathbf{A}_i$ that runs on n_i nodes and whose accuracy bounds Φ_h^- and Φ_h^+ satisfy $\Phi_h^+ = \varphi\Phi_h^-$ for sufficiently small constants $\varphi > \vartheta$. Let $\mathbf{C}$ be an f-resilient consensus algorithm for a network of n nodes that has running time R and uses messages of at most M bits. Then there exists a $\Theta(R)$-pulser $\mathbf{A}$ that*

- *runs on n nodes and has resilience f,*
- *stabilises in time $T(\mathbf{A}) \in \max\{T(\mathbf{A}_0), T(\mathbf{A}_1)\} + O(R)$,*
- *has nodes broadcast $M(\mathbf{A}) \in \max\{M(\mathbf{A}_0), M(\mathbf{A}_1)\} + O(M)$ bits per time unit, and*
- *has skew $2d$ and whose accuracy bounds Φ^- and Φ^+ satisfy that $\Phi^+ = \varphi\Phi^-$.*

Given the above lemma, it is relatively straightforward to show Theorem 1. Essentially, we can prove the claim for $f \in \bigcup_{k\geq 0}[2^k, 2^{k+1})$ using induction on k. As the base case, we use $f = 0$, that is, pulse synchronisation algorithms that tolerate *no* faulty nodes. These are trivial to obtain for any n: we pick a single node as a leader that generates a pulse when $\Phi^+ - \vartheta d$ time has passed on its local clock. Whenever the leader node pulses, all other nodes observe this within d time units. We have all other nodes generate a pulse whenever they observe the leader node generating a pulse. Thus, for $f = 0$ we have algorithms that stabilise in $O(d)$ time, broadcast $O(1)$ bits in $O(d)$ time, and have accuracy bounds such that $\Phi^- = \Phi^+/\vartheta - d$. For the inductive step, we can assume that f'-resilient pulse synchronisation algorithms exist for all $f' < 2^k$ and $n' \geq n(f')$ and apply Lemma 5.

The auxiliary results. In order to show Lemma 5, we use two main ingredients: (1) a pulse synchronisation algorithm whose stabilisation mechanism is triggered by a resynchronisation pulse and (2) a resynchronisation algorithm providing the latter. These ingredients are formalised in the following two theorems.

▶ **Theorem 6.** *Let $f \geq 0$, $n > 3f$ and $(1+\sqrt{5})/3 > \vartheta > 1$. Suppose for a network of n nodes there exist*

- *an f-resilient synchronous consensus algorithm $\mathbf{C}$, and*
- *an f-resilient resynchronisation algorithm $\mathbf{B}$ with skew $\rho \in O(d)$ and sufficiently large separation window $\Psi \in O(R)$ that tolerates clock drift of ϑ,*

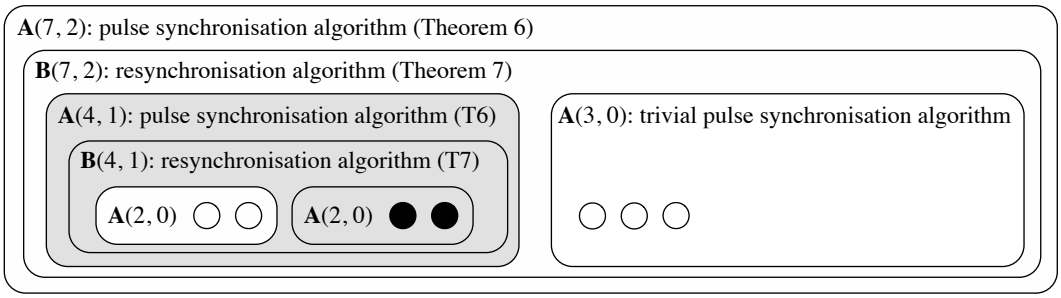


Figure 1 Recursively building a 2-resilient pulse synchronisation algorithm $\mathbf{A}(7,2)$ over 7 nodes. The construction utilises low resilience pulse synchronisation algorithms to build high resilience resynchronisation algorithms which can then be used to obtain highly resilient pulse synchronisation algorithms. Here, the base case consists of trivial 0-resilient pulse synchronisation algorithms $\mathbf{A}(2,0)$ and $\mathbf{A}(3,0)$ over 2 and 3 nodes, respectively. Two copies of $\mathbf{A}(2,0)$ are used to build a 1-resilient resynchronisation algorithm $\mathbf{B}(4,1)$ over 4 nodes using Theorem 7. The resynchronisation algorithm $\mathbf{B}(4,1)$ is used to obtain a pulse synchronisation algorithm $\mathbf{A}(4,1)$ via Theorem 6. Now, the 1-resilient pulse synchronisation algorithm $\mathbf{A}(4,1)$ over 4 nodes is used together with the trivial 0-resilient algorithm $\mathbf{A}(3,0)$ to obtain a 2-resilient resynchronisation algorithm $\mathbf{B}(7,2)$ for 7 nodes and the resulting pulse synchronisation algorithm $\mathbf{A}(7,2)$. White nodes represent correct nodes and black nodes represent faulty nodes. The gray blocks contain too many faulty nodes for the respective algorithms to correctly operate, and hence, they may have arbitrary output.

where $\mathbf{C}$ *runs in* $R = R(f)$ *rounds and lets nodes send at most* $M = M(f)$ *bits per round. Then a* $\varphi_0(\vartheta) \in 1 + O(\vartheta - 1)$ *exists so that for any constant* $\varphi > \varphi_0(\vartheta)$ *and sufficiently large* $T \in O(R)$*, there exists an* f*-resilient pulse synchronisation algorithm* $\mathbf{A}$ *for* n *nodes that*

- *has skew* $\sigma = 2d$ *and satisfies the accuracy bounds* $\Phi^- = T$ *and* $\Phi^+ = T\varphi$*,*
- *stabilises in* $T(\mathbf{B}) + O(R)$ *time and has nodes broadcast* $M(\mathbf{B}) + O(M)$ *bits per time unit.*

To apply the above theorem, we require suitable consensus and resynchronisation algorithms. We rely on consensus algorithms from prior work and construct efficient resynchronisation algorithms ourselves. The idea is to combine pulse synchronisation algorithms that have *low resilience* to obtain resynchronisation algorithms with *high resilience*.

▶ **Theorem 7.** *Let* $f, n_0, n_1 \in \mathbb{N}$*,* $n = n_0 + n_1$*,* $f_0 = \lfloor (f-1)/2 \rfloor$*,* $f_1 = \lceil (f-1)/2 \rceil$*, and* $1 < \vartheta \leq 1.007$*. Suppose that for some given* $\Psi \in \Omega(1)$*, sufficiently small constant* $\varphi > \varphi_0(\vartheta)$*, and* $T_0 \in \Theta(\Psi)$*, it holds that for any* $h \in \{0,1\}$ *and* $T_0 \leq T \in O(\Psi)$ *there exists a pulse synchronisation algorithm* $\mathbf{A}_h$ *that*

- *runs on* n_h *nodes and tolerates* f_h *faulty nodes,*
- *has skew* $\sigma = 2d$ *and accuracy bounds* $\Phi_h^- = T$ *and* $\Phi_h^+ = T\varphi$*.*

Then there exists a resynchronisation algorithm $\mathbf{B}$ *with skew* $\rho \in O(d)$ *and separation window of length* Ψ *that generates a resynchronisation pulse by time* $\max\{T(\mathbf{A}_0), T(\mathbf{A}_1)\} + O(\Psi)$*, where nodes broadcast only* $O(1)$ *additional bits per time unit.*

Given a suitable consensus algorithm, one can readily combine Theorems 6 and 7 to obtain Lemma 5. Therefore, we can reduce the problem of constructing an f-resilient pulse synchronisation algorithm to finding algorithms that tolerate up to $\lfloor f/2 \rfloor$ faults and recurse; see Figure 1 for an example on how the two types of algorithms are interleaved.

In the remainder of this paper, we overview the main ideas behind the above two theorems. As the proofs are relatively involved due to a large number of technicalities arising from the

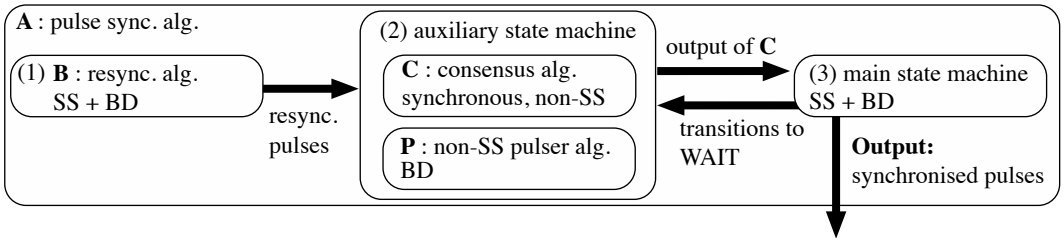


Figure 2 Constructing a self-stabilising (SS) and Byzantine fault-tolerant (BD) pulse synchronisation algorithm **A** out of a Byzantine fault-tolerant but non-stabilising pulse synchronisation algorithm **P**, synchronous consensus algorithm **C**, and resynchronisation algorithm **B**. All algorithms run on the same node set. (1) The resynchronisation algorithm **B** eventually outputs a good resynchronisation pulse, which resets the stabilisation mechanism used by the auxiliary state machine. (2) The auxiliary state machine simulates the executions of **C** using **P**. Simulations are initiated either due to nodes transitioning to a special WAIT state of the main state machine (see Figure 3) or a certain time after a resynchronisation pulse. (3) The main state machine. It generates pulses when a consensus instance outputs "1" and, when stabilised, guarantees re-initialisation of the consensus algorithm by the auxiliary state machine.

uncertainties introduced by the clock drift and message delay, we focus on summarising the key ideas and deliberately skip over a number of details and avoid formalising the claims. All the missing details and full proofs are given in the full paper [27].

4 The self-stabilising pulse synchronisation algorithm (Theorem 6)

We now overview the key elements in the construction of Theorem 6 illustrated in Figure 2:

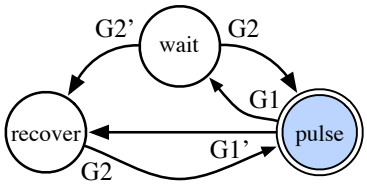
- a non-self-stabilising pulse synchronisation algorithm **P**,
- a synchronous, non-self-stabilising consensus routine **C**,
- a self-stabilising resynchronisation algorithm **B**, and
- the constructed pulse synchronisation algorithm **A**.

Non-self-stabilising pulse synchronisation. The first component we need is a non-self-stabilising pulse synchronisation algorithm **P** that tolerates Byzantine faults. To this end, we use a variant of the classic clock synchronisation algorithm by Srikanth and Toeug [33] that avoids transmitting clock values in favour of unlabelled pulses. As we do not require self-stabilisation for now, we can assume that all nodes receive an *initialisation signal* during the time window $[0, \tau)$ for a given parameter τ. The following theorem summarises the properties of the algorithm.

▶ **Theorem 8.** *Let $n > 1$, $f < n/3$, and $\tau > 0$. If every correct node receives an initialisation signal during $[0, \tau)$, then there exists a pulse synchronisation algorithm* **P** *such that:*

- *all correct nodes generate the first pulse (after initialisation) within time $O(\vartheta^2 d\tau)$,*
- *the pulses have skew $2d$,*
- *the accuracy bounds are $\Phi^- \in \Omega(\vartheta d)$ and $\Phi^+ \in O(\vartheta^2 d)$, and*
- *the algorithm communicates at most one bit per time unit.*

We can simulate synchronous message-passing algorithms with the above algorithm as follows. Assuming that no transient failures or new initialisation signals occur after time τ, by time $O(\vartheta^2 d\tau)$ the algorithm starts to generate pulses with skew $2d$ and accuracy bounds $\Phi^- \in \Omega(\vartheta d)$ and $\Phi^+ \in O(\vartheta^2 d)$. We can set the $\Omega(\vartheta d)$ term to be large enough so that all



Transition guards:

G1 ≡ T_1 expires and saw $\geq n - f$ 'pulse' messages within time T_1 at some point before the timeout expires

G1'≡ T_1 expires and G1 is not satisfied

G2 ≡ auxiliary machine signals 'output 1'

G2'≡ T_{wait} expires or auxiliary machine signals 'output 0'

Figure 3 The main state machine. When a node transitions to state PULSE, it generates a pulse event and sends a PULSE message to all nodes. When the node transitions to state WAIT, it broadcasts a WAIT message to all nodes. Guard G1 employs a sliding window memory buffer, which stores any PULSE messages that have arrived within time T_1. When a correct node transitions to PULSE, it resets a local timer of length T_1. Once it expires, either Guard G1 or Guard G1' become satisfied. Similarly, the timer T_{wait} is reset when a node transitions to WAIT. Once it expires, Guard G2' is satisfied and the node transitions from WAIT to RECOVER. The node transitions to state PULSE when Guard G2 is satisfied, which requires an "output 1" signal from the auxiliary state machine.

correct nodes can complete local computations and send/receive messages for each simulated round $i - 1$ before the ith pulse occurs. Thus, nodes can associate each message with a distinct round i (by counting locally) and simulate synchronous message-passing algorithms.

The self-stabilising algorithm. The general idea is to repeatedly simulate $\mathbf{C}$ to agree on the time of the next pulse. However, we must deal with an arbitrary initial system state. In particular, the correct nodes may be scattered over the states, with inconsistent memory content, and also the timers employed in the transition guards may have arbitrary values (within their domains). Nonetheless, assume for the moment that there is a small window of length $\rho \in O(d)$ during which each node receives a *resynchronisation pulse*, which triggers the initialisation of the stabilisation mechanism.

The construction relies on two components: (1) a main state machine given in Figure 3 and (2) an auxiliary state machine that acts as a wrapper for an arbitrary consensus algorithm. The main state machine is responsible for generating pulses, whereas the auxiliary state machine generates signals that drive the main state machine. The main machine works as follows: whenever a node enters the PULSE state, it waits for some time to see if at least $n - f$ nodes generated a pulse within a short time window. If not, the system has not stabilised, and the node goes into the RECOVER state to indicate this. Otherwise, the node goes into the WAIT state, where it remains for long enough to (a) separate any subsequent pulses from previous ones and (b) receive the next signal from the auxiliary machine. Once stabilised, the auxiliary machine is guaranteed to send the signal "1" within bounded time. This indicates that the node should pulse again. If no signal arrives on time or the signal is "0", this means that the system has not stabilised and the node goes into the RECOVER state.

While the auxiliary state machine is slightly more involved, the basic idea is simple: (a) nodes try to check whether at least $n - f$ nodes transition to the WAIT state in the main state machine *in a short enough time window* (that is, a time window that would suffice during correct operation) and (b) then use a consensus routine to agree on this observation. Assuming that all correct nodes participate in the simulation of the consensus routine, we get the following:

- If the consensus algorithm $\mathbf{C}$ outputs "0", then some $v \in G$ did not see $n - f$ nodes transitioning to WAIT in a short time window, and hence, the system has not yet stabilised.
- If the consensus algorithm $\mathbf{C}$ outputs "1", then every $v \in G$ agrees that a transition to WAIT happened recently.

In particular, the idea is that when the system operates correctly, the consensus simulation will always succeed and output "1" at every correct node.

The obvious problem here is that the consensus routine is not self-stabilising and it operates in a synchronous model of computation. To remedy the latter problem, we use the algorithm from Theorem 8 to simulate round-based execution. However, this requires that an initialisation signal is generated within a time window of length τ, thus requiring some level of synchrony among the correct nodes. To wiggle our way out of this issue, we carefully construct the main state machine and auxiliary machine to satisfy the following properties:

1. The main state machine guarantees that if *some* correct node transitions to WAIT, then after a short interval no correct node transitions to WAIT for an extended period of time.
2. If a node $u \in G$ sees at least $n - f$ nodes transitioning to WAIT in a short time window (including itself), then the node attempts to start a consensus instance with input "1".
3. If node $u \in G$ attempts to start a simulation of consensus with input "1", then at least $n - 2f > f$ correct nodes $v \in G$ must have recently transitioned to WAIT. As all nodes can reliably detect this event, this essentially ensures that their auxilliary machines synchronise. This way, we can guarantee that all correct nodes initialise a new consensus instance within τ time of each other and generate a consistent output.
4. If this output is "1", all correct nodes generate a synchronised pulse and the system stabilises. Otherwise, all of them transition to state RECOVER.
5. If no $u \in G$ attempts to start a simulation of consensus with input "1" within a certain time, we make sure that all correct nodes end up in RECOVER. Here, we exploit that any consensus instance can be made *silent* [26], which means that no messages are sent by correct nodes if they all have input "0". Hence, even if not all correct nodes actually participate in an instance, it does not matter as long as no correct node has input "1".

Thus, either the system stabilises within a certain time or all correct nodes end up in state RECOVER. This is where we utilise the resynchronisation signals: when a resynchronisation signal is received, the nodes reset a local timer. Since the resynchronisation signal has a small skew of $\rho \in O(d)$, these timers expire within a relatively small time window as well. If the timer expires when all correct nodes are in the RECOVER state, then they can explicitly restart the system in synchrony, also resulting in stabilisation. The key here is to get a good resynchronisation pulse at some point, so that no spurious resynchronisation pulses interfere with the described stabilisation mechanism until it is complete. Once succesful, no correct nodes transition to RECOVER anymore. Thus, any subsequent resynchronisation pulses do not affect pulse generation. For a detailed discussion and formal analysis, see [27].

5 Generating resynchronisation pulses (Theorem 7)

The final ingredient is a mechanism to generate resynchronisation pulses; see Figure 4 for the general structure of the construction. Recall that a *good* resynchronisation pulse is an event triggered at all correct nodes within a small time interval, followed by at least Ψ time during which no correct node triggers a new such event. In order to construct an algorithm that generates such an event, we partition the set of n nodes into two disjoint *blocks* of roughly $n/2$ nodes. Each block runs an instance of a pulse synchronisation algorithm tolerating f_i faults, where $f_0 + f_1 + 1 = f$ (and $f_0 \approx f_1 \approx f/2$). For these two algorithms, we choose different pulsing frequencies (that is, accuracy bounds) that are roughly coprime integer multiples of the desired separation window Ψ. Both algorithms are used as potential sources of resynchronisation pulses. The idea behind our construction is illustrated in Figure 5. If both instances stabilise, it is not difficult to set up the frequencies such that $\mathbf{A}_i$ eventually generates a pulse that is not followed by a pulse from $\mathbf{A}_{1-i}$ within time Ψ.

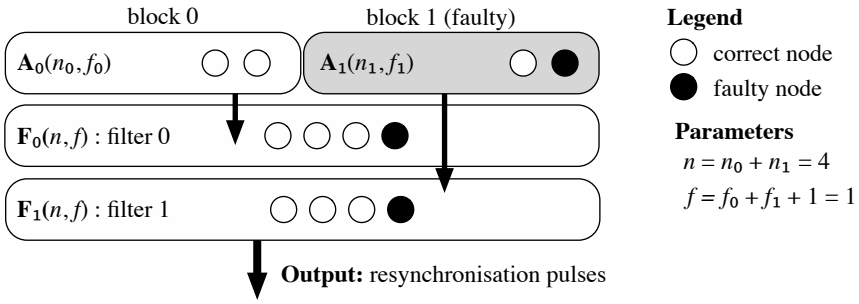


Figure 4 Construction of an f-resilient resynchronisation algorithm on n nodes from f_i-resilient pulse synchronisation algorithms on n_i nodes, where $f = f_0 + f_1 + 1$ and $n = n_0 + n_1$. The n nodes are divided into two groups of n_0 and n_1 nodes. These groups run pulse synchronisation algorithms $\mathbf{A}_0$ and $\mathbf{A}_1$, respectively. At least one of these algorithms is guaranteed to stabilise eventually. Here, $\mathbf{A}_1$ (gray block) has too many faulty nodes and does not stabilise. All of the n nodes together run two filtering mechanisms $\mathbf{F}_0$ and $\mathbf{F}_1$ for the outputs of $\mathbf{A}_0$ and $\mathbf{A}_1$, respectively. These ensure that no correct node locally generates a resynchronisation pulse without all correct nodes registering this event, and then apply timeout constraints to enforce the desired frequency bounds.

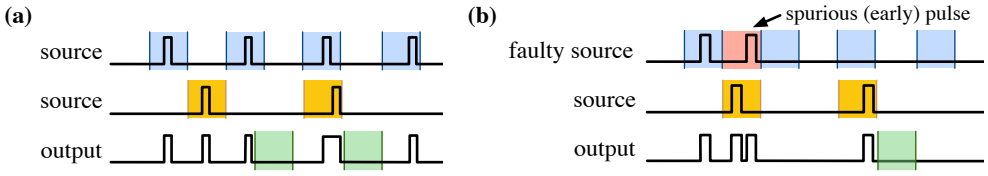


Figure 5 Idea of the resynchronisation algorithm. We take two pulse sources with (up to scaling) coprime frequencies and output the logical OR of the two sources. In this example, the pulses of the first source should occur in the blue regions, whereas the pulses of the second source should hit the yellow regions. The green regions indicate a period where a pulse from either source is followed by at least Ψ time of silence. Eventually, such a region appears. (a) Two correct sources that pulse with set frequencies. (b) One faulty source that produces spurious pulses. Here, a pulse occurs too early (red region), and thus, we then enforce that the faulty source is silenced for $\Theta(\Psi)$ time.

Unfortunately, one of the instances (but not both) could have more than f_i faulty nodes, never stabilise, and thus generate possibly inconsistent pulses at arbitrary points in time. We overcome this by a two-step filtering process illustrated in Figure 6. First, we apply a number of threshold votes ensuring that if a pulse of a block is considered as a candidate resynchronisation pulse by *some* correct node, then *all* correct nodes observe this event. Second, we locally *filter out* any observed events that do not obey the prescribed frequency bounds for the respective block. Thus, the faulty block either generates (possibly inconsistent) pulses within the prescribed frequency bounds only, or its influence is suppressed entirely (for sufficiently long time). Either way, the correctly operating block will eventually succeed in generating a resynchronisation pulse. Further details and all missing proofs appear in the full version of this paper [27].

Acknowledgements. We are grateful to Danny Dolev for numerous discussions on the pulse synchronisation problem and detailed comments on early drafts of this paper. We also wish to thank Borzoo Bonakdarpour, Janne H. Korhonen, Christian Scheideler, Jukka Suomela, and anonymous reviewers for their helpful comments.

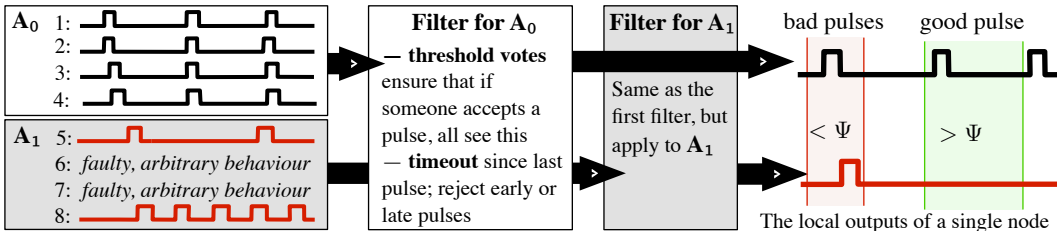


Figure 6 Example of the resynchronisation construction for 8 nodes tolerating 2 faults. We partition the network into two parts, each running a pulse synchronisation algorithm $\mathbf{A}_i$. The output of $\mathbf{A}_i$ is fed into the respective filter and any pulse that passes the filtering is used as a resynchronisation pulse. The filtering consists of (1) having *all* nodes in the network participate in a threshold vote to see if anyone thinks a pulse from $\mathbf{A}_i$ occurred (i.e. enough nodes running $\mathbf{A}_i$ generated a pulse) and (2) keeping track when was the last time a pulse from $\mathbf{A}_i$ occurred to check that the accuracy bounds of $\mathbf{A}_i$ are respected: pulses that appear too early or too late are ignored.

References

1 M. K. Aguilera and S. Toueg. Simple bivalency proof that t-resilient consensus requires $t+1$ rounds. *Information Processing Letters*, 71(3):155–158, 1999.

2 M. Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *PODC 1983*, pages 27–30, 1983.

3 M. Ben-Or, D. Dolev, and E. N. Hoch. Fast self-stabilizing Byzantine tolerant digital clock synchronization. In *PODC 2008*, pages 385–394, 2008.

4 P. Berman, J. A. Garay, and K. J. Perry. Towards optimal distributed consensus. In *FOCS 1989*, pages 410–415, 1989.

5 P. Berman, J. A. Garay, and K. J. Perry. Bit optimal distributed consensus. In *Computer Science: Research and Applications*, pages 313–321, 1992.

6 A. Daliot, D. Dolev, and H. Parnas. Self-stabilizing pulse synchronization inspired by biological pacemaker networks. In *SSS 2003*, pages 32–48, 2003.

7 Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

8 D. Dolev. The Byzantine generals strike again. *Journal of Algorithms*, 3(1):14–30, 1982.

9 D. Dolev, M. Függer, C. Lenzen, and U. Schmid. Fault-tolerant algorithms for tick-generation in asynchronous logic. *Journal of the ACM*, 61(5):30:1–30:74, 2014.

10 D. Dolev, J. Y. Halpern, and H. R. Strong. On the possibility and impossibility of achieving clock synchronization. *Journal of Computer and System Sciences*, 32(2):230–250, 1986.

11 D. Dolev, K. Heljanko, M. Järvisalo, J. H. Korhonen, C. Lenzen, J. Rybicki, J. Suomela, and S. Wieringa. Synchronous counting and computational algorithm design. *Journal of Computer and System Sciences*, 82(2):310–332, 2016.

12 D. Dolev and E. N Hoch. Byzantine self-stabilizing pulse in a bounded-delay model. In *SSS 2007*, pages 234–252, 2007.

13 D. Dolev, N. A. Lynch, S. S. Pinter, E. W. Stark, and W. E. Weihl. Reaching approximate agreement in the presence of faults. *Journal of the ACM*, 33(3):499–516, 1986.

14 D. Dolev and R. Reischuk. Bounds on information exchange for Byzantine agreement. *Journal of the ACM*, 32(1):191–204, 1985.

15 S. Dolev. *Self-Stabilization*. Cambridge, MA, 2000.

16 S. Dolev and J. L. Welch. Self-stabilizing clock synchronization in the presence of Byzantine faults. *Journal of the ACM*, 51(5):780–799, 2004.

17 A. D. Fekete. Asymptotically optimal algorithms for approximate agreement. *Distributed Computing*, 4(1):9–29, 1990.

18 P. Feldman and S. Micali. Optimal algorithms for Byzantine agreement. In *STOC 1988*, pages 148–161, 1988.

19 M. J. Fischer and N. A. Lynch. A lower bound for the time to assure interactive consistency. *Information Processing Letters*, 14(4):183–186, 1982.

20 P. Khanchandani and C. Lenzen. Self-stabilizing Byzantine clock synchronization with optimal precision. In *SSS 2016*, pages 213–230, 2016.

21 V. King and J. Saia. Breaking the $O(n^2)$ bit barrier. *Journal of the ACM*, 58(4):1–24, 2011.

22 L. Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(I):52–78, 1985.

23 L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

24 C. Lenzen, M. Függer, M. Hofstätter, and U. Schmid. Efficient construction of global time in SoCs despite arbitrary faults. In *DSD 2013*, pages 142–151, 2013.

25 C. Lenzen and J. Rybicki. Efficient counting with optimal resilience. In *DISC 2015*, pages 16–30, 2015.

26 C. Lenzen and J. Rybicki. Near-optimal self-stabilising counting and firing squads. In *SSS 2016*, pages 263–280, 2016.

27 C. Lenzen and J. Rybicki. Self-stabilising Byzantine clock synchronisation is almost as easy as consensus, 2017. Full version. URL: `http://arxiv.org/abs/1705.06173`.

28 C. Lenzen, J. Rybicki, and J. Suomela. Towards optimal synchronous counting. In *PODC 2015*, pages 441–450, 2015.

29 J. Lundelius and N. Lynch. An upper and lower bound for clock synchronization. *Information and Control*, 62(2–3):190–204, 1984.

30 M. C. Pease, R. E. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

31 M. O. Rabin. Randomized Byzantine generals. In *FOCS 1983*, pages 403–409, 1983.

32 M. Raynal. *Fault-tolerant agreement in synchronous message-passing systems*. Morgan & Claypool, 2010.

33 T. K. Srikanth and S. Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, 1987.

34 J. L. Welch and N. Lynch. A new fault tolerant algorithm for clock synchronization. *Information and Computation*, 77(1):1–36, 1988.

Neuro-RAM Unit with Applications to Similarity Testing and Compression in Spiking Neural Networks*

Nancy Lynch[1], Cameron Musco[2], and Merav Parter[3]

1 **CSAIL, MIT, Cambridge, USA**
`lynch@csail.mit.edu`
2 **CSAIL, MIT, Cambridge, USA**
`cnmusco@mit.edu`
3 **Weizmann Institute, Rehovot, Israel**
`merav.parter@weizmann.ac.il`

Abstract

We study distributed algorithms implemented in a simplified biologically inspired model for *stochastic spiking neural networks*. We focus on tradeoffs between computation time and network complexity, along with the role of noise and randomness in efficient neural computation.

It is widely accepted that neural spike responses, and neural computation in general, is inherently stochastic. In recent work, we explored how this stochasticity could be leveraged to solve the 'winner-take-all' leader election task. Here, we focus on using randomness in neural algorithms for similarity testing and compression. In the most basic setting, given two n-length patterns of firing neurons, we wish to distinguish if the patterns are equal or ϵ-far from equal.

Randomization allows us to solve this task with a very compact network, using $O\left(\frac{\sqrt{n}\log n}{\epsilon}\right)$ auxiliary neurons, which is sublinear in the input size. At the heart of our solution is the design of a t-round neural random access memory, or indexing network, which we call a *neuro-RAM*. This module can be implemented with $O(n/t)$ auxiliary neurons and is useful in many applications beyond similarity testing – e.g., we discuss its application to compression via random projection.

Using a VC dimension-based argument, we show that the tradeoff between runtime and network size in our neuro-RAM is near optimal. To the best of our knowledge, we are the first to apply these techniques to stochastic spiking networks. Our result has several implications – since our neuro-RAM can be implemented with deterministic threshold gates, it shows that, in contrast to similarity testing, randomness does not provide significant computational advantages for this problem. It also establishes a separation between feedforward networks whose gates spike with sigmoidal probabilities, and well-studied deterministic sigmoidal networks, whose gates output real number sigmoidal values, and which can implement a neuro-RAM much more efficiently.

1998 ACM Subject Classification F.1.1 Models of Computation – Self-modifying machines (e.g., neural networks), F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases spiking neural networks, biological distributed algorithms, circuit design

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.33

1 Introduction

Biological neural networks are arguably the most fascinating distributed computing systems in our world. However, while studied extensively in the fields of computational neuroscience

* Full version available at `https://arxiv.org/abs/1706.01382`


© Nancy Lynch, Cameron Musco, and Merav Parter;
licensed under Creative Commons License CC-BY
31st International Symposium on Distributed Computing (DISC 2017).
Editor: Andréa W. Richa; Article No. 33; pp. 33:1–33:16

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

and artificial intelligence, they have received little attention from a distributed computing perspective. Our goal is to study biological neural networks through the lens of distributed computing theory. We focus on understanding tradeoffs between computation time, network complexity, and the use of randomness in implementing basic algorithmic primitives, which can serve as building blocks for high level pattern recognition, learning, and processing tasks.

Spiking Neural Network (SNN) Model. We work with biologically inspired *spiking neural networks* (SNNs) [18, 19, 12, 15], in which neurons fire in discrete pulses in synchronous rounds, in response to a sufficiently high membrane potential. This potential is induced by spikes from neighboring neurons, which can have either an excitatory or inhibitory effect (increasing or decreasing the potential). As observed in biological networks, neurons are either strictly inhibitory (all outgoing edge weights are negative) or excitatory. As we will see, this restriction can significantly affect the power of these networks.

A key feature of our model is stochasticity – each neuron is a probabilistic threshold unit, spiking with probability given by applying a sigmoid function to its potential. While a rich literature focuses on deterministic circuits [21, 13] we employ a stochastic model as it is widely accepted that neural computation is stochastic [1, 24, 9].

Computational Problems in SNNs. We consider an n-bit binary input vector X, which represents the firing status of a set of input neurons. Given a (possibly multi-valued) function $f : \{0,1\}^n \to \{0,1\}^m$, we seek to design a network of spiking neurons that converges to an output vector $Z = f(X)$ (or any $Z \in f(X)$ if f is multi-valued) as quickly as possible using few auxiliary (non-input or output) neurons.

The number of auxiliary neurons used corresponds to the "node complexity" of the network [14]. Designing circuits with small node complexity has received a lot of attention – e.g., the work of [10] on PARITY and [3] on AC_0. Much less is known, however, on what is achievable in spiking neural networks. For most of the problems we study, there is a trivial solution that uses $\Theta(n)$ auxiliary neurons for inputs of size n. Hence, we primarily focus on designing *sublinear* size networks – with n^{1-c} auxiliary neurons for some c.

Past Work: WTA. Recently, we studied the 'winner-take-all' (WTA) leader election task in SNNs [17]. Given a set of firing input neurons, the network is required to converge to a single firing output – corresponding to the 'winning' input. In that work, we critically leveraged the noisy behavior of our spiking neuron model: randomness is key in breaking the symmetry between initially identical firing inputs.

This Paper: Similarity Testing and Compression. In this paper, we study the role of randomness in a different setting: for similarity testing and compression. Consider the basic similarity testing problem: given $X_1, X_2 \in \{0,1\}^n$, we wish to distinguish the case when $X_1 = X_2$ from the case when the Hamming distance between the vectors is large – i.e., $\mathrm{d_H}(X_1, X_2) \geq \epsilon n$ for some parameter ϵ. This problem can be solved very efficiently using randomness – it suffices to sample $O(\log n/\epsilon)$ indices and compare X_1 and X_2 at these positions to distinguish the two cases with high probability. Beyond similarity testing, similar compression approaches using random input subsampling or hashing can lead to very efficient routines for a number of data processing tasks.

1.1 A Neuro-RAM Unit

To implement the randomized similarity testing approach described above, and to serve as a foundation for other random compression methods in spiking networks, we design a basic *indexing module*, or random access memory, which we call a *neuro-RAM*. This module solves:

▶ **Definition 1** (Indexing). Given $X \in \{0,1\}^n$ and $Y \in \{0,1\}^{\log n}$ which is interpreted as an integer in $\{0, ..., n-1\}$, the indexing problem is to output the value of the Y^{th} bit of X[1].

Our neuro-RAM uses a sublinear number of auxiliary neurons and solves indexing with high probability on any input. We focus on characterizing the trade-off between the convergence time and network size of the neuro-RAM, giving nearly matching upper and lower bounds.

Generally, our results show that a compressed representation (e.g., the index Y) can be used to access a much larger datastore (e.g., X), using a very compact neural network. While binary indexing is not very 'neural' we can imagine similar ideas extending to more natural coding schemes used, for example, for memory retrieval, scent recognition, or other tasks.

Relation to Prior Work. Significant work has employed random synaptic connections between neurons – e.g., the Johnson-Lindenstrauss compression results of [2] and the work of Valiant [26]. While it is reasonable to assume that the initial synapses are random, biological mechanisms for changing connectivity (functional plasticity) act over relatively large time frames and cannot provide a new random sample of the network for each new input. In contrast, stochastic spiking neurons do provide fresh randomness to each computation. In general, transforming of a network with m possible random edges to a network with fixed edges and stochastic neurons requires $\Omega(m)$ auxiliary neurons and thus fails to fulfill our sublinearity goal, as there is typically at least one possible outgoing edge from each input. Our neuro-RAM can be thought of as improving the naive simulation – by reading a random entry of an input, we simulate a random edge from the specified neuron. Beyond similarity testing, we outline how our result can be used to implement Johnson-Lindenstrauss compression similar to [2] without assuming random connectivity.

1.2 Our Contributions

1.2.1 Efficient Neuro-RAM Unit

Our primary upper bound result is the following:

▶ **Theorem 2** (t-round Neuro-RAM). *For every integer $t \leq \sqrt{n}$, there is a (*recurrent*) SNN with $O(n/t)$ auxiliary neurons that solves the indexing problem in t rounds with high probability. In particular, there exists a neuro-RAM unit that contains $O(\sqrt{n})$ auxiliary neurons and solves the indexing problem in $O(\sqrt{n})$ rounds.*

Above, and throughout the paper 'with high probability' or w.h.p. to denotes with probability at least $1 - 1/n^c$ for some constant c. Theorem 2 is proven in Section 3.

Neuro-RAM Construction. The main idea is to first 'encode' the firing pattern of the input neurons X into the potentials of t neurons. These encoding neurons will spike with some probability dependent on their potential. However, simply recording the firing rates of the neurons to estimate this probability is too inefficient. Instead, we use a 'successive decoding

[1] Here, and throughout, for simplicity we assume n is a power of 2 so $\log n$ is an interger.

strategy', in which the firing rates of the encoding neurons are estimated at finer and finer levels of approximation, and adjusted through recurrent excitation or inhibition as decoding progresses. The strategy converges in $O(n/t)$ rounds – the smaller t is the more information is contained in the potential of a single neuron, and the longer decoding takes.

Theorem 2 shows a significant separation between our networks and traditional feedforward circuits where significantly sublinear sized indexing units are not possible.

▶ **Fact 3** (See Lower Bounds in [16]). *A circuit solving the indexing problem that consists of AND/OR gates connected in a feedforward manner requires $\Theta(n)$ gates. A feedforward circuit using linear threshold gates requires $\Theta(n/\log n)$ gates.*

We note, however, that our indexing mechanism *does not exploit the randomness of the spiking neurons*, and in fact can also be implemented with deterministic linear threshold gates. Thus, the separation between Theorem 2 and Fact 3 is entirely due to the recurrent (non-feedforward) layout of our network. Since any recurrent network using $O(m)$ neurons and converging in t rounds can be 'unrolled' into a feedforward circuit using $O(mt)$ neurons, Fact 3 shows that the tradeoff between network size and runtime in Theorem 2 is optimal up to a $\log n$ factor, if we use our spiking neurons in this restricted way. However, it does not rule out improvements using more sophisticated randomized strategies.

1.2.2 Lower Bound for Neuro-RAM in Spiking Networks

Surprisingly, we are able to show that despite the restricted way in which we use our spiking neuron model, significant improvements are not possible:

▶ **Theorem 4** (Lower Bound for Neuro-RAM in SNNs). *Any SNN that solves indexing in t rounds with high probability in our model must use at least $\Omega\left(\frac{n}{t\log^2 n}\right)$ auxiliary neurons.*

Theorem 4, whose proof is in Section 4, shows that the tradeoff in Theorem 2 is within a $\log^2 n$ factor of optimal. It matches the lower bound of Fact 3 for deterministic threshold gates up to a $\log n$ factor, showing that there is not a significant difference in the power of stochastic neurons and deterministic gates in solving indexing.

Reduction from SNNs to Deterministic Circuits. We first argue that the output distribution of any SNN is identical to the output distribution of an algorithm that first chooses a deterministic threshold circuit from some distribution and then applies it to the input. This is a powerful observation as it lets us apply Yao's principle: an SNN lower bound can be shown via a lower bound for deterministic circuits on any input distribution [27].

Deterministic Circuit Lower Bound via VC Dimension. We next show that any deterministic circuit that succeeds with high probability on uniform random inputs cannot be too small. The bound is via a VC dimension-based argument, which extends the work of [16]. As far as we are aware, we are the first to give a VC dimension-based lower bound for probabilistic and biologically plausible networks and we hope our work significantly expands the toolkit for proving lower bounds in this area. In contrast to our lower bounds on the WTA problem [17], which rely on indistinguishability arguments based on network structure, our new techniques allow us to give more general bounds for any network architecture.

Separation of Network Models. Aside from showing that randomness does not give significant advantages in constructing a neuro-RAM (contrasting with its importance in WTA and similarity testing), our proof of Theorem 4 establishes a separation between feedforward spiking networks and deterministic *sigmoidal circuits*. Our neurons spike with probability computed as a sigmoid of their membrane potential. In sigmoidal circuits, neurons output real numbers, equivalent to our spiking probabilities. A neuro-RAM can be implemented very efficiently in these networks:

▶ **Fact 5** (See [16], along with [19] for similar bounds). *There is a feedforward sigmoidal circuit solving the indexing problem using $O(\sqrt{n})$ gates.*[2]

In contrast, via an unrolling argument, the proof of Theorem 4 shows that any feedforward spiking network requires $\Omega\left(\frac{n}{\log^2 n}\right)$ gates to solve indexing with high probability.

It has been shown that feedforward sigmoidal circuits can significantly outperform standard feedforward linear threshold circuits [20, 16]. However, previously it was not known that restricting gates to spike with a sigmoid probability function rather than output the real value of this function significantly affected their power. Our lower bound, along with Fact 5, shows that in some cases it does. This separation highlights the importance of modeling spiking neuron behavior in understanding complexity tradeoffs in neural computation.

1.2.3 Applications to Randomized Similarity Testing and Compression

As discussed, our neuro-RAM is widely applicable to algorithms that require random sampling of inputs. In Section 5 we discuss our main application, to similarity testing – i.e., testing if $X_1 = X_2$ or if $\mathrm{d_H}(X_1, X_2) \geq \epsilon n$. It is easy to implement an exact equality tester using $\Theta(n)$ auxiliary neurons. Alternatively, one can solve exact equality with three auxiliary neurons using mixed positive and negative edge weights for the outgoing edges of inputs. However this is not biologically plausible – neurons typically have either all positive (excitatory) or all negative (inhibitory) outgoing edges, a restriction included in our model. Designing sublinear sized exact equality testers under this restriction seems difficult – simulating the three neuron solution requires at least $\Theta(n)$ auxiliary neurons – $\Theta(1)$ for each input.

By relaxing to similarity testing and applying our neuro-RAM, we can achieve sublinear sized networks. We can use $\Theta(\log n/\epsilon)$ neuro-RAMs, each with $O(\sqrt{n})$ auxiliary neurons to check equality at $\Theta(\log n/\epsilon)$ random positions of X_1 and X_2 distinguishing if $X_1 = X_2$ or if $\mathrm{d_H}(X_1, X_2) \geq \epsilon n$ with high probability. This is the first sublinear solution for this problem in the spiking neural networks. In Section 5, we discuss possible additional applications of our neuro-RAM to Johnson-Lindenstrauss random compression, which amounts to multiplying the input by a sparse random matrix – a generalization of input sampling.

2 Computational Model and Preliminaries

2.1 Network Structure

We now give a formal definition of our computational model. A *Spiking Neural Network* (SNN) $\mathcal{N} = \langle X, Z, A, w, b\rangle$ consists of n input neurons $X = \{x_1, \ldots, x_n\}$, m output neurons $Z = \{z_1, \ldots, z_m\}$, and ℓ auxiliary neurons $A = \{a_1, ..., a_\ell\}$. The directed, weighted synaptic

[2] Note that [20] shows that general deterministic sigmoidal circuits can be simulated by our spiking model. However, the simulation blows up the size of the circuit size by $\sqrt{n}$, giving $\Theta(n)$ auxiliary neurons.

connections between X, Z, and A are described by the weight function $w : [X \cup Z \cup A] \times [X \cup Z \cup A] \to \mathbb{R}$. A weight $w(u, v) = 0$ indicates that a connection is not present between neurons u and v. Finally, for any neuron v, $\mathrm{b}(v) \in \mathbb{R}_{\geq 0}$ is the activation bias – as we will see, roughly, v's membrane potential must reach $\mathrm{b}(v)$ for a spike to occur with good probability.

The weight function defining the synapses in our networks is restricted in a few notable ways. The in-degree of every input neuron x_i is zero. That is, $w(u, x) = 0$ for all $u \in [X \cup Z \cup A]$ and $x \in X$. This restriction bears in mind that the input layer might in fact be the output layer of another network and so incoming connections are avoided to allow for the composition of networks in higher level modular designs. Additionally, each neuron is either inhibitory or excitatory: if v is inhibitory, then $w(v, u) \leq 0$ for every u, and if v is excitatory, then $w(v, u) \geq 0$ for every u. All input and output neurons are excitatory.

2.2 Network Dynamics

An SNN evolves in discrete, synchronous rounds as a Markov chain. The firing probability of every neuron at time t depends on the firing status of its neighbors at time $t - 1$, via a standard sigmoid function, with details given below.

For each neuron u, and each time $t \geq 0$, let $u^t = 1$ if u fires (i.e., generates a spike) at time t. Let u^0 denote the initial firing state of the neuron. Our results will specify the initial input firing states $x_j^0 = 1$ and assume that $u^0 = 0$ for all $u \in [Z \cup A]$. For each non-input neuron u and every $t \geq 1$, let $pot(u, t)$ denote the membrane potential at round t and $p(u, t)$ denote the corresponding firing probability ($\Pr[u^t = 1]$). These values are calculated as:

$$pot(u,t) = \sum_{v \in X \cup Z \cup A} w_{v,u} \cdot v^{t-1} - b(u) \text{ and } p(u,t) = \frac{1}{1 + e^{-pot(u,t)/\lambda}} \tag{1}$$

where $\lambda > 0$ is a *temperature parameter*, which determines the steepness of the sigmoid. It is easy to see that λ does not affect the computational power of the network. A network can be made to work with any λ simply by scaling the synapse weights and biases appropriately.

For simplicity we assume that $\lambda = \frac{1}{\Theta(\log n)}$. Thus by (1), if $pot(u, t) \geq 1$, then $u^t = 1$ w.h.p. and if $pot(u, t) \leq -1$, $u^t = 0$ w.h.p. (recall that w.h.p. denotes with probability at least $1 - 1/n^c$ for some constant c). Aside from this fact, the only other consequence of (1) we use in our constructions is that $pot(u, t) = 0 \implies p(u, t) = 1/2$. That is, we use our spiking neurons entirely as random threshold gates, which fire w.h.p. when the incoming potential from their neighbors' spikes exceeds $\mathrm{b}(u)$, don't fire w.h.p. when the potential is below $\mathrm{b}(u)$, and fire randomly when the input potential equals the bias. It is an open question if there are any problems which require using the full power of the sigmoidal probability function.

2.3 Additional Notation

For any vector x we let x_i denote the value at its i^{th} position, starting from x_0. Given binary $x \in \{0, 1\}^n$, we use $\mathrm{dec}(x)$ to indicate the integer encoded by x. That is, $\mathrm{dec}(x) = \sum_{i=0}^{n-1} x_i \cdot 2^i$. Given an integer x we use $\mathrm{bin}(x)$ to denote its binary encoding, where the number of digits used in the encoding will be clear from context. We will often think of the firing pattern of a set of neurons as a binary string. If $B = \{y_1, ..., y_m\}$ is a set of m neurons then $B^t \in \{0, 1\}^m$ is the binary string corresponding to their firing pattern at time t. Since the input is typically fixed for some number of rounds, we often just write X to refer to the n-bit string corresponding to the input firing pattern.

Boolean Circuits. We mention that SNNs are similar to boolean circuits, which have received enormous attention in theoretical computer science. A circuit consists of gates (e.g., threshold gates, probabilistic threshold gates) connected in a directed acyclic graph. This restriction means that a circuit does not have feedback connections or self-loops, which we do use in our SNNs. While we do not work with circuits directly, for our lower bound, we show a transformation from an SNN to a linear threshold circuit. We sometimes refer to circuits as *feedforward* networks, indicating that their connections are cycle-free.

3 Neuro-RAM Network

In this section we prove our main upper bound:

▶ **Theorem 6** (Efficient Neuro-RAM Network). *There exists an SSN with $O(\sqrt{n})$ auxiliary neurons that solves indexing in $5\sqrt{n}$ rounds. Specifically, given inputs $X \in \{0,1\}^n$, and $Y \in \{0,1\}^{\log n}$, which are fixed for all rounds $t \in \{0,...,5\sqrt{n}\}$, the output neuron z satisfies: if $X_{\mathrm{dec}(Y)} = 1$ then $z^{5\sqrt{n}} = 1$ w.h.p. Otherwise, if $X_{\mathrm{dec}(Y)} = 0$, $z^{5\sqrt{n}} = 0$ w.h.p.*

Theorem 6 easily generalizes to other network sizes, giving Theorem 2, which states the full size-time tradeoff. Here we discuss the intuition behind the basic construction. The full details and proof are given in Appendices A.1 and A.2 of our full paper.

We divide the n input neurons X into $\sqrt{n}$ buckets each containing $\sqrt{n}$ neurons[3]:

$$X_0 = \{x_0, ..., x_{\sqrt{n}-1}\}, ..., X_{\sqrt{n}-1} = \{x_{(\sqrt{n}-1)\sqrt{n}}, ..., x_{n-1}\}.$$

Throughout, all our indices start from 0. We encode the firing pattern of each bucket X_i via the potential of a *single* neuron e_i. Set $w(x_{i\sqrt{n}+j}, e_i) = 2^{\sqrt{n}-j}$ for all $i, j \geq 0$. Thus, for every round t, the total potential contributed to e_i by the firing of the inputs in bucket X_i is:

$$\sum_{j=0}^{\sqrt{n}-1} x_{i\sqrt{n}+j} \cdot 2^{\sqrt{n}-j} = 2 \cdot \mathrm{dec}(\bar{X}_i). \tag{2}$$

where $\bar{X}_i$ is the reversal of X_i and $\mathrm{dec}(\cdot)$ gives the decimal value of a binary string, as defined in the preliminaries. We set $\mathrm{b}(e_i) = 2^{\sqrt{n}+2} + 2^{\sqrt{n}} - 1$. We will see later why this is an appropriate value. We defer detailed discussion of the remaining connections to e_i for now, first giving a general description of the network construction.

In addition to the *encoding neurons* $e_0, ..., e_{\sqrt{n}-1}$, we have *decoding neurons* $d_{0,k}, ..., d_{\sqrt{n}-1,k}$ for $k = 1, 2, 3$ ($3\sqrt{n}$ neurons total). The idea is to select a bucket X_i (via e_i) using the first $\log\sqrt{n} = \frac{\log n}{2}$ bits in the index Y. Let $Y_1 \stackrel{\text{def}}{=} \{y_0, ..., y_{\frac{\log n}{2}-1}\}$ and $Y_2 \stackrel{\text{def}}{=} \{y_{\frac{\log n}{2}}, ..., y_{\log n-1}\}$ be the higher and lower order bits of Y respectively. It is not hard to see that using $O(\sqrt{n})$ neurons we can construct a network that processes Y_1 and uses it to select e_i with $i = \mathrm{dec}(Y_1)$. When a bucket is selected, the potential of any e_j with $j \neq \mathrm{dec}(Y_1)$ is significantly depressed compared to that of e_i and so after this selection stage, only e_i fires.

We then use the decoding neurons to 'read' *each bit of the potential encoded in* e_i. The final output is selected from each of these bits using the lower order bits Y_2, which can again be done efficiently with $O(\sqrt{n})$ neurons. We call this phase the decoding phase since e_i encodes the value (in decimal) of its bucket X_i, and we need to decode from that value the bit of the appropriate neuron inside that bucket.

[3] Throughout we assume for simplicity that $n = 2^{2m}$ for some integer m. This ensures that $\sqrt{n}$, $\log n$, and $\log\sqrt{n}$ are integers. It will be clear that if this is not the case, we can simply pad the input, which only affects our time and network size bounds by constant factors.

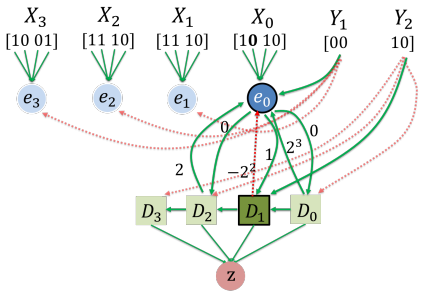


Figure 1 Illustration of the neuro-RAM module. D_i represents the set of 3 decoding neurons for each bit: $\{d_{i,1}, d_{i,2}, d_{i,3}\}$. The dotted lines from Y_1 and Y_2 represent connections to the buckets and decoding neurons which are not currently selected. The index encoded by Y is marked in bold and the selected encoding and decoding neurons are highlighted.

The decoding process works as follows: initially, e_i will fire only if the *first bit* of bucket i is on. Note that the weight from this bit to e_i is $2^{\sqrt{n}}$ and thus more than double the weight from any other input bit. Thus, by appropriately setting $\mathrm{b}(e_i)$, we can ensure that the setting of this single bit determines if e_i fires initially.

If the first bit is the correct bit to output (i.e., if the last $\frac{\log n}{2}$ bits of the index Y_2 encode position 0), this will trigger the output z to fire. Otherwise, we iterate. If e_i in fact fired, this triggers inhibition that cancels out the potential due to the first bit of bucket i. So e_i will now only fire if the *second bit* of X_i is on. If e_i did not fire, the opposite will happen. Further excitation will be given to e_i again ensuring that it can fire as long as the second bit of X_i is on. The network iterates in this way, successively reading each bit, until we reach the one encoded by Y_2 and the output fires. The first decoding neuron for position j, $d_{j,1}$, is responsible to triggering the output to fire if j is the correct bit encoded by Y_2. The second decoding neuron $d_{j,2}$ is responsible for providing excitation when e_i does not fire. Finally, the third decoding neuron $d_{j,3}$ provides inhibition when e_i does fire.

In Appendix A.1 of our full paper, we describe the first stage in which we use the first $\log n/2$ index bits to select the bucket to which the desired index belongs to.

In Appendix A.2, we discuss the second phase where we use the last $\log n/2$ bits of Y, to select the desired index inside the bucket i. Our successive decoding process is synchronized by a clock mechanism. This clock mechanism consists of chain of $\Theta(\sqrt{n})$ neurons that govern the timing of the $\Theta(\sqrt{n})$ steps of our decoding scheme. Roughly, traversing the $\sqrt{n}$ bits of the chosen i^{th} bucket from left to right, we spend $O(1)$ rounds checking if the current index is the one encoded by Y_2. If yes, we output the value at that index and if not, the clock "ticks" and we move to the next candidate.

Note that our model and the proof of Theorem 6 assume that no auxiliary neurons or the output neuron fire in round 0. However, in applications it will often be desirable to run the neuro-RAM for multiple inputs, with execution not necessarily starting at round 0. We can easily add a mechanism that 'clears' the network once it outputs, giving:

▶ **Observation 7** (Running Neuro-RAM for Multiple Inputs). *The neuro-RAM of Theorem 6 can be made to run correctly given a sequence of multiple inputs.*

4 Lower Bound for Neuro-RAM in Spiking Networks

In this section, we show that our neuro-RAM construction is nearly optimal. Specifically:

▶ **Theorem 8.** *Any SNN solving indexing with probability* $\geq 1 - \frac{1}{2n}$ *in* t *rounds must use* $\ell = \Omega\left(\frac{n}{t \log^2 n}\right)$ *auxiliary neurons.*

This result matches the lower bound for deterministic threshold gates of Fact 3 up to a $\log n$ factor, demonstrating that the use of randomness cannot give significant runtime advantages for the indexing problem. Even if one just desires a constant (e.g., 2/3) probability of success, a lower bound of $\Omega\left(\frac{n}{t \log^3 n}\right)$ applies: by replicating any network with success probability 2/3, $\Theta(\log n)$ times and taking the majority output (which can be computed with just a single additional auxiliary neuron), we obtain a network that solves the problem w.h.p.

4.1 High Level Approach and Intuition

The proof of Theorem 8 proceeds in a number of steps, which we overview here.

Reduction to Deterministic Indexing Circuit. We first observe that a network with ℓ auxiliary neurons solving the indexing problem in t rounds can be unrolled into a feedforward circuit with t layers and ℓ neurons per layer. We then show that the output distribution of a feedforward stochastic spiking circuit is identical to the output distribution if we first draw a deterministic linear threshold circuit (still with t layers and ℓ neurons per layer) from a certain distribution, and evaluate our input using this random circuit.

This equivalence is powerful since it allows us to apply Yao's principle [27]: assuming the existence of a feedforward SNN solving indexing with probability $\geq 1 - \frac{1}{2n}$, given any distribution of the inputs X, Y, there must be some deterministic linear threshold circuit $\mathcal{N}_D$ which solves indexing with probability $\geq 1 - \frac{1}{2n}$ over this distribution.

If we consider the uniform distribution over X, Y, this success probablity ensures via an averaging argument that for at least 1/2 of the 2^n possible values of X, $\mathcal{N}_D$ succeeds for at least a $1 - \frac{1}{2n}$ fraction of the possible Y inputs. Note, however, that the Y can only take on n possible values – thus this ensures that for 1/2 the possible values of X, $\mathcal{N}_D$ succeeds for *all possible values of the index* Y. Let $\mathcal{X}$ be the set of 'good inputs' for which $\mathcal{N}_D$ succeeds.

Lower Bound for Deterministic Indexing on a Subset of Inputs. We have now reduced our problem to giving a lower bound on the size of a deterministic linear threshold circuit which solves indexing on an arbitrary subset $\mathcal{X}$ of $\frac{1}{2} \cdot 2^n = 2^{n-1}$ inputs. We do this using VC dimension techniques inspired by the indexing lower bound of [16].

The key idea is to observe that if we fix some input $X \in \mathcal{X}$, then given Y, $\mathcal{N}_D$ evaluates the function $f_X : \{0,1\}^{\log n} \to \{0,1\}$, whose truth table is given by X. Thus $\mathcal{N}_D$ can be viewed as a circuit for evaluating any function $f_X(Y)$ for $X \in \mathcal{X}$, where the X inputs are 'programmable parameters', which effectively change the thresholds of some gates.

It can be shown that the VC dimension of the class of functions computable by a fixed a linear threshold circuit with m gates and variable thresholds is $O(m \log m)$. Thus for a circuit with t layers and ℓ gates per layer, the VC dimension is $O(\ell t \log(\ell t))$ [5]. Further, as a consequence of Sauer's Lemma [23, 25, 4], defining the class of functions $\mathcal{F} = \{f_X \text{ for any } X \in \mathcal{X}\}$, since $|\mathcal{F}| = |\mathcal{X}| = 2^{n-1}$, we have $VC(\mathcal{F}) = \Theta(n/\log n)$. These two VC dimension bounds, in combination with the fact that we know $\mathcal{N}_D$ can compute any function in $\mathcal{F}$ if its input bits are fixed appropriately, imply that $\ell t \cdot \log(\ell t) = \Omega(n/\log n)$. Rearranging gives $\ell = \Omega\left(\frac{n}{t \log^2 n}\right)$, completing Theorem 8.

4.2 Reduction to Deterministic Indexing Circuit

We now give the argument explained above in detail, first describing how any SNN that solves indexing w.h.p. implies the existence of a deterministic feedforward linear threshold circuit which solves indexing for a large fraction of possible inputs X.

▶ **Lemma 9** (Conversion to Feedforward Network). *Consider any SNN $\mathcal{N}$ with ℓ auxiliary neurons, which given input $X \in \{0,1\}^n$ that is fixed for rounds $\{0,...,t\}$, has output z satisfying $\Pr[z^t = 1] = p$. Then there is a feedforward SNN $\mathcal{N}_F$ (an SNN whose directed edges form an acyclic graph) with $(t-1)\cdot(\ell+1)$ auxiliary neurons also satisfying $\Pr[z^t = 1] = p$ when given X which is fixed for rounds $\{0,...,t\}$.*

Proof. Let $B = A \cup z$ – all non-input neurons. We produce $t-1$ duplicates of each auxiliary neuron $a \in A : \{a_1,...,a_{t-1}\}$ and of $z : \{z_1,...,z_{t-1}\}$, which are split into layers $B_1,...,B_{t-1}$. For each incoming edge from a neuron u to v and each $i \geq 2$ we add an identical edge from u_{i-1} to v_i. Any incoming edges from input neurons to u are added to each u_i for all $i \geq 1$. Finally connect z to the appropriate neurons in B_{t-1} (including z_{t-1} if there is a self-loop).

In round 1, the joint distribution of the spikes B_1^1 in $\mathcal{N}_F$ is identical to the distribution of B^1 in $\mathcal{N}$ since these neurons have identical incoming connections from the inputs, and since any incoming connections from other auxiliary neurons are not triggered in $\mathcal{N}$ since none of these neurons fire at time 0.

Assuming via induction that B_i^i is identically distributed to B^i, since B_{i+1} only has incoming connections from B_i and the inputs which are fixed, then the distribution of B_{i+1}^{i+1} identical to that of B^{i+1}. Thus B_{t-1}^{t-1} is identically distributed to B^{t-1}, and since the output in $\mathcal{N}_F$ is only connected to B_{t-1} its distribution is the same in round t as in $\mathcal{N}$. ◀

▶ **Lemma 10** (Conversion to Distribution over Deterministic Threshold Circuits). *Consider any spiking sigmoidal network $\mathcal{N}$ with ℓ auxiliary neurons, which given input $X \in \{0,1\}^n$ that is fixed for rounds $\{0,...,t\}$, has output neuron z satisfying $\Pr[z^t = 1] = p$. Then there is a distribution $\mathcal{D}$ over feedforward deterministic threshold circuits with $(t-1)\cdot(\ell+1)$ auxiliary gates that, for $\mathcal{N}_D \sim \mathcal{D}$ with output z, $\Pr_{\mathcal{D}}[z^t = 1] = p$ when presented input X.*

Proof. We start with $\mathcal{N}_F$ obtained from Lemma 9. This circuit has $t-1$ layers of $\ell+1$ neurons $B_1,...,B_{t-1}$. Given $X \in \{0,1\}^n$ that is fixed for rounds $\{0,...,t\}$, $\mathcal{N}_F$ has $\Pr[z^t = 1] = p$, which matches the firing probability of the output z in $\mathcal{N}$ in round t.

Let $\mathcal{D}$ be a distribution on deterministic threshold circuits that have identical edge weights to $\mathcal{N}_F$. Additionally, for any (non-input) neuron $u \in \mathcal{N}_F$, letting $\bar{u}$ be the corresponding neuron in the deterministic circuit, set the bias $b(\bar{u}) = \eta$, where η is distributed according to a logistic distribution with mean $\mu = b(u)$ and scale $s = \lambda$. The random bias is chosen independently for each u. It is well known that the cumulative density function of this distribution is equal to the sigmoid function. That is:

$$\Pr[\eta \leq x] = \frac{1}{1 + e^{-\frac{x - b(u)}{\lambda}}}. \tag{3}$$

Consider $\mathcal{N}_D \sim \mathcal{D}$ and any neuron u in the first layer B_1 of $\mathcal{N}_F$. u only has incoming edges from the input neurons X. Thus, its corresponding neuron $\bar{u}$ in $\mathcal{N}_D$ also only has incoming edges from the input neurons. Let $W = \sum_{x \in X} w(x,u) \cdot x^0$. Then we have:

$$\begin{aligned}
\Pr_{\mathcal{D}}[\bar{u}^1 = 1] = \Pr[W - \eta \geq 0] = \Pr[\eta \leq W] & \qquad \text{(Deterministic threshold)} \\
= \frac{1}{1 + e^{-\frac{W - b(u)}{\lambda}}} & \qquad \text{(Logistic distribution CDF (3))} \\
= \Pr[u^1 = 1]. & \qquad \text{(Spiking sigmoid dynamics (1))}
\end{aligned}$$

Let $\bar{B}_i$ denote the neurons in $\mathcal{N}_D$ corresponding to those in B_i. Since in round 1, all neurons in B_1 fire independently and since all neurons in $\bar{B}_1$ fire independently as their random biases are chosen independently, the joint firing distribution of B_1^1 is identical to that of $\bar{B}_1^1$.

By induction assume that $\bar{B}_i^i$ is identically distributed (over the random choice of deterministic network $\mathcal{N}_D \sim \mathcal{D}$) to B_i^i. Then for any $u \in B_{i+1}$ we have by the same argument as above, conditioning on some fixed firing pattern V of B_i in round i:

$$\Pr_{\mathcal{D}}[\bar{u}^{i+1} = 1 | \bar{B}_i^i = V] = \Pr[u^{i+1} = 1 | B_i^i = V].$$

Conditioned on $B_i^i = V$, the neurons in B_{i+1} fire independently in round $i+1$. So do the neurons of $\bar{B}_{i+1}$ due to their independent choices of random biases. Thus, the above implies that the distribution of $\bar{B}_{i+1}^{i+1}$ conditioned on $\bar{B}_i^i = V$ is identical to the distribution of B_{i+1}^{i+1}. This holds for all V, so, the full joint distribution of $\bar{B}_{i+1}^{i+1}$ is identical to that of B_{i+1}^{i+1}.

We conclude by noting that the same argument applies for the outputs of $\mathcal{N}_F$ and $\mathcal{N}_D$ since $\bar{B}_{t-1}^{t-1}$ is identically distributed to B_{t-1}^{t-1}. ◀

Lemma 10 is simple but powerful – it demonstrates that *the output distribution of a spiking sigmoid network is identical to the output distribution of a deterministic feedforward threshold circuit drawn from some distribution $\mathcal{D}$.* Thus, the performance of any SNN is equivalent to the performance of a randomized algorithm which first selects a linear threshold circuit using $\mathcal{D}$ and then applies this circuit to the input. This lets us show:

▶ **Lemma 11** (Application of Yao's Principle). *Assume there exists an SNN $\mathcal{N}$ with ℓ auxiliary neurons, which given any inputs $X \in \{0,1\}^n$ and $Y \in \{0,1\}^{\log n}$ which are fixed for rounds $\{0, ..., t\}$, solves indexing with probability $\geq 1 - \delta$ in t rounds. Then there exists a feedforward deterministic linear threshold circuit $\mathcal{N}_D$ with $(t-1) \cdot (\ell + 1)$ auxiliary gates which solves indexing with probability $\geq 1 - \delta$ given X, Y drawn uniformly at random.*

Proof. We use the idea of Yao's principle, employing an averaging argument to show that the existence of a randomized circuit succeeding with high probability implies the existence of a deterministic circuit succeeding with high probability on uniform random inputs. Specifically, given X, Y drawn uniformly at random, $\mathcal{N}$ solves indexing with probability $\geq 1 - \delta$ (since by assumption, it succeeds with this probability for any X, Y). By Lemma 10, $\mathcal{N}$ performs identically to an algorithm which selects a deterministic circuit from some distribution $\mathcal{D}$ and then applies it to the input. So at least one circuit in the support of $\mathcal{D}$ must succeed with probability $\geq 1 - \delta$ on X, Y drawn uniformly at random, since the success probability of $\mathcal{N}$ on the uniform distribution is just an average over the deterministic success probabilities. ◀

From Lemma 11 we have a corollary which concludes our reduction from our spiking sigmoid lower bound to a lower bound on deterministic indexing circuits.

▶ **Corollary 12** (Reduction to Deterministic Indexing on a Subset of Inputs). *Assume there exists an SNN $\mathcal{N}$ with ℓ auxiliary neurons, which, given inputs $X \in \{0,1\}^n$ and $Y \in \{0,1\}^{\log n}$ which are fixed for rounds $\{0, ..., t\}$, solves indexing with probability $\geq 1 - \frac{1}{2n}$ in t rounds. Then there exists some subset of inputs $\mathcal{X} \subseteq \{0,1\}^n$ with $|\mathcal{X}| \geq 2^{n-1}$ and a feedforward deterministic linear threshold circuit $\mathcal{N}_D$ with $(t-1) \cdot (\ell + 1)$ auxiliary gates which solves indexing given any $X \in \mathcal{X}$ and any index $Y \in \{0,1\}^{\log n}$.*

Proof. Applying Lemma 11 yields $\mathcal{N}_D$ which solves indexing on uniformly random X, Y with probability $1 - \frac{1}{2n}$. Let $\mathbb{I}(X, Y) = 1$ if $\mathcal{N}_D$ solves indexing correctly on X, Y and 0

otherwise. Then:

$$1 - \frac{1}{2n} \leq \frac{1}{n \cdot 2^n} \sum_{X \in \{0,1\}^n} \sum_{Y \in \{0,1\}^{\log n}} \mathbb{I}(X,Y) = \mathop{\mathrm{E}}_{X \text{ uniform from } \{0,1\}^n} \left[\frac{1}{n} \sum_{Y \in \{0,1\}^{\log n}} \mathbb{I}(X,Y) \right]$$

which in turn implies:

$$\mathop{\mathrm{E}}_{X \text{ uniform from } \{0,1\}^n} \left[\frac{1}{n} \sum_{Y \in \{0,1\}^{\log n}} (1 - \mathbb{I}(X,Y)) \right] \leq \frac{1}{2n}. \tag{4}$$

If $\frac{1}{n} \sum_{Y \in \{0,1\}^{\log n}} (1 - \mathbb{I}(X,Y)) \neq 0$ then $\frac{1}{n} \sum_{Y \in \{0,1\}^{\log n}} (1 - \mathbb{I}(X,Y)) \geq \frac{1}{n}$ just by the fact that the sum is an integer. Thus, for (4) to hold, we must have $\frac{1}{n} \sum_{Y \in \{0,1\}^{\log n}} (1 - \mathbb{I}(X,Y)) = 0$ for at least $\frac{1}{2}$ of the inputs $X \in \{0,1\}^n$. That is, $\mathcal{N}_D$ solves indexing for every input index on some subset $\mathcal{X}$ with $|\mathcal{X}| \geq \frac{1}{2}|\{0,1\}^n| \geq 2^{n-1}$. ◀

4.3 Lower Bound for Deterministic Indexing on a Subset of Inputs

With Corollary 12 in place, we now turn to lower bounding the size of a deterministic linear threshold circuit $\mathcal{N}_D$ which solves the indexing problem on some subset of inputs $\mathcal{X}$ with $|\mathcal{X}| \geq 2^{n-1}$. To do this, we employ VC dimension techniques first introduced for bounding the size of linear threshold circuits computing indexing on all inputs [16].

Consider fixing some input $X \in \mathcal{X}$, such that the output of $\mathcal{N}_D$ is just a function of the index Y. Specifically, with X fixed, $\mathcal{N}_D$ computes the function $f_X : \{0,1\}^{\log n} \to \{0,1\}$ whose truth table is given by X. Note that the output of $\mathcal{N}_D$ with X fixed is equivalent to the output of a feedforward linear threshold circuit $\mathcal{N}_D^X$ where each gate with an incoming edge from $x_i \in X$ has its threshold adjusting to reflect the weight of this edge if $x_i = 1$.

We define two sets of functions. Let $\mathcal{F} = \{f_X | X \in \mathcal{X}\}$ be all functions computable using some $\mathcal{N}_D^X$ as defined above. Further, let $\mathcal{G}$ be the set of all functions computabled by any circuit $\mathcal{N}'_D$ which is generated by removing the input gates of $\mathcal{N}_D$ and adjusting the threshold on each remaining gate to reflect the effects of any inputs with $x_i = 1$. We have $\mathcal{F} \subseteq \mathcal{G}$ and hence, letting $VC(\cdot)$ denote the VC dimension of a set of functions have: $VC(\mathcal{F}) \leq VC(\mathcal{G})$. We can now apply two results. The first gives a lower bound $VC(\mathcal{F})$:

▶ **Lemma 13** (Corollary 3.8 of [4] – Consequence of Sauer's Lemma [23, 25]). *For any set of boolean functions $\mathcal{H} = \{h\}$ with $h : \{0,1\}^{\log n} \to \{0,1\}$:*

$$VC(\mathcal{H}) \geq \frac{\log |\mathcal{H}|}{\log n + \log e}.$$

We next upper bound $VC(\mathcal{G})$. We prove in Appendix B of our full paper:

▶ **Lemma 14** (Linear Threshold Circuit VC Bound). *Let $\mathcal{H}$ be the set of all functions computed by a fixed feedforward linear threshold circuit with $m \geq 2$ gates (i.e., fixed edges and weights), where each gate has a variable threshold. Then: $VC(\mathcal{H}) \leq 3m \log m$.*

Applying the bounds of Lemmas 13 and 14 along with $VC(\mathcal{F}) \leq VC(\mathcal{G})$ gives:

▶ **Lemma 15** (Deterministic Circuit Lower Bound). *For any set $\mathcal{X} \subseteq \{0,1\}^n$ with $|\mathcal{X}| \geq 2^{n-1}$, any feedforward deterministic linear threshold circuit $\mathcal{N}_D$ with m non-input gates which solves indexing given any $X \in \mathcal{X}$ and any index $Y \in \{0,1\}^{\log n}$ must have $m = \Omega\left(\frac{n}{\log^2 n}\right)$.*

Proof. Let $\mathcal{F}$ and $\mathcal{G}$ be as defined in the beginning of the section. We have $VC(\mathcal{F}) \leq VC(\mathcal{G})$. At the same time, by Lemma 13 we have $VC(\mathcal{F}) \geq \frac{\log|\mathcal{F}|}{\log n + \log e} = \frac{\log|\mathcal{X}|}{\log n + \log e} \geq \frac{cn}{\log n}$ for some fixed constant c. By Lemma 14 we have $VC(\mathcal{G}) \leq 3m \log m$. We thus can conclude that $\frac{cn}{\log n} \leq 3m \log m$, and so $m = \Omega\left(\frac{n}{\log^2 n}\right)$. ◀

We conclude by proving our main lower bound:

Proof of Theorem 8. The existence of a spiking sigmoidal network with ℓ auxiliary neurons, solving indexing with probability $\geq 1 - \frac{1}{2n}$ in t rounds implies via Corollary 12 the existence of a feedforward deterministic linear threshold circuit with $(t-1)\ell + 1$ non-input gates solving indexing on some subset of inputs $\mathcal{X}$ with $|\mathcal{X}| \geq 2^{n-1}$. So by Lemma 15, $\ell \cdot t = \Omega\left(\frac{n}{\log^2 n}\right)$. ◀

5 Applications to Similarity Testing and Compression

5.1 Similarity Testing

▶ **Theorem 16** (Similarity Testing). *There exists an SNN with $O\left(\frac{\sqrt{n}\log n}{\epsilon}\right)$ auxilary neurons that solves the approximate equality testing problem in $O(\sqrt{n})$ rounds. Specifically, given inputs $X_1, X_2 \in \{0,1\}^n$ which are fixed for all rounds $t \in \{0, .., 5\sqrt{n}+2\}$, the output z satisfies w.h.p. $z^{5\sqrt{n}+2} = 1$ if $\mathrm{d_H}(X_1, X_2) \geq \epsilon n$. Further if $X_1 = X_2$ then $z^{5\sqrt{n}+2} = 0$ w.h.p.*

Our similarity testing network uses $K = \Theta\left(\frac{\log n}{\epsilon}\right)$ copies of our neuro-RAM network from Theorem 6, labeled $S_{1,k}$ and $S_{2,k}$ for all $k \in \{1, ..., K\}$. The idea is to employ $\log n$ auxiliary neurons $Y_k = y_{1,k}, ..., y_{\log n, k}$ whose values encode a *random index* $i \in \{0, ..., n-1\}$. By feeding the inputs (X_1, Y_k) and (X_2, Y_k) into S_1 and S_2, we can check whether X_1 and X_2 match at position i.

Checking $\Theta\left(\frac{\log n}{\epsilon}\right)$ different random indices suffices to identify if $\mathrm{d_H}(X_1, X_0) \geq \epsilon n$ w.h.p. Further, if $X_1 = X_0$, they will never differ at any of the checks, and so the output will never be triggered. We use:

▶ **Observation 17.** *Consider $X_1, X_2 \in \{0,1\}^n$ with $\mathrm{d_H}(X_1, X_0) \geq \epsilon n$. Let $i_1, ..., i_T$ be chosen independently and uniformly at random in $\{0, ..., n-1\}$. Then for $T = \frac{c \ln n}{\epsilon}$,*

$$\Pr[(X_1)_{i_t} = (X_2)_{i_t} \text{ for all } t \in 1, ..., T] \leq \frac{1}{n^c}.$$

Proof. For any fixed t, $\Pr[(X_1)_{i_t} = (X_2)_{i_t}] = 1 - \frac{\epsilon n}{n} = 1 - \epsilon$ as we select indices at random. Additionally, each of these events is independent since $i_1, ...i_T$ are chosen independently so: $\Pr[(X_1)_{i_t} = (X_2)_{i_t} \text{ for all } t \in 1, ..., T] \leq (1-\epsilon)^T = \left(1-\epsilon\right)^{1/\epsilon}{}^{c \ln n} \leq \frac{1}{e^{c \ln n}} \leq \frac{1}{n^c}$. ◀

5.1.1 Implementation Sketch

It is clear that the above strategy can be implemented in the spiking sigmoidal network model – we sketch the construction here. By Theorem 6, we require $O\left(\frac{\sqrt{n}\log n}{\epsilon}\right)$ auxiliary neurons for the $2K = \Theta\left(\frac{\log n}{\epsilon}\right)$ neuro-RAMs employed, which dominates all other costs.

It suffices to present a random index to each pair of neuro-RAMs S_1, k an S_2, k for $5\sqrt{n}$ rounds (the number of rounds required for the network of Theorem 6 to process an n-bit input). To implement this strategy, we need two simple mechanisms, described below.

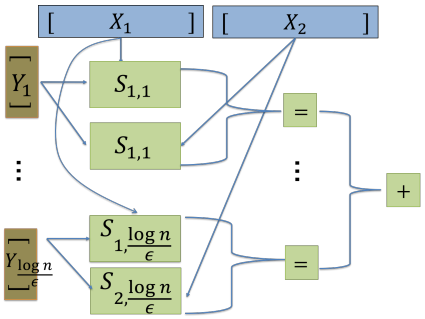


Figure 2 Illustration of our ϵ-approximate similarity testing network.

Random Index Generation: For each of the $\log n$ index neurons in Y_k we set $\mathrm{b}(y_i) = 0$ and add a self-loop $w(y_i, y_i) = 2$. In round 1, since they have no-inputs, each neuron has potential 0 and fires with probability 1/2. Thus, Y_k^1 represents a random index in $\{0, ..., n-1\}$. To propagate this index we can use a single auxiliary inhibitory neuron g, which has bias $\mathrm{b}(g) = 1$ and $w(x, g) = 2$ for every input neuron x. Thus, g fires w.h.p. in round 1 and continues firing in all later rounds, as long as at least one input fires.

We add an inhibitory edge from g to y_i for all i with weight $w(g, y_i) = -1$. The inhibitory edges from g will keep the random index 'locked' in place. The inhibitory weight of -1 prevents any y_i without an active self-loop from firing w.h.p. but allows any y_i with an active self-loop to fire w.h.p. since it will still have potential $\mathrm{b}(y_i) + w(y_i, y_i) - 1 = 1$.

If both inputs are 0, g will not fire w.h.p. However, here our network can just output 0 since $X_1 = X_2$ so it does not matter if the random indices stay fixed.

Comparing Outputs: We next handle comparing the outputs of $S_{1,k}$ and $S_{2,k}$. We use two neurons – $f_{1,k}$ and $f_{2,k}$. $f_{1,k}$ is excitatory and fires w.h.p. if at least one of $S_{1,k}$ or $S_{2,k}$ has an active output. $f_{2,k}$ is an inhibitor that fires only if *both* $S_{1,k}$ and $S_{2,k}$ have active outputs. We then connect $f_{1,k}$ to z with weight $w(f_{1,k}, z) = 2$ and connect $f_{2,k}$ with weight $w(f_2, z) = -2$ for all k. We set $\mathrm{b}(z) = 1$. Thus, z fires in round $5\sqrt{n} + 2$ w.h.p. if for some k, *exactly one* of $S_{1,k}$ or $S_{2,k}$ has an active output in round $5\sqrt{n}$ and hence an inequality is detected. Otherwise, z does not fire w.h.p. This gives the output condition of Theorem 16.

5.2 Randomized Compression

We conclude by discussing informally how our neuro-RAM can be applied beyond similarity testing to other randomized compression schemes. Consider the setting where we are given n input vectors $X_i \in \{0,1\}^d$. Let $\mathbf{X} \in \{0,1\}^{n \times d}$ denote the matrix of all inputs. Think of d as being a large ambient dimension, which we would like to reduce before further processing.

One popular technique is *Johnson-Lindenstrauss (JL) random projection*, where $\mathbf{X}$ is multiplied by a random matrix $\mathbf{\Pi} \in \mathbb{R}^{d \times d'}$ with $d' << d$ to give the compressed dataset $\tilde{\mathbf{X}} = \mathbf{X}\mathbf{\Pi}$. *Regardless of the initial dimension* d, if d' is set large enough, $\tilde{\mathbf{X}}$ preserves significant information about $\mathbf{X}$. $d' = \tilde{O}(\log n)$ is enough to preserve the distances between all points, $d' = \tilde{O}(k)$ is enough to use $\tilde{\mathbf{X}}$ for approximate k-means clustering or k-rank approximation [6, 8], and $d' = \tilde{O}(n)$ preserves the full covariance matrix of the input and so $\tilde{\mathbf{X}}$ can be used for approximate regression and many other problems [7, 22].

JL projection has been suggested as a method for neural dimensionality reduction [2, 11], where $\mathbf{\Pi}$ is viewed as a matrix of random synapse weights, which connect the input neurons representing $\mathbf{X}$ to the output neurons representing $\tilde{\mathbf{X}}$. While this view is quite natural, we

often want to draw $\mathbf{\Pi}$ with *fresh randomness* for each input $\mathbf{X}$. This is not possible using changing synapse weights, which evolve over a relatively long time scale. Fortunately, it is possible to simulate these random connections using our neuro-RAM module.

Typically, $\mathbf{\Pi}$ is sparse so can be multiplied by efficiently. In the most efficient construction [7], it has just a single nonzero entry in each row which is a randomly chosen ± 1 placed in a uniform random position. Thus, computing a single bit of $\tilde{\mathbf{X}} = \mathbf{X}\mathbf{\Pi}$ requires selecting on average d/d' random columns of $\mathbf{X}$, multiplying their entries by a random sign and summing them together. This can be done with a set of neuro-RAMS, each using $O(\sqrt{d})$ auxiliary neurons which select the random columns of $\mathbf{X}$. In total, we need $\tilde{O}(d/d')$ networks – the maximum column sparsity of $\mathbf{\Pi}$ with high probability, yielding $O(d^{3/2}/d')$ auxiliary neurons total. In contrast, a naive simulation of random edges using spiking neurons requires $\Theta(d)$ auxiliary neurons, which is less efficient whenever $d' > d^{1/2}$. Additionally, our neuro-RAMs can be reused to compute multiple entries of $\tilde{\mathbf{X}}$, which is not the case for the naive simulation.

Traditionally, the value of an entry of $\tilde{\mathbf{X}}$ is a real number, which cannot be directly represented in a spiking neural network. In our construction, the value of the entry is encoded in its potential, and we leave as an interesting open question how this potential should be decoded or otherwise used in downstream applications of the compression.

Acknowledgments. We thank Mohsen Ghaffari – the initial ideas regarding the importanace of the indexing module came up while Merav Parter was visiting him at ETH Zurich. We also thank Sergio Rajsbaum, Ron Rothblum, and Nir Shavit for helpful discussions.

References

1 Christina Allen and Charles F Stevens. An evaluation of causes for unreliability of synaptic transmission. *PNAS*, 1994.

2 Zeyuan Allen-Zhu, Rati Gelashvili, Silvio Micali, and Nir Shavit. Sparse sign-consistent Johnson–Lindenstrauss matrices: Compression with neuroscience-based constraints. *PNAS*, 2014.

3 Eric Allender. A note on the power of threshold circuits. In *Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, 1989.

4 Martin Anthony and Peter L Bartlett. *Neural network learning: Theoretical foundations*. Cambridge University Press, 2009.

5 Eric B Baum and David Haussler. What size net gives valid generalization? In *Advances in Neural Information Processing Systems 2 (NIPS)*, 1989.

6 Christos Boutsidis, Anastasios Zouzias, and Petros Drineas. Random projections for k-means clustering. In *Advances in Neural Information Processing Systems 23 (NIPS)*, 2010.

7 Kenneth L Clarkson and David P Woodruff. Low rank approximation and regression in input sparsity time. In *Proceedings of the 45th Annual ACM Symposium on Theory of Computing (STOC)*, 2013.

8 Michael B Cohen, Sam Elder, Cameron Musco, Christopher Musco, and Madalina Persu. Dimensionality reduction for k-means clustering and low rank approximation. In *Proceedings of the 47th Annual ACM Symposium on Theory of Computing (STOC)*, 2015.

9 A Aldo Faisal, Luc PJ Selen, and Daniel M Wolpert. Noise in the nervous system. *Nature Reviews Neuroscience*, 9(4):292–303, 2008.

10 Merrick Furst, James B Saxe, and Michael Sipser. Parity, circuits, and the polynomial-time hierarchy. *Theory of Computing Systems*, 17(1):13–27, 1984.

11 Surya Ganguli and Haim Sompolinsky. Compressed sensing, sparsity, and dimensionality in neuronal information processing and data analysis. *Annual Review of Neuroscience*, 2012.

12 Wulfram Gerstner and Werner M Kistler. *Spiking neuron models: Single neurons, populations, plasticity*. Cambridge University Press, 2002.

13 John J Hopfield, David W Tank, et al. Computing with neural circuits- a model. *Science*, 233(4764):625–633, 1986.

14 Bill G Horne and Don R Hush. On the node complexity of neural networks. *Neural Networks*, 7(9):1413–1426, 1994.

15 Eugene M Izhikevich. Which model to use for cortical spiking neurons? *IEEE Transactions on Neural Networks*, 15(5):1063–1070, 2004.

16 Pascal Koiran. VC dimension in circuit complexity. In *Proceedings of the 11th Annual IEEE Conference on Computational Complexity*, 1996.

17 Nancy Lynch, Cameron Musco, and Merav Parter. Computational tradeoffs in biological neural networks: Self-stabilizing winner-take-all networks. In *Proceedings of the 8th Conference on Innovations in Theoretical Computer Science (ITCS)*, 2017.

18 Wolfgang Maass. On the computational power of noisy spiking neurons. In *Advances in Neural Information Processing Systems 9 (NIPS)*, 1996.

19 Wolfgang Maass. Networks of spiking neurons: the third generation of neural network models. *Neural Networks*, 10(9):1659–1671, 1997.

20 Wolfgang Maass, Georg Schnitger, and Eduardo D Sontag. On the computational power of sigmoid versus boolean threshold circuits. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, 1991.

21 Marvin Minsky and Seymour Papert. Perceptrons. 1969.

22 Tamas Sarlos. Improved approximation algorithms for large matrices via random projections. In *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, 2006.

23 Norbert Sauer. On the density of families of sets. *Journal of Combinatorial Theory, Series A*, 13(1):145–147, 1972.

24 Michael N Shadlen and William T Newsome. Noise, neural codes and cortical organization. *Current Opinion in Neurobiology*, 4(4):569–579, 1994.

25 Saharon Shelah. A combinatorial problem; stability and order for models and theories in infinitary languages. *Pacific Journal of Mathematics*, 41(1):247–261, 1972.

26 Leslie G Valiant. *Circuits of the Mind*. Oxford University Press on Demand, 2000.

27 Andrew Chi-Chin Yao. Probabilistic computations: Toward a unified measure of complexity. In *Proceedings of the 18th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, 1977.

How Large Is Your Graph?*

Varun Kanade[1], Frederik Mallmann-Trenn[2], and Victor Verdugo[3]

1 **Department of Computer Science, University of Oxford, Oxford, United Kingdom, and The Alan Turing Institute, London, United Kingdom**
2 **Département d'Informatique, École normale supérieure, PSL Research University, Paris, France, and Department of Computer Science, Simon Fraser University, Burnaby, Canada**
3 **Département d'Informatique, École normale supérieure, PSL Research University, Paris, France, and Departmento de Ingeniería Industrial, Universidad de Chile, Santiago, Chile**

Abstract

We consider the problem of estimating the graph size, where one is given only local access to the graph. We formally define a query model in which one starts with a *seed* node and is allowed to make queries about neighbours of nodes that have already been seen. In the case of undirected graphs, an estimator of Katzir *et al.* (2014) based on a sample from the stationary distribution π uses $O\left(\frac{1}{\|\pi\|_2} + d_{\text{avg}}\right)$ queries; we prove that this is tight. In addition, we establish this as a lower bound even when the algorithm is allowed to crawl the graph arbitrarily; the results of Katzir *et al.* give an upper bound that is worse by a multiplicative factor $t_{\text{mix}}(1/n^4)$.

The picture becomes significantly different in the case of directed graphs. We show that without strong assumptions on the graph structure, the number of nodes cannot be predicted to within a constant multiplicative factor without using a number of queries that are at least linear in the number of nodes; in particular, rapid mixing and small diameter, properties that most real-world networks exhibit, do not suffice. The question of interest is whether any algorithm can beat breadth-first search. We introduce a new parameter, generalising the well-studied conductance, such that if a suitable bound on it exists and is known to the algorithm, the number of queries required is sublinear in the number of edges; we show that this is tight.

1998 ACM Subject Classification G.3 Probability and Statistics

Keywords and phrases Estimation, Random Walks, Social Networks

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.34

1 Introduction

Networks contain a wealth of information and studying properties of networks may yield important insights. However, most networks of interest are very large and ordinary users may have rather restricted access to them. One of most basic questions about networks is the number of nodes contained in them. For example, the number of pages on the world wide web (WWW) is estimated to be just shy of 50 billion at the time of writing.[1] Facebook currently reports having about one and three quarter billion users;[2] Twitter reports having about 300 million active users. It is undesirable to rely on a small number of sources for such information. At times we might be interested in more specific graphs for which there is

* See [11] for the full version, `http://arxiv.org/abs/1702.03959`.
1 `www.worldwidewebsize.com`
2 Here, billion refers to one thousand million, not a million million.


© Varun Kanade, Frederik Mallmann-Trenn, and Victor Verdugo;
licensed under Creative Commons License CC-BY

no public information available at all. Is there a way to estimate the total number of nodes using rather limited access to these graphs?

Our model is motivated by the kind of access ordinary users may have to graphs of interest. For example, most social network companies provide some sort of an application programming interface (API). In the case of the world wide web, one option is to simply crawl. The graph query access models we use are formally defined in Section 2.1. For now, we mention four specific networks, each of which captures an important modelling aspect. First, Facebook is an undirected graph of friendships, and as long as privacy settings allow it, it is possible to request the number of friends for a given user and the identity of the friends. Second, the world wide web, which is a directed graph – it is possible to extract out-links on a given webpage; however, there is no obvious method to access all in-links. The third is what we refer to as the "fan" network – for a specific user it is possible to query who she is a fan of, however in terms of her fans only the number is revealed.[3] And finally, the fourth is the twitter network, which is directed, however both the followers and followees of a given user are accessible, as far as privacy settings allow. Obviously, the method to estimate the number of nodes may be rather different in each case. It is worth mentioning that the twitter network can essentially be treated as an undirected graph, however it still leaves open the possibility that differentiating in-links and out-links leads to better estimators.

While memory and computational requirements do act as constraints when dealing with large graphs, possibly the most important one is the rate limits set on queries made using the API. Even when crawling the web, there is the risk of simply being blocked if large volumes of requests are sent to the same server. Thus, the most scarce resource in this instance is the number of queries made about the graph. Our query model counts these costs strictly – essentially every time a new node is discovered, its degree is revealed at unit cost and a unit cost is incurred for every neighbour requested.

1.1 Related Work

There is a large body of literature on estimating statistical properties of graphs, reflecting the relevance of and interest in studying complex networks. Much of this work is in the more applied literature and in particular, we were unable to pin down a precisely defined query model. The work most closely related to ours is that of Katzir *et al.* [13]. They consider the setting where a random sample drawn from the stationary distribution of the random walk is available. If the random walk mixes rapidly and a suitable bound on the mixing time is known, this can be simulated in the models we consider. They show that $O\left(\frac{1}{\|\pi\|_2} + d_{\text{avg}}\right)$ queries suffice, where π is the stationary distribution and d_{avg} is the average degree. We show that this is tight when given access to a sample from the stationary distribution. When only neighbour queries are allowed, we show that this bound is tight up to a multiplicative factor of the mixing time and other polylogarithmic factors. It is worth mentioning what this bound actually yields in graphs where the average degree is small, something common to most real world networks. It is always the case that $\|\pi\|_2^{-1} \leq \sqrt{n}$, where equality holds in the case of regular graphs. Thus, the number of queries required is significantly sublinear in the number of nodes. For graphs with power law degree distributions with parameter $\beta = 2$, Katzir *et al.* calculated that $\|\pi\|_2^{-1} = O(n^{1/4}\log(n))$. See Section 3.1 for further discussion.

In more recent work, Hardiman and Katzir give another estimator based on counting

[3] Although it is not obvious which of the extant networks have this property, there are close approximations such as Blogger. In any case, it is a very natural intermediate model between the second and the fourth.

shared neighbours [12]. They give a slightly better bound on the number of nodes sampled than the earlier work of Katzir *et al.* [13]. However, it is unclear that this can be implemented efficiently in the query model in our paper. Cooper *et al.* [5] obtain estimators for the number of edges, triangles and nodes. The estimators are based on random walks over the graph, but in particular, for estimating the number of nodes, the transition probabilities are not inversely proportional to the degree. Their estimator relies on counting the time required for the k^{th} return to a particular vertex. It seems unlikely that either their random walk or their estimator can be implemented in a query efficient manner. Brautbar and Kearns [3] show that estimating the size of cycle takes $\Omega(\sqrt{n})$ queries in the model where one has access two types of queries: one can query (i) outgoing edges and (ii) a node chosen uniformly at random. They also obtained bounds on the query complexity of the maximum degree and the clustering coefficient. Musco *et al.* [17] develop a distributed collision based approach to estimate the graph size and the average degree – in this model several random walks traverse the graph and count the number of collisions with other random walks. Dasgupta *et al.* [7] provide an estimator of the average degree that uses $O(\log(U)\log\log(U))$ samples, where U is an upper bound on the maximum degree. Somewhat surprisingly, this seems to be an easier task than estimating the number of nodes.

Cooper and Frieze [4] estimate the graph size in a dynamic setting where the graph grows over time and there exists an agent that is allowed to move through the network every time a new node arrives. They prove that this agent visits asymptotically a constant fraction of the vertices. If one has access to the complete neighbourhood of a node when it is visited, it is shown by Mihail *et al.* that the expected time in which a walk discovers a power law random graph is sublinear [16]. There has been some older theoretical work on estimating the size of graphs motivated primarily by search algorithms [14, 15, 18]. They consider trees and directed acyclic graphs, however, the model of graph access is unrelated and they do not present bounds on the estimates.

Related to the question of estimating graph properties is that of testing whether graphs have specific properties. Property testing has received much attention in the last two decades with properties of undirected graphs being one of the important areas of focus (see *e.g.,* [9]). Directed graphs have received somewhat less attention; there are two main query models, *unidirectional* where only out-neighbours may be queried, and *bidirectional* where both in and out-neighbours may be queried [1]. Bender and Ron showed that there exist properties such as strong connectivity, where the query complexity may be either $O(1)$ or $\Omega(\sqrt{n})$ depending on the model used, even when allowing two-sided error. More recently, Czumaj *et al.* have shown that if a property can be tested with constant query complexity in the bidirectional setting, it can be tested with sublinear query complexity in the unidirectional model [6]. Although, these query models are closely related to the ones in this paper, a crucial aspect exploited by most property testing algorithms is the ability to sample nodes uniformly at random, something that is not available in our setting.

A closely related line of work is that on estimating properties of distributions from samples; of most interest, in our case is the support size. This problem has a long history going back at least to Good and Turing [10]. We only mention a few recent relevant results here. Given access to uniformly sampled elements from a set, $O(\sqrt{n})$ samples suffice to derive a good estimate of the set size using the birthday problem [8]. However, it is not clear how to sample from the uniform distribution over the nodes of the graph in the query model we consider. Valiant and Valiant show that support size can be estimated to a good accuracy using $O(n/\log(n))$ samples for any distribution [19, 20]. However, their result requires that any element in the support has $\Omega(1/n)$ probability mass, something that is not true for

the stationary distribution of a random walk on a graph. (In the case of directed graphs this problem becomes even more severe, since some nodes may have exponentially small stationary probability mass.)

1.2 Our contributions

Firstly, our contribution is to express the problem of estimating graph properties formally. As discussed in the introduction, networks of interest vary significantly in terms of what access might be easily available to an ordinary user. Keeping in mind the examples of Facebook, the web, the fan-network and Twitter, we introduce different types of oracles that provide access to the graph.

The focus of this work is on estimating the number of nodes. For any $\varepsilon > 0$ and $\delta > 0$, we say that an algorithm (with access to a query oracle) provides an ε-accurate estimate of the number of nodes, if with probability at least $1 - \delta$ it outputs $\widehat{n}$, such that $|n - \widehat{n}| \leq \varepsilon n$. The main quantity to be optimised is the number of oracle queries, though all algorithms considered in this paper are also computationally highly efficient. Allowed queries are defined precisely in Section 2.1. Here, we point out that a unit cost must be paid for every disclosed neighbour as well as to know the degree of a node. All algorithms have access to the *identifier* of one *seed* node in the graph to begin with. Throughout we will assume that ε and δ are constants and the use of $O(\cdot)$ and $\Omega(\cdot)$ notation in this paper hides all dependence on ε and δ.

Undirected Graphs. Katzir *et al.* [13] implicitly assume the ability to sample from the stationary distribution. They show that in this setting $O\left(\frac{1}{\|\pi\|_2} + d_{\text{avg}}\right)$ samples from the stationary distribution π suffice, where d_{avg} is the average degree. If the graph is connected and a suitable bound on the mixing time exists which is known to the algorithm, $O(t_{\text{mix}}(1/n^4))$[4] queries suffice to draw one node from a distribution that is close (up to inverse polynomial factors in variation distance) to the stationary distribution using only neighbour queries. This gives an upper bound of $O\left(t_{\text{mix}}(1/n^4) \cdot \left(\frac{1}{\|\pi\|_2} + d_{\text{avg}}\right)\right)$ queries with a neighbour query oracle (Corollary 2). In terms of lower bounds, we establish that

- (Theorem 5) Any algorithm with access to random samples from the *stationary distribution* π and outputs a 0.1-accurate estimate of the number of nodes with probability at least 0.99, requires $\Omega\left(\frac{1}{\|\pi\|_2} + d_{\text{avg}}\right)$ samples.
- (Theorem 7) Any algorithm with access to neighbour queries and outputs a 0.1-accurate estimate of the number of nodes with probability at least 0.99, requires $\Omega\left(\frac{1}{\|\pi\|_2} + d_{\text{avg}}\right)$ queries.

We remark that there is a gap between the upper bound and lower bound when considering an oracle with neighbour access. A question left open by our work is whether the multiplicative factor of $t_{\text{mix}}(1/n^4)$ is required, or whether a more efficient estimator can be designed.

Directed Graphs. The estimator of Katzir *et al.* is not applicable in the setting when graphs are directed, unless the query model allows in-neighbour queries as well as out-neighbour queries, in which case all results in the undirected setting hold. The reason for this is that even if one did receive a sample drawn from the stationary distribution, it is no longer the case that $\pi_v \propto \deg(v)$, a crucial property exploited by the estimator of Katzir *et al.*. In

[4] $t_{\text{mix}}(1/n^4)$ is the expected time it takes until the random walk is $1/n^4$-close to the stationary distribution w.r.t. to the total variation distance.

fact, unless strong assumptions are made on the relationship between the in-degree and the out-degree (*e.g.,* being Eulerian), no simple expression for π in terms of degrees exists.[5]

We provide constructions of graphs that demonstrate that low average degree, rapid mixing and small diameter are not sufficient to design algorithms to estimate the graph size with sublinear (in n) query complexity. In fact, we show that when only given access to a sample from the stationary distribution, a superpolynomially large sample may be required even for graphs with constant average degree, logarithmic diameter and rapid mixing. The reason for this is that in the case of directed graphs most of the stationary mass can be concentrated on a very small number of nodes.

In order to understand the query complexity when neighbour-queries are allowed, we define a new parameter called *general conductance*. Roughly speaking a graph has ε-general conductance ϕ_ε, if every set containing at most $(1-\varepsilon)n$ nodes has at least ϕ_ε fraction of directed edges going out. If a suitable bound on the value of ϕ_ε is known, we show that a simple edge-sampling algorithm outputs an estimate to within relative error ε while using $O(n/\phi_\varepsilon)$ queries (Theorem 10) and that this is almost tight, in the sense that any algorithm that outputs an estimate that is even slightly better than ε requires at least $\Omega(n/\phi_\varepsilon)$ queries (Theorem 12). The algorithm only requires access to out-neighbours, while the lower bound holds even with respect to an oracle that allows in-neighbour queries, which means that it also applies in the case of undirected graphs.

1.3 Discussion

In terms of improvements to our results, the most interesting question is whether any reasonable subclass of directed graphs are amenable to significantly improved query complexity (ideally sublinear) for the problem of estimating the number of nodes. The constructions in Section 4 show that having low average degree, small diameter and rapid mixing is not enough. For undirected graphs, it is an interesting question whether the extra factor of mixing time t_{mix} must be paid, when only neighbour queries are allowed. It is conceivable that an improved estimator that can handle correlated pairs of nodes can be designed, so as to not waste all but one query for every t_{mix} queries. Finally, it'd be interesting to study the question of estimating other properties of graphs, number of edges, number of triangles, *etc.* in this framework.

The model choices we made reflect the publicly available access to most extant networks; in particular, we were very stringent with accounting – every neighbour query counts as unit cost. Many APIs return the list of neighbours, although in chunks of a fixed size, *e.g.,* 100. It is hard to argue that 100 should be treated as constant in the context of social networks. Nevertheless, if we wanted a list of all followers of Barack Obama, this would still result in a very large number of queries. A natural extension to the query model in this paper is to allow the entire neighbourhood (possibly restricted to the out-neighbourhood in the case of directed graphs) to be revealed at unit cost. Estimators such as the one involving common neighbours of Hardiman and Katzir [12] can be implemented efficiently under such a model. Understanding the query complexity of estimation in these more powerful models is an interesting question.

[5] We mention that the distribution is closely related to the PageRank distribution. However, the PageRank random walk jumps to a uniformly random node in the graph with a small probability; this is done to avoid problems when the graph is not strongly connected.

2 Model and Preliminaries

Graphs. Graphs $G = (V, E)$ considered in this paper may be directed or undirected; typically we assume $|V| = n$ and $|E| = m$, though if there is scope for confusion we use $|V|$ or $|E|$ explicitly. For undirected graphs, for a node $v \in V$, we denote by $N(v)$ its *neighbourhood*, *i.e.,* $N(v) = \{w \mid \{v, w\} \in E\}$, and its degree by $\deg(v) := |N(v)|$. In the case of directed graphs, we denote $N^+(v) := \{w \mid (v, w) \in E\}$ its *out-neighbourhood* and by $\deg^+(v) := |N^+(v)|$ its *out-degree*. Similarly, $N^-(v) := \{u \mid (u, v) \in E\}$ denotes its *in-neighbourhood* and $\deg^-(v) := |N^-(v)|$ its *in-degree*. Furthermore, d_{avg} denotes the average degree, i.e, $\sum_{v \in V} \deg(v)/n$. Whenever there is scope for confusion, we use the notations $\deg_G(u)$, $N_G(v)$, $d_{\text{avg}}(G)$, *etc.* to emphasise that the terms are with respect to graph G.

Random walks in graphs. A discrete-time lazy random walk $(X_t)_{t \geq 0}$ on a graph $G = (V, E)$ is defined by a Markov chain with state space V and transition matrix $P = (p(u, v))_{u,v \in V}$ defined as follows: For every $u \in V$, $p(u, u) = 1/2$ (*Laziness*). In the undirected setting, for every $v \in N(u)$, $p(u, v) = 1/(2 \deg(u))$. In the directed setting, for every $v \in N^+(u)$, $p(u, v) = 1/(2 \deg^+(u))$. The transition probabilities can be expressed in matrix form as $P = (I + \mathcal{D}^{-1}A)/2$, where A is the adjacency matrix of G, $\mathcal{D}$ is the diagonal matrix of node degrees (only out-degrees if G is directed), and I is the identity. Let $p^t(u, \cdot)$ denote the distribution over nodes of a random walk at time step t with $X_0 = u$. For the most part, we will consider (strongly) connected graphs. Together with laziness, this ensures that the stationary distribution of the random walk, denoted by π, is unique and given by $\pi P = \pi$. In the undirected case, the form of the stationary distribution is particularly simple, $\pi(u) = \deg(u)/(2|E|)$; furthermore, the random walk is reversible, *i.e.,* $\pi(u)p(u, v) = \pi(v)p(v, u)$. As before, π_G is used to emphasise that the stationary distribution is respect to graph G.

Mixing time. To measure how far $p^t(u, \cdot)$ is from the stationary distribution we consider the *total variation distance*; for distributions μ, ν over sample space Ω the total variation distance is $\|\mu - \nu\|_{\mathsf{TV}} = \frac{1}{2} \sum_{x \in \Omega} |\mu(x) - \nu(x)|$. The *mixing time* of the random walk is defined as $t_{\text{mix}} := \max_{u \in V} \min\{t \geq 1 \mid \|p^t(u, \cdot) - \pi\|_{\mathsf{TV}} \leq e^{-1}\}$. Although the choice of e^{-1} is arbitrary, it is known that after $t_{\text{mix}} \log(1/\varepsilon)$ steps, the total variation distance is at most ε.

2.1 Query Model

In this section, we formally define the query model that allows us to access the graph. We consider four different neighbour query oracles, $\mathcal{O}$, $\overrightarrow{\mathcal{O}}$, $\overrightarrow{\mathcal{O}}(1)$ and $\overrightarrow{\mathcal{O}}(2)$. We also assume that all oracles have graphs stored as adjacency lists. In the case of directed graphs, there are two adjacency lists for every vertex, one for in-neighbours and one for out-neighbours. No assumption is made regarding the order in which the adjacency lists are stored. In words, $\mathcal{O}$ captures access to undirected graphs such as social networks like Facebook, $\overrightarrow{\mathcal{O}}$ captures directed graphs such as the world wide web, where only out-edges are available, $\overrightarrow{\mathcal{O}}(1)$ captures directed graphs such as fan-networks, where the in-degree but not in-neighbours may be available, and $\overrightarrow{\mathcal{O}}(2)$ captures directed graphs such as Twitter where access is available to both in-edges and out-edges.

All oracles also make use of labelling functions[6] ; for some space L, a labelling function $\ell : V \to L$ used by an oracle is an injection. We allow ℓ to be defined dynamically. The labelling function we use throughout the paper is the *consecutive labelling function* defined as follows: The label set is $L = \mathbb{N}$. If S denotes all the vertices labelled by the oracle so far, for a new vertex $v \notin S$ picked by the oracle, the label is assigned as follows: if $S = \emptyset$, $\ell(v) = 1$, else $\ell(v) = \max\{\ell(u) \mid u \in S\} + 1$. For these neighbour query oracles, any algorithm can essentially make two types of queries, (i) init, and (ii) (l, i) for undirected graphs or (l, i, etype) for directed graphs, where etype is either in, out. We assume that oracles use a labelling function ℓ.

(i) init: The oracle initialises a set $S := \{v\}$, where v is chosen to be an arbitrary node in the graph. The different oracles respond as defined below.
- $\mathcal{O}$ responds with $(\ell(v), \deg(v))$ for some $v \in V$. $\overrightarrow{\mathcal{O}}$ responds with $(\ell(v), \deg^+(v))$ for some $v \in V$.
- $\overrightarrow{\mathcal{O}}(1)$ and $\overrightarrow{\mathcal{O}}(2)$ both respond with $(\ell(v), \deg^+(v), \deg^-(v))$.

(ii) (l, i) or (l, i, etype): $\mathcal{O}$ only responds to query of type (l, i) and the remaining to queries of the type (l, i, etype).
- For query (l, i), if there is $v \in S$ such that $\ell(v) = l$ and $i \leq \deg(v)$, then $\mathcal{O}$ returns $(\ell(u), \deg(u))$, where u is the i^{th} element in the adjacency list of v. The oracle updates $S \leftarrow S \cup \{u\}$. Otherwise it returns null.
- For query (l, i, out), if there is $v \in S$ such that $\ell(v) = l$ and $i \leq \deg^+(v)$, then $\overrightarrow{\mathcal{O}}$ returns $(\ell(u), \deg^+(u))$, while both $\overrightarrow{\mathcal{O}}(1)$ and $\overrightarrow{\mathcal{O}}(2)$ return $(\ell(u), \deg^+(u), \deg^-(u))$ where u is the i^{th} element on the out-neighbour adjacency list of v. The oracles all update $S \leftarrow S \cup \{u\}$. Otherwise, all oracles return null.
- For query (l, i, in), $\overrightarrow{\mathcal{O}}$ and $\overrightarrow{\mathcal{O}}(1)$ always return null. If there exists $v \in S$, such that $\ell(v) = l$ and $i \leq \deg^-(v)$, then $\overrightarrow{\mathcal{O}}(2)$ returns $(\ell(u), \deg^+(u), \deg^-(u))$, where u is the i^{th} element on the adjacency list of in-neighbours of v; otherwise, it returns null. In the case of $\overrightarrow{\mathcal{O}}(2)$, it updates $S \leftarrow S \cup \{u\}$.

It is worth pointing out that a response of null provides no new information to the algorithm. The algorithm only knows that it hadn't received l as a label before or that the degree of the node with label l is strictly smaller than i, things it already knew. This is because the oracle maintains a history of past queries; if this were not the case the algorithm could generate (random) labels and try to find out whether they corresponded to nodes in the graph. However, even for oracles that don't maintain such state explicitly, by choosing ℓ to be collision-resistant hash function, essentially the same behaviour can be achieved.

In some of our proofs, we also allow oracles to return side information. These are denoted by a superscript s, *e.g.,* $\mathcal{O}^s$. Access to oracles with (truthful) side information can only help to reduce query complexity, since an algorithm can choose to ignore any side information it receives. Finally, we say that an algorithm is *sensible* if it does not make a query to which it already knows the answer. It is clear that given any algorithm, there is a sensible algorithm that is at least as good as the original algorithm. The *sensible* algorithm merely simulates the original algorithm and whenever the original algorithm made a query to which the answer was known, the *sensible* algorithm simulates the oracle response. Finally, we consider the *stationary query oracle*, $\mathcal{O}^\pi$, which when queried returns $(\ell(v), \deg(v))$, where $v \sim \pi$; π is the stationary distribution.

[6] We do assume that there is unit cost in sending the label of a node to the algorithm, thus implicitly we may think of every label having a bit-representation that is logarithmic in the size of the graph. However, the bit-length of the label reveals minimal information, which we assume that the algorithm has access to anyway.

3 Undirected graphs

In this section, we focus on the question of estimating the number of nodes in undirected graphs. We show that the results obtained by Katzir *et al.* [13] are essentially tight, up to a factor of the mixing time and polylogarithmic terms in n. This suggests that rapid mixing is a critical condition for being able to estimate the size of the graph.

3.1 Results of Katzir *et al.*

In this section, we discuss the result of Katzir *et al.* [13] regarding estimating the number of nodes in a graph. Though, they don't discuss this formally, their method essentially boils down to having access to the stationary query oracle $\mathcal{O}^\pi$. For the sake of completeness, we outline the estimator of Katzir *et al.* here. Let

$$X = \{(\ell(x_1), \deg(x_1)), (\ell(x_2), \deg(x_2)), \ldots, (\ell(x_r), \deg(x_r))\}$$

be drawn from the stationary query oracle $\mathcal{O}^\pi$. Let $\Psi_1 = \sum_{i=1}^{r} \deg(x_i)$ and $\Psi_{-1} = \sum_{i=1}^{r} 1/\deg(x_i)$; let $C = \sum_{i \neq j} \mathbb{1}(\ell(x_i) = \ell(x_j))$ denote the random variable counting the number of collisions. Then, it is fairly straightforward to see that $\mathbb{E}[\Psi_1 \Psi_{-1}] = r + 2n\binom{r}{2} \|\pi\|_2^2$ and $\mathbb{E}[C] = 2\binom{r}{2} \|\pi\|_2^2$. Katzir *et al.* use the the following as an estimator for the number of nodes $\widehat{n} := \frac{\Psi_1 \Psi_{-1} - r}{C}$. They prove the following result.

▶ **Theorem 1** (Katzir et al. [13]). *Let $\varepsilon > 0$ and $\delta > 0$. Suppose that the number of samples is $r \geq 1 + \frac{32}{\varepsilon^2 \delta} \max\left\{\frac{1}{\|\pi\|_2}, d_{\text{avg}}\right\}$, where $d_{\text{avg}} = 2m/n$. Then, $\mathbb{P}[|\widehat{n} - n| \geq \varepsilon n] \leq \delta$.*

Given a bound T on t_{mix} and access to oracle $\mathcal{O}$, the stationary query oracle $\mathcal{O}^\pi$ can be approximately simulated. (We assume that the graph is connected.) We simply perform a random walk on the graph for $T \log(1/\rho)$ steps, if a sample from a distribution at most ρ far from π in total variation distance is desired. Using the above theorem, we get the following straightforward corollary. The proof of the corollary follows by simulating s random walks for $O(T \log s)$ time steps (queries) ensuring that each random walk has a total variation distance of s^{-2} from π.

▶ **Corollary 2.** *Let $\varepsilon > 0$ and $\delta > 0$. For a connected, undirected graph, $G = (V, E)$ let T be such that $t_{\text{mix}} \leq T$. Then there exists an algorithm that given T and access to oracle $\mathcal{O}$, outputs $\widehat{n}$, such that, $\mathbb{P}[|\widehat{n} - n| \geq \varepsilon n] \leq \delta$. Furthermore, the number of queries made by the algorithm to $\mathcal{O}$ is $O(Ts \log s)$, where $s = O\left(\frac{1}{\varepsilon^2 \delta} \cdot \max\left\{\frac{1}{\|\pi\|_2}, d_{\text{avg}}\right\}\right)$.*

The key question is, are these bounds tight? In Section 3.2, we give much stronger lower bounds, where we show that for any valid degree sequence $\mathbf{d} = (d_1, \ldots, d_n)$ on n nodes, there exists a sequence $\mathbf{d}'$ on $2n$ nodes, such that any algorithm with access to a stationary oracle for either sequence $\mathbf{d}$ or $\mathbf{d}'$, cannot distinguish between the two unless it makes $\Omega\left(\frac{1}{\|\pi\|_2} + d_{\text{avg}}\right)$ queries. Here, we discuss simple instances where the bound in Theorem 1 is tight; we don't give formal proofs which are relatively straightforward.

- Let G_1 and G_2 be d-regular graphs on n and $2n$ nodes respectively, for $d < n$. Thus, samples from the stationary distribution are essentially just uniformly chosen random nodes in the graph. As the degrees are identical, they reveal no additional information. Thus, the algorithm has to query until a collision is observed, which requires $\Omega(\sqrt{n})$ queries. Clearly for these graphs $\frac{1}{\|\pi\|_2} = \Theta(\sqrt{n})$.

- We define the *sun* graph as a K_n with an additional edge out of each vertex to a node with degree 1. The *bright sun* graph is the same as the sun graph, except that there is a path of length 2 coming out of each vertex, rather than just an edge. Under the stationary oracle model, $\Omega(n)$ queries are required before any degree 1 or 2 nodes are returned since the total probability on these nodes is $O(1/n)$. As can be seen for these graphs $d_{\text{avg}} = \Theta(n)$.

The above examples show that there are graphs for which the bound of Katzir *et al.* is tight. In this paper, we show a significantly stronger statement – given any degree sequence $\mathbf{d}$, there are graphs for which the bound is almost tight.

3.2 Lower Bounds in the Stationary Query Model

In this section, we show that for any undirected, connected graph $G = (V, E)$, there exists an undirected, connected graph $\widetilde{G}$, such that any algorithm which has access to either the oracle $\mathcal{O}^{\pi}(G)$ or $\mathcal{O}^{\pi}(\widetilde{G})$, cannot distinguish between the two with significant probability without making $\Omega\left(\frac{1}{\|\pi\|_2} + d_{\text{avg}}\right)$ queries. Thus, it cannot output an estimate $\widehat{n}$ satisfying $|\widehat{n} - |V|| < |V|/2$ with probability 2/3. The graphs $(\widetilde{G})$ we construct are constructed as follows. For the lower bound of $\Omega\left(\frac{1}{\|\pi\|_2}\right)$ we consider two connected copies of the original graph. Using a coupling we show, by considering the total variation distance, that the resulting graph is indistinguishable from the original graph unless a node is sampled twice ("collision"). For the lower bound of $\Omega\left(d_{\text{avg}}\right)$ we augment the original graph by adding a 3-regular expander to it. This time we use a coupling together with the variation distance to show that it requires $\Omega(d_{\text{avg}})$ queries to sample at least once a node from the 3-regular expander and hence making the graphs indistinguishable if fewer samples are taken.

▶ **Lemma 3.** *Let $G = (V, E)$ be an undirected, connected graph with $|V| = n$ and $|E| \geq n$. Then there exists a graph $\widetilde{G} = (\widetilde{V}, \widetilde{E})$, with $|\widetilde{V}| = 2n$, such that any algorithm given access to either $\mathcal{O}^{\pi}(G)$ or $\mathcal{O}^{\pi}(\widetilde{G})$ with equal probability, cannot distinguish between the two with probability greater than $\frac{2}{3}$, unless it makes at least $\Omega\left(\frac{1}{\|\pi_G\|_2}\right)$ queries. As a consequence, no algorithm can output $\widehat{n}$ satisfying $|\widehat{n} - n^*| < n^*/2$ w.p. at least 2/3, where $n^* = n$ if the chosen graph is G and $2n$ if it is $\widetilde{G}$.*

▶ **Lemma 4.** *Let $G = (V, E)$ be an undirected, connected graph with $|V| = n$ and $|E| \geq n$. Then there exists a graph $\widetilde{G} = (\widetilde{V}, \widetilde{E})$, with $|\widetilde{V}| = 2n$, such that any algorithm given access to either $\mathcal{O}^{\pi}(G)$ or $\mathcal{O}^{\pi}(\widetilde{G})$ with equal probability, cannot distinguish between the two with probability greater than $\frac{2}{3}$, unless it makes at least $\Omega\left(d_{\text{avg}}(G)\right)$ queries. As a consequence, no algorithm can output $\widehat{n}$ satisfying $|\widehat{n} - n^*| < n^*/2$ w.p. at least 2/3, where $n^* = n$ if the oracle chosen corresponds to G and $n^* = 2n$ if it corresponds to $\widetilde{G}$.*

The proof can be found in the full version. As a consequence of Lemma 3 and Lemma 4, we get the following theorem.

▶ **Theorem 5.** *Given an undirected, connected graph $G = (V, E)$, there exist graphs $\widetilde{G}$, $\bar{G}$ with $2|V|$ nodes and a constant $p < 1$, such that any algorithm that is given access to one of three oracles $\mathcal{O}^{\pi}(G)$, $\mathcal{O}^{\pi}(\widetilde{G})$ and $\mathcal{O}^{\pi}(\bar{G})$, chosen with equal probability, requires $\Omega\left(\frac{1}{\|\pi_G\|_2} + d_{\text{avg}}(G)\right)$ queries to distinguish between them with probability at least p. As a consequence, any algorithm that outputs $\widehat{n}$, such that $|\widehat{n} - n^*| < n^*/2$ requires at least $\Omega\left(\frac{1}{\|\pi_G\|_2} + d_{\text{avg}}(G)\right)$ queries, where $n^* = n$ if the graph is G and $n^* = 2n$ if the graph is either $\widetilde{G}$ or $\bar{G}$.*

3.3 Oracle Sampling from the Neighbour Query Model

In this section, we show that with access to the oracle $\mathcal{O}(G)$, any algorithm that predicts the number of nodes in a graph G to within a small constant fraction requires $\Omega\left(\frac{1}{\|\pi_G\|_2} + d_{\text{avg}}\right)$ queries. For proving the lower bounds we use graphs generated according to the *configuration model* [2]. A vector $\mathbf{d} = (d_1, d_2, \ldots, d_n)$ is said to be *graphical* if there exists an undirected graph on n nodes such that vertex $i \in [n]$ has degree d_i. We briefly describe here how graphs are generated in the configuration model:

1. Create disjoint sets W_i, for $i \in \{1, \ldots n\}$, with $|W_i| = d_i$. The elements of W_i are called *stubs*.
2. Create a uniform random maximum matching in the set $\bigcup_{i=1}^{n} W_i$ (note that $\sum_{i=1}^{n} d_i$ must be even since $\mathbf{d}$ is graphical).
3. For a stub edge $\{x, y\}$ in the matching, such that $x \in W_i$ and $y \in W_j$, the edge $\{i, j\}$ is added to the graph.

The above procedure creates a graph where vertex i has degree exactly d_i. However, the graph may not be simple, *i.e.,* it may have multiple edges and self-loops. Also, this procedure does not necessarily produce a uniform distribution over graphs having degree sequence $\mathbf{d}$. The expected number of multi-edges and self-loops in the graph is, for many interesting graph families, only a small fraction and in any graph with bounded degree their expected number is a constant. We use $G \sim \mathcal{G}(\mathbf{d})$ to denote that a graph G was generated in the configuration model with degree sequence $\mathbf{d}$.

Recall the definition of a *sensible* algorithm as one that never makes a query to which it already knows the answer. A *sensible* algorithm has the following behaviour: (i) for every $u \in V$ and $i \leq \deg(u)$ ($\deg^+(u)$ and $\deg^-(u)$, respectively) it makes the query $(\ell(u), i)$ ($(\ell(u), i, \mathsf{in})$ and $(\ell(u), i, \mathsf{out})$, respectively) at most once, and (ii) it never queries a $(\ell(v), i)$ if it has not received $\ell(v)$ as a valid label or if $i \notin [1, \deg(v)]$. Note that there exists for any algorithm, in the above defined query model, a sensible implementation which needs at most as many queries as the original algorithm.[7] Clearly, any sensible algorithm wouldn't query $(\ell(u), j)$ after querying $(\ell(v), i)$. The degree sequences we uses to construct the lower bound graphs are essentially the same as in the stationary query model. This time, however, we are facing changing probability distributions which depend on the random choices revealed so far. The proof of the following lemma can be found in the full version.

▶ **Lemma 6.** *Let $\varepsilon > 0$. Let $\boldsymbol{d} = (d_1, \ldots, d_n)$ be an arbitrary graphical sequence and let $D := \sum_{i=1}^{n} d_i$. Let $G \sim \mathcal{G}(d)$, and let $\widetilde{G}$ be an arbitrary graph with degree sequence $\boldsymbol{d}$. There exists an implementation of an oracle $\mathcal{O}^s(G)$ (with side-information) such that if $(\ell(X_t), \deg_G(X_t))_{t=1}^{T}$ is the sequence of responses to a* sensible *algorithm, where T equals*

$$\min\left\{\min\{\tau \geq 0 \mid \textit{all neighbours of all known nodes are disclosed using } \tau \textit{ queries}\}, \frac{\varepsilon\sqrt{D}}{16}\right\},$$

and if $(\ell(\widetilde{X}_t), \deg_{\widetilde{G}}(\widetilde{X}_t))_{t=1}^{T}$ is the sequence returned by oracle $\mathcal{O}^\pi(\widetilde{G})$, then there exists a coupling so that the sequences $((\ell(X_t), \deg_G(X_t)))_{t=1}^{T}$ and $((\ell(\widetilde{X}_t), \deg_{\widetilde{G}}(\widetilde{X}_t)))_{t=1}^{T}$ are identical with probability at least $1 - \varepsilon$.

[7] For technical reasons, in the proof of Lemma 6, we use an oracle $\mathcal{O}^s$ with *s*ide-information as an extension of $\mathcal{O}$: $\mathcal{O}^s$ returns, upon query, exactly the same information as $\mathcal{O}$, but can add additional truthful information. In particular, we allow the oracle when queried $(\ell(v), i)$ to not only return the corresponding node $(\ell(u), \deg(u))$, where u is the i^{th} element in the adjacency list of v, but also the index, say j, in the adjacency list of u which corresponds to v.

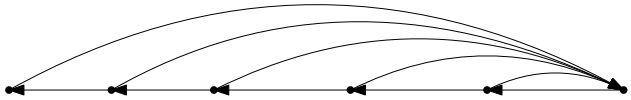


Figure 1 The Line graph on 6 nodes.

The proof of the main theorem of this section can be found in the full version.

▶ **Theorem 7.** *Let* $\boldsymbol{d} = (d_1, \ldots, d_n)$ *be a graphical vector satisfying* $\min_i d_i \geq 3$*. Then there exists a graphical* $\widetilde{\boldsymbol{d}} = (\widetilde{d}_1, \ldots, \widetilde{d}_n, \widetilde{d}_{n+1}, \ldots, \widetilde{d}_{2n})$ *with* $\min_i \widetilde{d}_i \geq 3$*, such that for* $G \sim \mathcal{G}(\boldsymbol{d})$ *and* $\widetilde{G} \sim \mathcal{G}(\widetilde{\boldsymbol{d}})$*, there exists* $c > 0$*, such that any algorithm with access to one of two oracles* $\mathcal{O}(G)$ *or* $\mathcal{O}(\widetilde{G})$ *chosen with equal probability, cannot distinguish between the two with probability greater than* $1 - c$ *unless it makes* $\Omega\left(\frac{1}{\|\pi_G\|_2} + d_{\text{avg}}(G)\right)$ *queries to the oracle.*

4 Directed Graphs

In this section, we consider the query complexity of estimating the number of nodes in directed graphs. We first observe that estimating n using the approach of Katzir *et al.* [13] is not possible since the stationary distribution of a node is in general not proportional to its degree. Another obstacle is that the stationary distribution of a node can be exponentially small as the graphs in Figure 1 and Figure 2 illustrate. In particular, it takes an exponentially large sample drawn from the stationary distribution to distinguish between the line graph of Figure 1 on n nodes and the line graph on $2n$ nodes, since the probability mass of the additional nodes is $2^{-\Omega(n)}$. It is also not very difficult to show that even with access to one of the two oracles $\overrightarrow{\mathcal{O}}$ or $\overrightarrow{\mathcal{O}}(1)$, $\Omega(n)$ queries are required to distinguish the line graph on n vertices from the line graph on $2n$ vertices.

As the example of the line graph reveals, unlike in the undirected case, rapid mixing and low average degree are not sufficient conditions to design a good estimator of the number of nodes using sublinear number of queries. The line graph shows that in the directed case, rapid mixing does not imply short (directed) diameter. One might hope that if one throws small diameter into the mix, in addition to low average degree and rapid mixing, a better estimator could be designed. Below, we show that this is not the case. The problem of estimation remains stubbornly hard, and $\Omega(n)$ queries to the oracle $\overrightarrow{\mathcal{O}}$, and $2^{\Omega(n)}$ queries to the stationary query oracle are required to achieve a good estimate of the number of nodes.

These observations suggest that exploring the graph, *e.g.,* through breadth-first search, is much faster than sampling from the stationary distribution. The question of interest is whether there is a property, satisfied by graphs of interest, which yields a query complexity better than $\Omega(m)$. We answer this positively in Section 4.1, where we introduce a parameter that generalises the conductance ϕ_ε and give almost tight bounds on the number of queries required to estimate n up to an ε relative error. Our Algorithm EdgeSampling takes this parameter as an input and terminates after $O(n/\phi_\varepsilon)$ queries which can be much smaller than the sample complexity of breadth-first search.[8]

[8] The results in Section 3 can be used if access to $\overrightarrow{\mathcal{O}}(2)$ is provided. In this case, we can simply treat the graph as being undirected. However, it is still interesting to understand whether the distinction between in-neighbours and out-neighbours allows one to design better estimators. At present, we are unaware of any graphs where this might be the case.

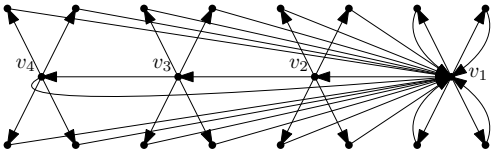


Figure 2 The graph Comet$(20, 4)$.

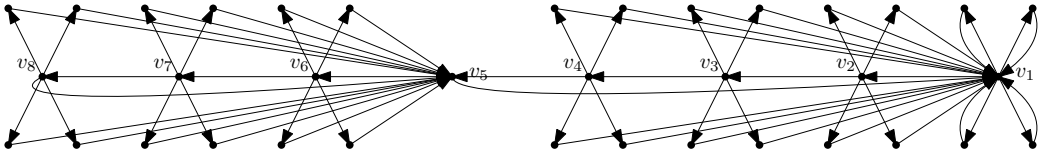


Figure 3 The graph DoubleComet$(40, 8)$. DoubleComet$(2n, 2k)$ consists of two copies of Comet(n, k) connected in the following way. First we remove all directed neighbours of v_{k+1} as well as the edge (v_k, v_1). Then we add (v_k, v_{k+1}), (v_{k+1}, v_{k+2}) as well as (v_{k+1}, v_1).

The Comet Graph

The *Comet graph*, Comet(n, k) is constructed as follows. Assume that k divides n. There is a directed cycle on the vertices $v_1, v_2, \ldots, v_k$, with edges (v_i, v_{i+1}) for $1 \leq i < k$ and (v_k, v_1). We denote these k vertices as *centres*. For every $\ell \in [k]$, there is a directed star $S_\ell = \{(v_\ell, v_{\ell,j}) : j \in [n/k - 1]\}$ with centre in v_ℓ of degree $n/k - 1$. For each leaf $v_{\ell,j}$ in star S_ℓ with $\ell \in [k], j \in [n/k - 1]$, there is a directed edge to the first star centre v_1, that is, $\{(v_{\ell,j}, v_1) : \ell \in [1, k], j \in [n/k - 1]\}$. We write v_ℓ^G and $v_{\ell,j}^G$ to emphasise that the nodes belong to graph G.

▶ **Theorem 8.** *Let n be a multiple of k. Then Comet(n, k) has mixing time $t_{\text{mix}} = O(1)$ and diameter k. Furthermore, any algorithm requires at least $\Omega((\frac{n}{k})^{k-1})$ queries to $\mathcal{O}^\pi$ and $\Omega(n)$ queries to $\overrightarrow{\mathcal{O}}$ to distinguish between $G =$ Comet(n, k) and $\widetilde{G} =$ Comet$(2n, 2k)$.*

The proof can be found in the full version. Note that the above results only apply to the oracles $\overrightarrow{\mathcal{O}}$ and $\mathcal{O}^\pi$, but not to $\overrightarrow{\mathcal{O}}(1)$, since the in-degrees make it easy to distinguish between the two graphs. However, it is straightforward to extend the graph such that the sample complexity remains $\Omega(n)$ even if the in-degrees are known; thus even with access to $\overrightarrow{\mathcal{O}}(1)$, $\Omega(n)$ queries are required.

▶ **Observation 9.** *Let n be a multiple of k. Then DoubleComet$(2n, 2k)$ (defined in Figure 3) has mixing time $t_{\text{mix}} = O(1)$ and diameter $2k$. Furthermore, any algorithm requires at least $\Omega(n)$ queries to $\overrightarrow{\mathcal{O}}(1)$ to distinguish between $G =$ Comet(n, k) and $\widetilde{G} =$ DoubleComet$(2n, 2k)$ on $2n$ nodes.*

4.1 Assuming a Bound on the Connectivity

In this section we introduce the parameter *general conductance*. We first recall some graph notation in the directed setting. Given a non-empty proper subset of vertices $S \subset V$, let $\deg^+(S) = |\{(u, v) \in E : u \in S\}|$ be the out-degree of S. The *cut* of S, ∂S, is the set of edges crossing between S and $V \setminus S$, that is, $\partial S = \{(u, v) \in E : u \in S, v \notin S\}$. The *general conductance* of S, $\phi(S)$, is the ratio between the cut of S, and the out-degree of S. That is, $\phi(S) = |\partial S| / \deg^+(S)$. Given $\varepsilon \in (0, 1)$, the graph ε-general conductance, ϕ_ε, is the minimum of $\phi(S)$ over every non-empty proper subset of V of size at most $(1 - \varepsilon)|V|$, *i.e.*,

$\phi_\varepsilon(G) = \min_{S\subseteq V: 1\leq|S|\leq(1-\varepsilon)|V|} \phi(S)$. Note that the parameter ϕ_ε decreases monotonically as ε decreases. In the undirected setting for $\varepsilon = 1/2$ this is just what is commonly known as the conductance.[9] In the following we describe the algorithm that estimates the graph size.

Upper bound in terms of the general conductance

We consider algorithm EdgeSampling for estimating the number of nodes. The algorithm takes as input the parameter ϕ, a lower bound on the general conductance ϕ_ε. The query complexity is $O(n/\phi_\varepsilon)$ and the output estimate $\widehat{n}$ satisfies $(1-\varepsilon)n \leq \widehat{n} \leq n$ with arbitrary confidence controlled by an input parameter ℓ. Observe that $O(n/\phi_\varepsilon)$ can be much smaller than the run time of breadth-first search $\Omega(m)$.

Algorithm overview. The algorithm works as follows. At each time step the algorithm maintains a counter Y. If at some point the counter exceeds the threshold ℓ, then the algorithm terminates. The algorithm divides the queries into blocks of length at most $2/\phi$ corresponding to the execution of the **for** loop. In each block, at every step the algorithm samples one outgoing edge uniformly at random from those available and not queried before. If at any step a new node is disclosed, then this finishes the block (break of the **for** loop) and the counter Y is decreased by 1. If the block finishes without finding a new node, then the counter is increased by 1. Once the counter reaches ℓ, which will happen eventually, then the algorithm outputs the number of nodes it discovered. Even though our goal is to minimise the query complexity, it is worth noticing that the time and space complexity can be kept low by choice of suitable data structures. Although the oracle returns labels of nodes, we use nodes and their labels interchangeably in the algorithm and the analysis of Theorem 10. The proof can be found in the full version.

▶ **Theorem 10.** *Algorithm* EdgeSampling$(\overrightarrow{\mathcal{O}}, \ell, \phi)$ *on graph G has a query complexity of* $\min\{2(2n+\ell)/\phi, m\}$ *and outputs an estimate $\widehat{n} \leq n$. Furthermore, if G has general conductance $\phi_\varepsilon(G)$ of at least ϕ, then the algorithm satisfies $\widehat{n} \geq (1-\varepsilon)n$ w.p. at least $1-2^{-\ell}$.*

▶ **Observation 11.** *The error made by Algorithm* EdgeSampling *is one-sided – the estimate never exceeds n. Allowing a two-sided error and given ε, one can instead output an estimate with smaller additive error. Consider a modification of Algorithm* EdgeSampling$(\overrightarrow{\mathcal{O}}, \ell, \phi)$ *on graph G which takes the additional parameter ε and outputs $\widehat{n}^* := |S_t|(1+\frac{\varepsilon}{2-\varepsilon})$ instead of $|S_t|$. If G has general conductance $\phi_\varepsilon(G)$ of at least ϕ, then $\widehat{n}^*$ satisfies $|n-\widehat{n}^*| \leq \frac{\varepsilon}{2-\varepsilon}n$ w.p. at least $1-2^{-\ell}$. The poof can be found in the full version article.*

Lower bound in terms of the general conductance

In the following we show that the bound of Observation 11 is almost tight. Recall that, given ϕ_ε, the modified version of Algorithm EdgeSampling in Observation 11 returns an estimate with and additive error of at most $\frac{\varepsilon}{2-\varepsilon}n$ using $O(n/\phi_\varepsilon)$ queries. In what follows we show that any algorithm, given the values ϕ_ε and ε, cannot output an estimate with an error smaller

[9] The definition of conductance in the directed setting is more involved and more importantly doesn't seem be directly relevant to the question of estimating the number of nodes. It suffers from similar problems as the skewed stationary distributions. Graphs having poor connectivity to a large fraction of the nodes may still have very *good* conductance if the total mass of the poorly connected nodes under the stationary distribution is very small.

Algorithm 1 EdgeSampling($\overrightarrow{\mathcal{O}}, \ell, \phi$).

$Y_0 = 0$ (fail surplus counter)
$v = (node)\ \overrightarrow{\mathcal{O}}.\mathsf{init}$ (query oracle to get the initial node)
$S_0 = \{v\}$
$E_0 = \{(v,i) \mid i \leq \deg^+(v)\}$ (set of undisclosed edges)
$t = 1$
while $Y_t \leq \ell$ **do**
  for $\tau = 1$ **to** $2/\phi$ **do**
    choose (u,i) uniformly at random from $E_{(t-1)\cdot 2/\phi+\tau-1}$.
    $v = (node)\ \overrightarrow{\mathcal{O}}.(u,i,\mathsf{out})$
    if $v \notin S_{t-1}$ **then**
      $S_t = S_{t-1} \cup \{v\}$
      $E_{(t-1)\cdot 2/\phi+\tau} \leftarrow (E_{(t-1)\cdot 2/\phi+\tau-1} \cup \{(v,i) \mid i \leq \deg^+(v)\}) \setminus \{(u,i)\}$
      break
    else
      $E_{t\cdot 2/\phi} \leftarrow E_{t\cdot 2/\phi-1} \setminus \{(u,i)\}$
  if $|S_t| = |S_{t-1}| + 1$ **then**
    $Y_t \leftarrow Y_t - 1$
  else
    $Y_t \leftarrow Y_t + 1$
    $S_t \leftarrow S_{t-1}$
  $t \leftarrow t + 1$
Output $|S_t|$.

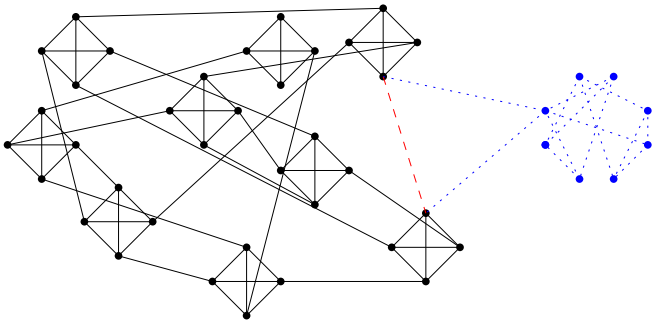


Figure 4 The graphs of Theorem 12. G contains the black nodes and the black and red (dashed) edges which form a $\lceil 1/\phi \rceil$-regular expander on cliques of size $\lceil 1/\phi \rceil$. G' is obtained by removing the red (dashed) edge and adding the blue (dotted) graph. At least one blue edge needs to be sampled, which takes $\Omega(n/\phi)$ time, in order to estimate n accurately.

than $\frac{\varepsilon-\delta}{2-\varepsilon-\delta}n$ unless it makes $\Omega(n/\phi_\varepsilon)$ queries, for any $\delta < \varepsilon/2$. We prove the following theorem for undirected graphs using $\mathcal{O}$, but it should be clear that the same result holds for directed graphs, by making the graph directed, with symmetric edges, and using $\overrightarrow{\mathcal{O}}(2)$ (and hence also for oracles $\overrightarrow{\mathcal{O}}$ and $\overrightarrow{\mathcal{O}}(1)$).

▶ **Theorem 12.** *Let $n \in \mathbb{N}$, $\phi \in [1/n, 1]$ and $\varepsilon \in (0, 1/2]$. There exists an undirected graph with general conductance $\phi_\varepsilon = \Theta(\phi)$ such that any algorithm with access to $\mathcal{O}$ requires $\Omega(n/\phi_\varepsilon)$ queries to output $\widehat{n}$ such that $|n - \widehat{n}| \leq \frac{\varepsilon-\delta}{2-\varepsilon-\delta}n$ w.p. at least $2/3$ for any $\delta < \varepsilon/2$.*

References

1 Michael A. Bender and Data Ron. Testing properties of directed graphs: Acyclicity and connectivity. *Random Structures and Algorithms*, 20(2):184–205, 2002.

2 Béla Bollobás. A probabilistic proof of an asymptotic formula for the number of labelled regular graphs. *European Journal of Combinatorics*, 1(4):311–316, 1980. doi:10.1016/S0195-6698(80)80030-8.

3 Mickey Brautbar and Michael Kearns. Local algorithms for finding interesting individuals in large networks. In *In Proceedings of ICS 2010*, pages 188–199, 2010.

4 Colin Cooper and Alan Frieze. Crawling on web graphs. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 419–427. ACM, 2002.

5 Colin Cooper, Tomasz Radzik, and Yiannis Siantos. Estimating network parameters using random walks. In *Computational Aspects of Social Networks, 4th International Conference on*, pages 33–40, 2012.

6 Artur Czumaj, Pan Peng, and Christian Sohler. Relating two property testing models for bounded degree directed graphs. In *Proceedings of the ACM Symposium on the Theory of Computing (STOC)*, 2016.

7 Anirban Dasgupta, Ravi Kumar, and Tamás Sarlós. On estimating the average degree. In *Proceedings of the 23rd international conference on World Wide Web*, pages 795–806. ACM, 2014. doi:10.1145/2566486.2568019.

8 Mark Finkelstein, Howard G Tucker, and Jerry Alan Veeh. Confidence intervals for the number of unseen types. *Statistics & Probability Letters*, 37(4):423–430, 1998.

9 Oded Goldreich. Introduction to testing graph properties. Survey article available at http://www.wisdom.weizmann.ac.il/~oded/COL/tgp-intro.pdf, 2010.

10 I. J. Good. The population frequencies of species and the estimation of population parameters. *Biometrika*, 40(3/4):237–264, 1953.

11 Varun Kanade, Frederik Mallmann-Trenn, and Victor Verdugo. How large is your graph? *CoRR*, abs/1702.03959, 2017. URL: http://arxiv.org/abs/1702.03959.

12 Liran Katzir and Stephen J. Hardiman. Estimating clustering coefficients and size of social networks via random walk. *ACM Transactions on the Web*, 9(4):19:1–19:20, 2015.

13 Liran Katzir, Edo Liberty, Oren Somekh, and Ioana A. Cosma. Estimating sizes of social networks via biased sampling. *Internet Mathematics*, 10(3-4):335–359, 2014. doi:10.1080/15427951.2013.862883.

14 Donald E. Knuth. Estimating the efficiency of backtrack programs. *Mathematics of computation*, 29(129):122–136, 1975.

15 Alberto Marchetti-Spaccamela. On the estimate of the size of a directed graph. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 317–326. Springer, 1988.

16 Milena Mihail, Amin Saberi, and Prasad Tetali. Random walks with lookahead on power law random graphs. *Internet Mathematics*, 3(2):147–152, 2006.

17 Cameron Musco, Hsin-Hao Su, and Nancy A. Lynch. Ant-inspired density estimation via random walks: Extended abstract. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, pages 469–478, 2016.

18 Leonard Pitt. A note on extending knuth's tree estimator to directed acyclic graphs. *Information processing letters*, 24(3):203–206, 1987.

19 Gregory Valiant and Paul Valiant. Estimating the unseen: an n/log(n)-sample estimator for entropy and support size, shown optimal via new clts. In *Proceedings of the forty-third annual ACM symposium on Theory of computing*, pages 685–694. ACM, 2011.

20 Gregory Valiant and Paul Valiant. Estimating the unseen: improved estimators for entropy and other properties. In *Advances in Neural Information Processing Systems*, pages 2157–2165, 2013.

Tight Bounds for Connectivity and Set Agreement in Byzantine Synchronous Systems

Hammurabi Mendes[1] and Maurice Herlihy[2]

1 **Mathematics and Computer Science Department, Davidson College, Davidson, NC, USA**
hamendes@davidson.edu

2 **Computer Science Department, Brown University, Providence, RI, USA**
mph@cs.brown.edu

Abstract

In this paper, we show that the protocol complex of a Byzantine synchronous system can remain $(k-1)$-connected for up to $\lceil t/k \rceil$ rounds, where t is the maximum number of Byzantine processes, and $t \geq k \geq 1$. This topological property implies that $\lceil t/k \rceil + 1$ rounds are necessary to solve k-set agreement in Byzantine synchronous systems, compared to $\lfloor t/k \rfloor + 1$ rounds in synchronous crash-failure systems. We also show that our connectivity bound is tight as we indicate solutions to Byzantine k-set agreement in exactly $\lceil t/k \rceil + 1$ synchronous rounds, at least when n is suitably large compared to t. In conclusion, we see how Byzantine failures can potentially require one extra round to solve k-set agreement, and, for n suitably large compared to t, at most that.

1998 ACM Subject Classification F.1.1 Models of Computation

Keywords and phrases Byzantine, synchronous, k-set agreement, topology, connectivity

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.35

1 Introduction

A *task* is a distributed coordination problem where multiple processes start with private inputs, communicate among themselves (by shared memory or message passing), and halt with outputs consistent with the task specification. There are *crash-failure* systems [1], where processes can fail only by permanent, unannounced halting, or *Byzantine-failure* systems [18], where processes can fail arbitrarily, even maliciously. Communication among processes can be *synchronous* or *asynchronous*. In *synchronous* systems, communication and computation are organized in discrete rounds. In each round, each non-faulty process performs as follows, in order:

(i) sends a message;
(ii) receives all messages sent in the current round by the other processes; and
(iii) performs internal computation.

In *asynchronous* systems, processes may have different relative speeds, and communication is subject to unbound, finite delays.

The problem of consensus in the synchronous Byzantine message-passing model was among the earliest to be investigated, and upper and lower consensus bounds in that model are well-understood. In this paper, we turn our attention to the bounds for problems such as k-set agreement, using concepts and techniques adapted from combinatorial topology. We can capture all possible information dissemination patterns permitted by this model in a single combinatorial structure called a *simplicial complex* (or just *complex*). A classical topological property of a simplicial complex is its level of *connectivity*, which is, roughly speaking, the


© Hammurabi Mendes and Maurice Herlihy;
licensed under Creative Commons License CC-BY

dimension below which it has no holes. Many classical proofs of consensus impossibility can be reformulated as showing that certain complexes are 0-connected (also called *path-connected*), and all known impossibility proofs for k-set agreement rely on showing that certain complexes are $(k-1)$-connected. Very informally, the higher the degree of connectivity imposed by the adversary, the weaker the model's computational power. Here, we present the first tight bounds on connectivity for the synchronous Byzantine message-passing model.

Prior work using topological techniques is discussed in Section 2. Our operational setting is detailed in Section 3, and our topological model is formalized in Section 4.

Our **first contribution** comes in Section 5. We show that, in a Byzantine synchronous system, the protocol complex can remain $(k-1)$-*connected* for $\lceil t/k \rceil$ rounds, where t is an upper bound on the number of Byzantine processes. Perhaps surprisingly, this is only *one* more round than the upper bound for crash-failure systems ($\lfloor t/k \rfloor$, shown in [8]). In order to show that, as part of our **second contribution**, we conceive a combinatorial operator modeling the ability of Byzantine processes to *equivocate* – that is, to transmit ambiguous state information – without revealing their Byzantine nature. We compose this operator with regular crash-failure operators, extending the protocol complex connectivity for one extra round. As noted before, connectivity is of interest because a $(k-1)$-connected protocol complex prevents important problems such as k-set agreement [7, 9] from having solutions.

Our **third contribution** comes in Section 6. We show that the above connectivity bound is *tight* in certain settings (described in Section 6), by solving k-set agreement in $\lceil t/k \rceil + 1$ rounds. We do so with a full-information protocol that assumes n suitably large compared to t. The protocol suits well our purpose of tightening the $\lceil t/k \rceil$ bound, and also exposes clearly *the reason why* $\lceil t/k \rceil + 1$ rounds is enough to solve k-set agreement.

These results give new insight into the power of Byzantine adversaries for problems beyond consensus. Although Byzantine adversaries seem much more powerful than crash-failure ones, we show that a Byzantine adversary can impose at most *one* additional synchronous round beyond that imposed by a crash-failure adversary. In terms of solvability vs. number of rounds, the penalty for moving from crash to Byzantine failures, captured by $(k-1)$-connectivity in the protocol complex, can be *quite limited* in synchronous systems, particularly when n is relatively large compared to t.

2 Related Work

The Byzantine failure model was initially introduced by Lamport, Shostak, and Pease [18]. The use of simplicial complexes to model distributed computations was introduced by Herlihy and Shavit [15]. The asynchronous computability theorem for general tasks in [16] details the approach for asynchronous wait-free computation in the crash-failure model. This model was recently generalized by Gafni, Kuznetsov, and Manolescu [10]. Computability in Byzantine asynchronous systems, where tasks are constrained in terms of non-faulty inputs, was recently considered in [19].

The k-set agreement problem was originally defined by Chaudhuri [7]. Alternative formulations with different validity notions, or failure/communication settings, are discussed in [22, 9]. A full characterization of optimal translations between different failure settings is given in [2, 23], which requires different number of rounds depending on the relation between the number of faulty processes, and the number of participating processes.

The relationship between connectivity and the impossibility of k-set agreement is described explicitly or implicitly in [8, 16, 24]. Recent work by Castañeda, Gonczarowski, and Moses [6] considers an issue of chains of hidden values, a concept loosely explored here. The approach

based on shellability and layered executions for lower bounds in connectivity has been used by Herlihy, Rajsbaum, and Tuttle [14, 13, 12], assuming crash-failure systems, synchronous or asynchronous.

3 Operational Model

We have $n+1$ processes[1] $\mathbb{P} = \{P_0, \ldots, P_n\}$ communicating by message-passing via pairwise, reliable channels (*authenticated channels* in the literature [5]). Technically, all transmitted messages are delivered uniquely, and with sender reliably identified.

At most t processes are *faulty* or *Byzantine* [18], and may display arbitrary, even malicious behavior, at any point in the execution. The actual behavior of Byzantine processes is defined by an *adversary*. Byzantine processes may execute the protocol correctly or incorrectly, at the discretion of the adversary. Processes that perform internal state transitions and message exchanges in strict accordance to the protocol for rounds 1 up to some r (inclusive) are called *non-faulty processes up to round* r, and are denoted by $\mathbb{G}^r$. Also, *faulty processes up to round* r are denoted by $\mathbb{B}^r = \mathbb{P} \setminus \mathbb{G}^r$. A non-faulty process up to any round $r \geq 1$ is called simply *non-faulty* or *correct*, which we denote by $\mathbb{G}$.

We model processes as state machines. The input value (resp. output value) of a non-faulty process P_i is written I_i (resp. O_i). Byzantine processes may have *apparent* inputs, denoted as above, and defined as one of the valid input values transmitted to other processes in the first round of computation. Each non-faulty process P_i has an internal state called *view*, which we denote by $\text{view}(P_i)$. In the beginning of the protocol, $\text{view}(P_i)$ is I_i. At any round r, any non-faulty process:
(1) sends its internal state to all other processes;
(2) receives the state information from other processes;
(3) concatenates that information to its own internal state.
After completing some number of iterations, each process applies a decision function δ to its current state in order to decide O_i. Thus, we assume that processes follow a *full-information* protocol [13].

For simplicity of notation, we define a round 0 where processes are simply assigned their inputs. Without losing generality, all processes are assumed non-faulty up to round 0: $\mathbb{G}^0 = \mathbb{P}$ and $\mathbb{B}^0 = \emptyset$. For any round $r \geq 0$, a *global state up to round* r formally specifies:
(1) the non-faulty processes up to round r; and
(2) the view of all non-faulty processes up to round r.

4 Topological Model

We now sketch the required concepts from combinatorial topology. For details, please refer to Munkres [20], Kozlov [17], or Herlihy *et al.* [11].

4.1 Basics

A *simplicial complex* $\mathcal{K}$ consists of a finite set V along with a collection of subsets of V closed under containment. An element of V is called a *vertex* of $\mathcal{K}$. The set of vertices of $\mathcal{K}$ is referred by $V(\mathcal{K})$. Each set in $\mathcal{K}$ is called a *simplex*, usually denoted by lower-case Greek letters: σ, τ, etc. The *dimension* $\dim(\sigma)$ of a simplex σ is $|\sigma| - 1$.

[1] Choosing $n+1$ processes rather than n simplifies the topological notation, but slightly complicates the computing notation. Choosing n processes has the opposite trade-off. We choose $n+1$ for compatibility with prior work.

A subset of a simplex is called a *face*. The collection of faces of σ with dimension exactly x is called $\text{Faces}^x(\sigma)$. A face τ of σ is called *proper* if $\dim(\tau) = \dim(\sigma) - 1$. We use "$k$-simplex" as shorthand for "k-dimensional simplex", analogously in "k-face." The dimension $\dim(\mathcal{K})$ of a complex is the maximal dimension of its simplexes, and a *facet* of $\mathcal{K}$ is any simplex having maximal dimension in $\mathcal{K}$. A complex is said *pure* if all facets have dimension $\dim(\mathcal{K})$. In a pure complex, we define the *codimension* of σ in $\mathcal{K}$, denoted $\text{codim}_{\mathcal{K}}(\sigma)$, as $\dim(\mathcal{K}) - \dim(\sigma)$. The set of simplexes of $\mathcal{K}$ having dimension at most ℓ is a subcomplex of $\mathcal{K}$, which is called *ℓ-skeleton* of $\mathcal{K}$, denoted by $\text{skel}^{\ell}(\mathcal{K})$.

4.2 Maps

Let $\mathcal{K}$ and $\mathcal{L}$ be complexes. A *vertex map* f carries vertices of $\mathcal{K}$ to vertices of $\mathcal{L}$. If f additionally carries simplexes of $\mathcal{K}$ to simplexes of $\mathcal{L}$, it is called a *simplicial* map. A *carrier map* Φ from $\mathcal{K}$ to $\mathcal{L}$ takes each simplex $\sigma \in \mathcal{K}$ to a subcomplex $\Phi(\sigma) \subseteq \mathcal{L}$, such that for all $\sigma, \tau \in \mathcal{K}$, we have $\Phi(\sigma \cap \tau) \subseteq \Phi(\sigma) \cap \Phi(\tau)$. If additionally $\Phi(\sigma \cap \tau) = \Phi(\sigma) \cap \Phi(\tau)$, we say that the carrier map is *strict*. A simplicial map $\phi : \mathcal{K} \to \mathcal{L}$ is *carried by the carrier map* $\Phi : \mathcal{K} \to 2^{\mathcal{L}}$ if, for every simplex $\sigma \in \mathcal{K}$, we have $\phi(\sigma) \subseteq \Phi(\sigma)$.

Although we defined simplexes and complexes in a purely combinatorial way, they can also be interpreted geometrically. An n-simplex can be identified with the convex hull of $(n+1)$ affinely-independent points in the Euclidean space of appropriate dimension. This geometric realization can be extended to complexes. The point-set that underlies such *geometric complex* $\mathcal{K}$ is called the *polyhedron* of $\mathcal{K}$, denoted by $|\mathcal{K}|$. For any simplex σ, the *boundary* of σ, which we denote $\partial\,\sigma$, is the simplicial complex of $(\dim(\sigma) - 1)$-faces of σ. The *interior* of σ is defined as $\text{Int}\,\sigma = |\sigma| \setminus |\,\partial\,\sigma|$.

We can define simplicial/carrier maps between geometrical complexes. Given a simplicial map $\phi : \mathcal{K} \to \mathcal{L}$ (resp. carrier map $\Phi : \mathcal{K} \to 2^{\mathcal{L}}$), the polyhedrons of every simplex in $\mathcal{K}$ and $\mathcal{L}$ induce a continuous simplicial map $\phi_c : |\mathcal{K}| \to |\mathcal{L}|$ (resp. continuous carrier map $\Phi_c : |\mathcal{K}| \to |2^{\mathcal{L}}|$). We say ϕ (resp. ϕ_c) is carried by Φ if, for any $\sigma \in \mathcal{K}$, we have $|\phi(\sigma)| \subseteq |\Phi(\sigma)|$ (resp. $\phi_c(|\sigma|) \subseteq \Phi_c(|\sigma|)$).

4.3 Connectivity

In light of topology, two geometrical objects A and B are *homeomorphic* if, there is a bicontinuous map from A into B. In other words, there exists a continuous map between those objects, with a continuous inverse [21, 20].

▶ **Fact 1.** *[20] For any k-simplex σ, the boundary of σ is homeomorphic to a $(k-1)$-sphere, and σ is homeomorphic to a k-disk.*

We say that a simplicial complex $\mathcal{K}$ is *x-connected*, $x \geq 0$, if every continuous map of a subset of $|\mathcal{K}|$ homeomorphic to an x-sphere in $|\mathcal{K}|$ can be extended into a subset of $|\mathcal{K}|$ homeomorphic to an $(x+1)$-disk in $|\mathcal{K}|$. In analogy, think of the extremes of a pencil as a 0-disk, and the pencil itself as a 1-sphere (the extension is possible if 0-connected); the rim of a coin as a 1-sphere, and the coin itself as a 2-disk (the extension is possible if 1-connected); the outer layer of a billiard ball as a 2-sphere, and the billiard ball itself as a 3-disk (the extension is possible if 2-connected). For us, (-1)-connected is understood as *non-empty*, and (-2)-connected or lower imposes no restriction.

4.4 Pseudospheres & Shellability

▶ **Definition 2.** Let $\mathbb{S} = \{(P_i, S_i) : P_i \in \mathbb{P}'\}$, where each S_i is an arbitrary set and $\mathbb{P}' \subseteq \mathbb{P}$. A *pseudosphere* $\Psi(\mathbb{P}', \mathbb{S})$ is a simplicial complex where $\sigma \in \Psi(\mathbb{P}', \mathbb{S})$ if $\sigma = \{(P_i, V_i) : P_i \in \mathbb{P}', V_i \in S_i\}$.

Essentially, a pseudosphere is a simplicial complex formed by independently assigning values to all the specified processes. If $S_i = S$ for all $P_i \in \mathbb{P}'$, we simply write $\Psi(\mathbb{P}', S)$.

▶ **Definition 3.** A pure, simplicial complex $\mathcal{K}$ is *shellable* if we can arrange the facets of $\mathcal{K}$ in a linear order $\phi_0 \ldots, \phi_t$ such that $\left(\bigcup_{0 \leq i < k} \phi_i\right) \cap \phi_k$ is a pure $(\dim(\phi_k) - 1)$-dimensional simplicial complex for all $0 < k \leq t$. We call the above linear order $\phi_0, \ldots, \phi_t$ a *shelling order*.

Intuitively, a simplicial complex is shellable if it can be built by gluing its x-simplexes along their $(x-1)$ faces only, where x is the dimension of the complex. Note that $\phi_0, \ldots, \phi_t$ is a shelling order if any $\phi_i \cap \phi_j$ $(0 \leq i < j \leq t)$ is contained in a $(\dim(\phi_k) - 1)$-face of ϕ_k $(0 \leq k < j)$. Hence,

$$\text{for any } i < j \text{ exists } k < j \text{ where } (\phi_i \cap \phi_j) \subseteq (\phi_k \cap \phi_j) \text{ and } |\phi_j \setminus \phi_k| = 1. \tag{1}$$

Shellability and pseudospheres are important tools to characterize connectivity in simplicial complexes. The following lemmas are proved in [12] and [11] (pp. 252–253).

▶ **Lemma 4.** *Any pseudosphere $\phi(\mathbb{P}', \mathbb{S})$ is shellable, considering arbitrary $\mathbb{S} = \{(P_i, S_i) : \forall P_i \in \mathbb{P}'\}$.*

▶ **Lemma 5.** *For any $k \geq 1$, if the simplicial complex $\mathcal{K}$ is shellable and $\dim(\mathcal{K}) \geq k$ then $\mathcal{K}$ is $(k-1)$-connected.*

4.5 Nerve Theorem

Let $\mathcal{K}$ be a simplicial complex with a *cover* $\{\mathcal{K}_i : i \in I\} = \mathcal{K}$, where I is a finite index set. The *nerve* $\mathcal{N}(\{\mathcal{K}_i : i \in I\})$ is the simplicial complex with vertexes I and simplexes $J \subseteq I$ whenever $\mathcal{K}_J = \bigcap_{j \in J} \mathcal{K}_j \neq \emptyset$. We can characterize the connectivity of $\mathcal{K}$ in terms of the connectivity of the intuitively simpler nerve of $\mathcal{K}$ with the next theorem.

▶ **Theorem 6** (Nerve Theorem [17, 3])**.** *If for any $J \subseteq I$ denoting a simplex of $\mathcal{N}(\{\mathcal{K}_i : i \in I\})$ (thus, $\mathcal{K}_J \neq \emptyset$) we have that $\mathcal{K}_J$ is $(k - |J| + 1)$-connected, then $\mathcal{K}$ is k-connected if and only if $\mathcal{N}(\{\mathcal{K}_i : i \in I\})$ is k-connected.*

4.6 Protocol Complexes

We represent the evolution of the global state of the system throughout the rounds by simplicial complexes that we call *protocol complexes*. The first, round-0 protocol complex $\mathcal{K}_0$, represents the possible inputs attributed to processes. After each round r, the round-r protocol complex $\mathcal{K}^r$ represents all possible global states of the system at round r. We also call $\mathcal{K}_0$ the *input complex*, also denoted $\mathcal{I}$.

▶ **Definition 7.** For $r \geq 0$, a *name-view* simplex σ is such that:

1. $\sigma = \{(P_i, \text{view}^r(P_i)) : \forall P_i \in \mathbb{G}^r\}$, where $\text{view}^r(P_i)$ denotes P_i's view at round r; and
2. if $(P_i, \text{view}^r(P_i))$ and $(P_j, \text{view}^r(P_j))$ are both in σ, then $P_i \neq P_j$.

Unless otherwise noted, all of our simplicial and carrier maps f are such that $\text{names}(\sigma) = \text{names}(f(\sigma))$, that is, they map between vertices associated with the same processes.

▶ **Definition 8.** For any name-view simplex σ, define
1. $\text{names}(\sigma) = \{P_i : \exists V \text{ such that } (P_i, V) \in \sigma\}$; and
2. $\text{views}(\sigma) = \{V_i : \exists P \text{ such that } (P, V_i) \in \sigma\}$.

The round-0 protocol complex $\mathcal{K}^0$ has name-view n-simplexes $\sigma_I = \{(P_i, I_i) : \forall P_i \in \mathbb{G}^0\}$, representing all the possible process inputs in the beginning of the protocol. The round-r protocol complex $\mathcal{K}^r$, for any $r \geq 0$, is defined as follows: if $\sigma \in \mathcal{K}^r$, then $\sigma = \{(P_i, \text{view}^r(P_i)) : \forall P_i \in \mathbb{G}^r\}$, representing a possible global state of the system for round r.

5 Connectivity Upper Bound

Informally, if the adversary displays Byzantine behavior early in the execution, then in a synchronous, full-information protocol, subsequent communication among the non-faulty processes can reveal the identities of the Byzantine processes, using simple techniques inspired from [2, 4, 25]. Instead, it behooves the adversary to postpone malicious behavior to the very last round, where it cannot be detected.

Say that non-faulty processes start the computation with inputs in $V = \{v_0, \ldots, v_d\}$, *arbitrarily* assigned, with some $d \geq k$ and $t \geq k \geq 1$. To prove our upper bound, we show how the adversary can impose a particular *admissible* execution that preserves high connectivity in the protocol complex: by admissible, we mean an execution where at most t processes fail, with other processes behaving in accordance with the protocol.

Let $r = \lfloor t/k \rfloor$ and $m = t \bmod k$. We have r *crash rounds*, where in each round k processes fail by crashing, but display no Byzantine behavior. If $m > 0$, we have an extra *equivocation round*, where a single Byzantine process sends different views to different processes, causing extra confusion. This round-by-round execution produces a sequence of protocol complexes $\mathcal{K}^0, \ldots, \mathcal{K}^{r+1}$, related by carrier maps $\mathcal{C}^i : \mathcal{K}^{i-1} \to 2^{\mathcal{K}^i}$, for $1 \leq i \leq r$, and $\mathcal{E} : \mathcal{K}^r \to 2^{\mathcal{K}^{r+1}}$.

$$\mathcal{K}^0 \xrightarrow{\mathcal{C}^1} \mathcal{K}^1 \ldots \xrightarrow{\mathcal{C}^r} \mathcal{K}^r \underbrace{\xrightarrow{\mathcal{E}} \mathcal{K}^{r+1}}_{\text{only if } m > 0}. \tag{2}$$

5.1 A Quick Background Detour: The Tools of the Trade

In each of the first r rounds, exactly k processes are failed by the adversary. The crash-failure carrier maps are defined as follows [12, 11]:

▶ **Definition 9.** For any $1 \leq i \leq r$, the crash-failure operator $\mathcal{C}^i : \mathcal{K}^{i-1} \to 2^{\mathcal{K}^i}$ is such that

$$\mathcal{C}^i(\sigma) = \bigcup_{\tau \in \text{Faces}^{n-ik}(\sigma)} \Psi(\text{names}(\tau); [\tau : \sigma]) \tag{3}$$

for any $\sigma \in \mathcal{K}^{i-1}$, with $[\tau : \sigma]$ denoting the set of simplexes μ where $\tau \subseteq \mu \subseteq \sigma$.

▶ **Definition 10.** A *q-connected* carrier map $\Phi : \mathcal{K} \to 2^{\mathcal{L}}$ is a strict carrier map such that, for all $\sigma \in \mathcal{K}$, $\dim(\Phi(\sigma)) > q - \text{codim}_{\mathcal{K}}(\sigma)$ and $\Phi(\sigma)$ is $(q - \text{codim}_{\mathcal{K}}(\sigma))$-connected.

▶ **Definition 11.** A *q-shellable* carrier map $\Phi : \mathcal{K} \to 2^{\mathcal{L}}$ is a strict carrier map such that, for all $\sigma \in \mathcal{K}$, $\dim(\Phi(\sigma)) > q - \text{codim}_{\mathcal{K}}(\sigma)$ and $\Phi(\sigma)$ is shellable.

After r rounds, note that $\mathcal{K}^r$ only contains simplexes with dimension exactly $n - rk$. In [12, 11], the following lemmas are proved:

▶ **Lemma 12.** *For $1 \leq i \leq r$, the operator $\mathcal{C}^i : \mathcal{K}^{i-1} \to 2^{\mathcal{K}^i}$ is a $(k-1)$-shellable carrier map.*

▶ **Lemma 13.** *If $\mathcal{M}^1, \ldots, \mathcal{M}^x$ are all q-shellable carrier maps, and $\mathcal{M}^{x+1}$ is a q-connected carrier map, the composition $\mathcal{M}^1 \circ \ldots \mathcal{M}^x \circ \mathcal{M}^{x+1}$ is a q-connected carrier map, for any $x \geq 0$.*

5.2 Byzantine Systems: Equivocation and Interpretation

After the crash-failure rounds, if $m > 0$ the adversary picks one of the remaining processes to behave maliciously at round $r+1$. This process, say P_b, may send different views to different processes (which is technically called *equivocation*), but, informally speaking, all views are "plausible." For example, two non-faulty processes P_i and P_j could be indecisive after round r on whether the global state is σ_1 or σ_2 in $\mathcal{K}^r$, while P_b, a Byzantine process, sends a state corresponding to σ_1 to P_i, and a state corresponding to σ_2 to P_j. The faulty process P_b *does not reveal* its Byzantine nature, yet it *promotes ambiguity* in the state information diffusion.

At the final round, when a non-faulty process receives the states sent from the other processes, it must decide correctly even if one other process equivocates. If the non-faulty process can receive simplexes σ_1 and σ_2, representing global states that differ in only one process's contribution (that is, $\dim(\sigma_1 \cap \sigma_2) = n - rk - 1$), then the *interpretation* of a message containing one such state must be the same as a message containing the other. We capture this notion using the *equivocation* operator, called $\mathcal{E}$, describing the behavior of a Byzantine process, coupled with an *interpretation* operator, called Interp, describing the required behavior of non-faulty processes. Informally, $\text{Interp}(\sigma_1) = \text{Interp}(\sigma_2)$ for processes in $\text{names}(\tau)$, where $\tau = \sigma_1 \cap \sigma_2$ with $\dim(\tau) = n - rk - 1$. Formally:

▶ **Definition 14.** For arbitrary simplexes σ_1 and σ_2 in $\mathcal{K}$, with $\dim(\mathcal{K}) = n - rk$, let $(P_i, \text{Interp}(\sigma_1)) = (P_i, \text{Interp}(\sigma_2))$ if and only if $\sigma_1 = \sigma_2$; **or** $P_i \in \text{names}(\tau)$ where $\tau = \sigma_1 \cap \sigma_2$ and $\dim(\tau) = n - rk - 1$.

▶ **Definition 15.** For any pure simplicial complexes $\mathcal{K}$ and $\mathcal{L}$ with $\dim(\mathcal{K}) \leq n - rk$ and $\mathcal{K} \supseteq \mathcal{L}$, the $\mathcal{K}$-equivocation operator $\mathcal{E}_\mathcal{K}$ is

$$\mathcal{E}_\mathcal{K}(\mathcal{L}) = \bigcup_{\tau \in \text{Faces}^{n-rk-1}(\mathcal{L})} \Psi(\text{names}(\tau); \{\text{Interp}(\sigma^*) : \sigma^* \in \mathcal{K}, \sigma^* \supset \tau\})). \tag{4}$$

Note that $\mathcal{E}_\mathcal{K}(\mathcal{L}) = \emptyset$ whenever $\dim(\mathcal{L}) < n - rk - 1$ or $\dim(\mathcal{K}) < n - rk$, and also that

$$\mathcal{E}_\mathcal{K}(\sigma) = \bigcup_{\tau \in \text{Faces}^{n-rk-1}(\sigma)} \Psi(\text{names}(\tau); \text{Interp}(\sigma)) \tag{5}$$

for any $\sigma \in \mathcal{K}$ with $\dim(\sigma) = n - rk$. For convenience of notation, define $\mathcal{E}_\mathcal{K}(\mathcal{K}) = \mathcal{E}(\mathcal{K})$.

5.3 Connectivity under Equivocation

Next, we investigate some technical properties of these constructions that allow us to prove that the final complex is $(k-1)$-connected.

▶ **Lemma 16.** *For any pure, shellable simplicial complex with $\dim(\mathcal{K}) \leq n - rk$, the $\mathcal{K}$-equivocation operator $\mathcal{E}_\mathcal{K}$ is a carrier map.*

Proof. Let $\tau \subseteq \sigma \in \mathcal{K}$. We show that $\mathcal{E}_{\mathcal{K}}(\tau) \subseteq \mathcal{E}_{\mathcal{K}}(\sigma)$. If $\dim(\tau) < n - rk - 1$ then $\mathcal{E}_{\mathcal{K}}(\tau) = \emptyset$ and $\mathcal{E}_{\mathcal{K}}(\tau) \subseteq \mathcal{E}_{\mathcal{K}}(\sigma)$ for any $\sigma \supseteq \tau \in \mathcal{K}$. Otherwise, if $\dim(\tau) = \dim(\sigma)$ then $\tau = \sigma$ and $\mathcal{E}_{\mathcal{K}}(\tau) = \mathcal{E}_{\mathcal{K}}(\sigma)$, as we assumed that $\sigma \supseteq \tau \in \mathcal{K}$. The remaining case is when $\dim(\tau) = n - rk - 1$ and $\dim(\sigma) = n - rk$, which makes $\mathcal{E}_{\mathcal{K}}(\tau) \subseteq \mathcal{E}_{\mathcal{K}}(\sigma)$ in light of Definition 15. ◀

Let $(\mathcal{C}^r \circ \mathcal{E})$ be the composite map such that $(\mathcal{C}^r \circ \mathcal{E})(\sigma) = \mathcal{E}_{\mathcal{C}^r(\sigma)}(\mathcal{C}^r(\sigma))$. While, for an arbitrary complex $\mathcal{K}$, $\mathcal{E}_{\mathcal{K}}$ is not a strict carrier map *per se*, we show in the following lemmas that $(\mathcal{C}^r \circ \mathcal{E})$ is a strict $(k-1)$-connected carrier map. Lemma 17 shows that $(\mathcal{C}^r \circ \mathcal{E})$ is a strict carrier map, and Lemma 18 shows that for any $\sigma \in \mathcal{K}^{r-1}$, $(\mathcal{C}^r \circ \mathcal{E})(\sigma)$ is $((k-1) - \operatorname{codim}_{\mathcal{K}^{r-1}}(\sigma))$-connected.

▶ **Lemma 17.** *$(\mathcal{C}^r \circ \mathcal{E})$ is a strict carrier map.*

Proof. Consider $\sigma, \tau \in \mathcal{K}^{r-1}$, with $\mathcal{L} = \mathcal{C}^r(\sigma)$ and $\mathcal{M} = \mathcal{C}^r(\tau)$. Both $\mathcal{L}$ and $\mathcal{M}$ are pure, shellable simplicial complexes with dimension $n - rk$ (Definition 9 and Lemma 12). Therefore, both the $\mathcal{L}$-equivocation and $\mathcal{M}$-equivocation operators are well-defined. Also, $\mathcal{C}^r$ is a strict carrier map, hence $\mathcal{L} \cap \mathcal{M} = \mathcal{C}^r(\sigma) \cap \mathcal{C}^r(\tau) = \mathcal{C}^r(\sigma \cap \tau)$. Note that $\mathcal{L} \cap \mathcal{M} = \mathcal{C}^r(\sigma \cap \tau)$, if not empty, is a pure, shellable simplicial complex with dimension $n - rk$. Therefore, the $(\mathcal{L} \cap \mathcal{M})$-equivocation operator is well-defined.

First, we show that $\mathcal{E}(\mathcal{L}) \cap \mathcal{E}(\mathcal{M}) \subseteq \mathcal{E}(\mathcal{L} \cap \mathcal{M})$, which implies one direction of our equality:

$$\mathcal{E}(\mathcal{C}^r(\sigma)) \cap \mathcal{E}(\mathcal{C}^r(\tau)) \subseteq \mathcal{E}(\mathcal{C}^r(\sigma) \cap \mathcal{C}^r(\tau)) = \mathcal{E}(\mathcal{C}^r(\sigma \cap \tau)).$$

For clarity, let $F(\mathcal{K}) = \operatorname{Faces}^{n-rk-1}(\mathcal{K})$. Then,

$$\mathcal{E}(\mathcal{L}) \cap \mathcal{E}(\mathcal{M}) = \bigcup_{\mu \in F(\mathcal{L})} \mathcal{E}_{\mathcal{L}}(\mu) \quad \cap \quad \bigcup_{\nu \in F(\mathcal{M})} \mathcal{E}_{\mathcal{M}}(\nu) = \bigcup_{\substack{\mu \in F(\mathcal{L}) \\ \nu \in F(\mathcal{M})}} \mathcal{E}_{\mathcal{L}}(\mu) \cap \mathcal{E}_{\mathcal{M}}(\nu).$$

For arbitrary $\mu \in F(\mathcal{L})$ and $\nu \in F(\mathcal{M})$, if $\mathcal{E}_{\mathcal{L}}(\mu) \cap \mathcal{E}_{\mathcal{M}}(\nu) \neq \emptyset$, consider two cases:

1. μ and ν are proper faces of $\phi \in (\mathcal{L} \cap \mathcal{M})$. In this case,

   $$\mathcal{E}_{\mathcal{L}}(\mu) \cap \mathcal{E}_{\mathcal{M}}(\nu) = \Psi(\operatorname{names}(\mu) \cap \operatorname{names}(\nu); \operatorname{Interp}(\phi)),$$

   which is inside $\mathcal{E}_{\mathcal{L}\cap\mathcal{M}}(\phi) \subseteq \mathcal{E}_{\mathcal{L}\cap\mathcal{M}}(\mathcal{L} \cap \mathcal{M})$.
2. Otherwise, $\mu \subset \phi_1 \in \mathcal{L}$ or $\nu \subset \phi_2 \in \mathcal{M}$. In this case,

   $$\mathcal{E}_{\mathcal{L}}(\mu) \cap \mathcal{E}_{\mathcal{M}}(\nu) = \Psi(\operatorname{names}(\mu) \cap \operatorname{names}(\nu); \operatorname{Interp}(\phi_1) \cap \operatorname{Interp}(\phi_2)).$$

   By Definition 14, the above is non-empty only when $\operatorname{Interp}(\phi_1) = \operatorname{Interp}(\alpha)$ with $\alpha \in \mathcal{L}$, $\operatorname{Interp}(\phi_2) = \operatorname{Interp}(\beta)$ with $\beta \in \mathcal{M}$, and there exists a non-empty set $\mathbb{P}'$ such that $\mathbb{P}' \subseteq \operatorname{names}(\mu) \cap \operatorname{names}(\nu) \subseteq \operatorname{names}(\gamma)$, where $\gamma = \alpha \cap \beta$ with $\dim(\gamma) = n - rk - 1$. Let $\mathbb{P}''$ be a maximal $\mathbb{P}'$ satisfying such condition. Note that $\gamma \in (\mathcal{L} \cap \mathcal{M})$, so $(\mathcal{L} \cap \mathcal{M}) \neq \emptyset$. Since $(\mathcal{L} \cap \mathcal{M})$ is non-empty, it is pure, shellable with dimension $n - rk$, there must exist a simplex $\gamma' \supset \gamma$ with dimension $n - rk$. Moreover, $\operatorname{Interp}(\gamma') = \operatorname{Interp}(\alpha) = \operatorname{Interp}(\phi_1)$ and $\operatorname{Interp}(\gamma') = \operatorname{Interp}(\beta) = \operatorname{Interp}(\phi_2)$ for processes in $\operatorname{names}(\gamma)$, given the definition of Interp. In conclusion, we have $\mathcal{E}_{\mathcal{L}}(\mu) \cap \mathcal{E}_{\mathcal{M}}(\nu) = \Psi(\mathbb{P}''; \operatorname{Interp}(\gamma')) \subseteq \Psi(\operatorname{names}(\gamma); \operatorname{Interp}(\gamma'))$, which is inside $\mathcal{E}_{\mathcal{L}\cap\mathcal{M}}(\gamma') \subseteq \mathcal{E}_{\mathcal{L}\cap\mathcal{M}}(\mathcal{L} \cap \mathcal{M})$.

In the other direction, we have $\mathcal{E}(\mathcal{L}\cap\mathcal{M}) \overset{\text{def}}{=} \mathcal{E}_{\mathcal{L}\cap\mathcal{M}}(\mathcal{L}\cap\mathcal{M}) \subseteq \mathcal{E}_{\mathcal{L}}(\mathcal{L}\cap\mathcal{M}) \subseteq \mathcal{E}_{\mathcal{L}}(\mathcal{L}) \overset{\text{def}}{=} \mathcal{E}(\mathcal{L})$, since

(i) $\mathcal{E}_{\mathcal{L}\cap\mathcal{M}}(\mathcal{X}) \subseteq \mathcal{E}_{\mathcal{L}}(\mathcal{X})$ for any $\mathcal{X} \subseteq \mathcal{L} \cap \mathcal{M}$ (Definition 15); and
(ii) $\mathcal{E}_{\mathcal{L}}$ is a carrier map (Lemma 16).

The same argument proves that $\mathcal{E}(\mathcal{L} \cap \mathcal{M}) \subseteq \mathcal{E}(\mathcal{M})$, and therefore $\mathcal{E}(\mathcal{L} \cap \mathcal{M}) \subseteq \mathcal{E}(\mathcal{L}) \cap \mathcal{E}(\mathcal{M})$. ◀

▶ **Lemma 18.** *For any $\sigma \in \mathcal{K}^{r-1}$, $\mathcal{E}(\mathcal{C}^r(\sigma))$ is $((k-1) - \text{codim}_{\mathcal{K}^{r-1}}(\sigma))$-connected.*

Proof. Consider $\sigma \in \mathcal{K}^{r-1}$ with $\text{codim}_{\mathcal{K}^{r-1}}(\sigma) \leq k$. By Lemma 12, $\mathcal{M} = \mathcal{C}^r(\sigma)$ is a pure, shellable simplicial complex with $\dim(\mathcal{M}) = n - rk = d$. By Definition 15, $\mathcal{E}(\mathcal{M})$ is well-defined and $\dim(\mathcal{E}(\mathcal{M})) = n - rk - 1 = d'$. Note that $d' \geq n - t \geq 2t \geq 2k$, since $n + 1 > 3t$ and $t \geq k$.

First, we show that $\mathcal{E}(\mathcal{M})$ is "highly-connected" – that is, $(2k-1)$-connected. We proceed by induction on $\mu_0 \dots \mu_\ell$, a shelling order of facets of $\mathcal{M}$.

Base. We show that $\mathcal{E}_{\mathcal{M}}(\mu_0)$ is $(2k-1)$-connected. Considering Definition 15, we have that $\mathcal{E}_{\mathcal{M}}(\mu_0) = \mathcal{E}_{\mathcal{M}}(\tau_0) \cup \dots \cup \mathcal{E}_{\mathcal{M}}(\tau_d)$, with $\tau_0 \dots \tau_d$ being all the proper faces of μ_0.
Consider the cover $\{\mathcal{E}_{\mathcal{M}}(\tau_i) : 0 \leq i \leq d\}$ of $\mathcal{E}_{\mathcal{M}}(\mu_0)$, and its associated nerve $\mathcal{N}(\{\mathcal{E}_{\mathcal{M}}(\tau_i) : 0 \leq i \leq d\})$. For any index set $J \subseteq I = \{0 \dots d\}$, let

$$\mathcal{K}_J = \bigcap_{j \in J} \mathcal{E}_{\mathcal{M}}(\tau_j) = \Psi(\bigcap_{j \in J} \text{names}(\tau_j); \text{Interp}(\mu_0))$$

For any J with $|J| \leq d$, we have $\cap_{j \in J} \text{names}(\tau_j) \neq \emptyset$, making $\mathcal{K}_J$ a non-empty pseudosphere with dimension $d' - |J| + 1 \geq 2k - |J| + 1$. So, $\mathcal{K}_J$ is $((2k-1) - |J| + 1)$-connected by Lemmas 4 and 5. The nerve is hence the $(d-1)$-skeleton of I, which is $(d-2) = (d'-1) \geq (2k-1)$-connected. By the Nerve Theorem, $\mathcal{E}_{\mathcal{M}}(\mu_0)$ is also $(2k-1)$-connected.

IH. Assume that $\mathcal{Y} = \cup_{0 \leq y < x} \mathcal{E}_{\mathcal{M}}(\mu_y)$ is $(2k-1)$ connected, and let $\mathcal{X} = \mathcal{E}_{\mathcal{M}}(\mu_x)$. We must show that $\mathcal{Y} \cup \mathcal{X} = \cup_{0 \leq y \leq x} \mathcal{E}(\mu_y)$ is $(2k-1)$-connected. Note that $\mathcal{X}$ is $(2k-1)$-connected by an argument identical to the one above for the base case $\mathcal{E}_{\mathcal{M}}(\mu_0)$. Besides,

$$\mathcal{Y} \cap \mathcal{X} = \left(\bigcup_{0 \leq y < x} \mathcal{E}_{\mathcal{M}}(\mu_y) \right) \cap \mathcal{E}_{\mathcal{M}}(\mu_x) = \bigcup_{0 \leq y < x} (\mathcal{E}_{\mathcal{M}}(\mu_y) \cap \mathcal{E}_{\mathcal{M}}(\mu_x)) \stackrel{\star}{=} \bigcup_{i \in S} \mathcal{E}_{\mathcal{M}}(\tau_i),$$

where $i \in S$ is such that $(\cup_{0 \leq y < x} \mu_y) \cap \mu_x = \cup_{i \in S} \tau_i$. The set S is well-defined since $\mathcal{M}$ is shellable. The step $(\star)$ holds because:

1. $\mathcal{Y} \cap \mathcal{X}$ must include at least $\bigcup_{i \in S} \mathcal{E}_{\mathcal{M}}(\tau_i)$; and
2. $\mathcal{E}_{\mathcal{M}}(\mu_y) \cap \mathcal{E}_{\mathcal{M}}(\mu_x) \neq \emptyset$ only if $\psi = \Psi(\text{names}(\mu_y \cap \mu_x); \text{Interp}(\mu_x))$ exists, the latter inside $\psi' = \Psi(\text{names}(\tau_j); \text{Interp}(\mu_x))$ for some $j \in S$, or we contradict the fact that $\mathcal{M}$ is shellable.

Using an argument identical to the one for $\mathcal{E}_{\mathcal{M}}(\mu_0)$, yet considering the cover $\{\mathcal{E}_{\mathcal{M}}(\tau_i) : i \in S\}$, the nerve of $\mathcal{X} \cap \mathcal{Y}$ is either the $(d-1)$-skeleton of S (if $S = \{0 \dots d\}$) or the whole simplex S (otherwise). By the Nerve Theorem, $\cup_{i \in S} \mathcal{E}_{\mathcal{M}}(\tau_i)$ is $(2k-1)$-connected. Once again, using the Nerve Theorem, since $\mathcal{Y}$ is $(2k-1)$-connected, $\mathcal{X}$ is $(2k-1)$-connected, and $\mathcal{Y} \cap \mathcal{X}$ is $(2k-1)$-connected, we have that $\mathcal{Y} \cup \mathcal{X}$ is $(2k-1)$-connected.

While the equivocation operator yields high connectivity $(2k-1)$ in the pseudosphere $\mathcal{C}^r(\sigma)$, the *composition* of $\mathcal{C}^r$ and $\mathcal{E}_{\mathcal{C}^r(\sigma)}(\mathcal{C}^r(\sigma))$ limits the connectivity to $(k-1)$, since the former map is only defined for simplexes with codimension $\leq k$. Formally, as $\mathcal{C}^r(\sigma) \neq \emptyset$ for any simplex $\sigma \in \mathcal{K}^{r-1}$ with $\text{codim}_{\mathcal{K}^{r-1}}(\sigma) \leq k$, we have that $\mathcal{E}(\mathcal{C}^r(\sigma))$ is $((k-1) - \text{codim}_{\mathcal{K}^{r-1}}(\sigma))$-connected. ◀

From Lemmas 17 and 18, we conclude the following.

▶ **Corollary 19.** *$(\mathcal{C}^r \circ \mathcal{E})$ is a $(k-1)$-connected carrier map.*

▶ **Theorem 20.** *An adversary can keep the protocol complex of a Byzantine synchronous system $(k-1)$-connected for $\lceil t/k \rceil$ rounds.*

Proof. If $m = 0$, $t \bmod k = 0$, and the adversary runs only the crash rounds failing k processes each time, for $r = \lfloor t/k \rfloor = \lceil t/k \rceil$ consecutive rounds. We have the following scenario:

$$(\mathcal{C}^1 \circ \ldots \circ \mathcal{C}^r)(\sigma).$$

Since $\mathcal{C}^i : \mathcal{K}^{i-1} \to 2^{\mathcal{K}^i}$ is a $(k-1)$-shellable carrier map for $1 \leq i \leq r$ (Lemma 12), the composition $(\mathcal{C}^1 \circ \ldots \circ \mathcal{C}^r)$ is a $(k-1)$-connected carrier map for any facet $\sigma \in \mathcal{I}$ (Lemma 13).

If $m > 0$, the adversary performs r crash rounds (failing k processes each time), followed by the extra equivocation round. We have the following scenario:

$$(\mathcal{C}^1 \circ \ldots \circ \mathcal{C}^{r-1} \circ (\mathcal{C}^r \circ \mathcal{E}))(\sigma). \tag{6}$$

Since $\mathcal{C}^i : \mathcal{K}^{i-1} \to \mathcal{K}^i$ is a $(k-1)$-shellable carrier map for $1 \leq i \leq r-1$ (Lemma 12), and $(\mathcal{C}^r \circ \mathcal{E})$ is a $(k-1)$-connected carrier map (Corollary 19), we have that the composition above $(\mathcal{C}^1 \circ \ldots \circ \mathcal{C}^{r-1} \circ (\mathcal{C}^r \circ \mathcal{E}))$ is a $(k-1)$-connected carrier map for any facet $\sigma \in \mathcal{I}$ (Lemma 13). ◀

6 k-Set Agreement and Lower Bound

The k-set agreement problem [7], is a fundamental task having important associations with protocol complex connectivity. In Byzantine systems, it can be difficult to characterize the input of a faulty process, since this process can ignore its "prescribed" input and behave as having a different one. This intrinsically leads to many alternative formulations for the problem in Byzantine systems [9]. In our algorithm, for each Byzantine process, we can commit to at most a single value transmitted as input. We define such value as the *apparent* input value of the Byzantine process. In our adopted formulation, each non-faulty process P_i starts with *any* value I_i from $V = \{v_0, \ldots, v_d\}$, with $d \geq k$ and $t \geq k \geq 1$, and finishes with a value O_i from V, respecting:

1. **Agreement.** At most k values are decided: $|\{O_i : P_i \in \mathbb{G}\}| \leq k$.
2. **Validity.** For any non-faulty process P_i, the output O_i is the input value of one of the participating processes.
3. **Termination.** The protocol finishes in a finite number of rounds.

The k-set agreement problem and connectivity are closely related. Lemma 21, proved in Appendix A, shows that no solution is possible for k-set agreement with a $(k-1)$-connected protocol complex, which, as seen in Section 5, can occur at least until round $\lceil t/k \rceil$.

▶ **Lemma 21.** *If, starting $\sigma \in \mathcal{I}$, the protocol complex $\mathcal{P}(\sigma)$ is $(k-1)$-connected, then no decision function δ solves the k-set agreement problem.*

Proof. Please refer to Appendix A. ◀

We now present a simple k-set agreement algorithm for Byzantine synchronous systems, running in $\lceil t/k \rceil + 1$ rounds. The procedure requires a relatively large number of processes compared to t: we assume $n + 1 \geq k(3t + 1)$. The procedure was designed with the purpose of tightening the connectivity lower bound, favoring simplicity over the optimality on the number of processes.

Algorithm 1 P_x.Agree(I)

```
1:  if k = 1 then
2:      return Decision(Multiset(Cont(p) output by consensus algorithm))
3:  Cont(w) ← ⊥ for all w ∈ T
4:  Cont(λ) ← I                                                    ▷ Gossip
5:  for ℓ : 1 to ⌈t/k⌉ + 1 do
6:      send(S_x^{ℓ−1} = {(w, Cont(w)) : |w| = ℓ − 1})
7:      upon recv(S_y^{ℓ−1} = {(w, v) : |w| = ℓ − 1, v ∈ V ∪ {⊥}}) from P_y
8:          Cont(wP_y) ← v for all (w, v) ∈ S_y^{ℓ−1}
9:      end upon
10: ℙ′ ← {P_i : P_i has a quorum}                                  ▷ Validation
11: if |ℙ′| = (n + 1) − t then
12:     Apply completion rule for all wb where b ∈ ℙ \ ℙ′ and |wb| = ⌈t/k⌉
13: g ← any g such that T(g) is pivotal                            ▷ Decision
14: for ℓ : ⌈t/k⌉ − 1 to 1 do
15:     Apply consensus rule for all non-validated wb where b ∈ ℙ(g) and |wb| = ℓ
16: return Decision(Multiset(Cont(p) : p ∈ T(g)))
```

Non-faulty processes initially execute a *gossip phase* for $\lceil t/k \rceil + 1$ rounds, followed by a *validation phase*, and a *decision phase*, where the output is chosen. Define $R = \lceil t/k \rceil$, and consider the following tree, where nodes are labeled with words over the alphabet $\mathbb{P}$. The root node is labeled as λ, which represents an empty string. Each node w such that $0 \leq |w| \leq R$ has $n + 1$ child nodes labeled wp for all $p \in \mathbb{P}$. Any non-faulty process P_i maintains such tree, denoted T_i.

6.1 The Gossip Phase

For each of the trees maintained by the processes, as discussed above, all nodes w are associated with the value $\text{Cont}_p(w)$, called the *contents* of w. The meaning of those trees is well-known [1]: after the gossip phase, if node $w = p_1 \dots p_x$ is such that $\text{Cont}_p(w) = v$, then p_x told that p_{x-1} told that $\dots$ p_1 had input v to p. The special value $\perp$ represents an absent input. We omit the subscript p when the process is implied or arbitrary. We divide the processes into k disjoint groups: $\mathbb{P}(g) = \{P_x \in \mathbb{P} : x = g \bmod k\}$, for $0 \leq g < k$. For any tree T, we call $T(g)$ the subtree of T having only nodes $wp \in T$ such that $p \in \mathbb{P}(g)$.

6.2 The Validation Phase

In the validation phase, if we have a set $\mathbb{Q}$ containing $(n + 1) - t$ processes that acknowledge all messages coming from process p (making sure that $p \in \mathbb{Q}$) in all rounds $1 \leq r \leq R$, we call such set the *quorum* of p, denoted $\text{Quorum}(p)$. Formally, $\text{Quorum}(p) = \mathbb{Q} \subseteq \mathbb{P}$ such that $p \in \mathbb{Q}$, $|\mathbb{Q}| \geq (n + 1) - t$, and $q \in \mathbb{Q}$ whenever $\text{Cont}(wp) = v$ implies $\text{Cont}(wpq) = v$, for any wp with $1 \leq |wp| \leq R$. It should be clear that every non-faulty process has a quorum containing at least all other non-faulty processes. If a process p has a quorum as seen by process $P_i \in \mathbb{G}$, we say that wp has been *validated* on P_i, for any wp with $1 \leq |wp| \leq R$. We also say that p has been validated on P_i in this case. Note that in our definition either all entries wp with $1 \leq |wp| \leq R$ are validated, or none is. Lemma 22 shows that validated entries are unique across non-faulty processes.

▶ **Lemma 22.** *If p has been validated on non-faulty processes P_i and P_j, then* $\text{Cont}_i(wp) = \text{Cont}_j(wp)$ *for any* $0 \leq |w| < R$.

Proof. If p has been validated on $P_i \in \mathbb{G}$, then $\text{Cont}_i(wp) = v$ implies $\text{Cont}_i(wpq) = v$ for $(n+1) - t$ different processes $q \in \mathbb{Q}_i$, and $\text{Cont}_j(wp) = v'$ implies $\text{Cont}_j(wpq) = v'$ for $(n+1) - t$ different processes $q \in \mathbb{Q}_j$, for any $0 \leq |w| < R$. As we have at most t non-faulty processes and $n + 1 > 3t$, $|\mathbb{Q}_i \cap \mathbb{Q}_j| \geq (n+1) - 2t > t + 1$, containing at least one non-faulty process that, in contradiction, would be broadcasting values consistently in its run. Hence, $v = \text{Cont}_i(wp)$ and $v' = \text{Cont}_j(wp)$ must be identical. ◀

6.3 The Decision Phase

In the decision phase, if we see t processes without a quorum, we have technically identified all non-faulty processes $\mathbb{B}$. In this case, we fill R-th round values of any $b \in \mathbb{B}$ using the *completion* rule: we make $\text{Cont}(wb) = v$ if we have $(n+1) - 2t$ processes $\mathbb{G}' \subseteq \mathbb{G}$ where $\text{Cont}(wbg) = v$ for any $g \in \mathbb{G}'$ and $|wb| = R$. If a process b has its R-round values completed as above in process $P_i \in \mathbb{G}$, we say that wb has been *completed* on P_i for any $|wb| = R$. Lemma 23 shows that completed entries are identical and consistent with validated entries across non-faulty processes. (Intuitively, the completion rule was done over identical values from correct processes.)

▶ **Lemma 23.** *If wp has been completed or validated on a non-faulty process P_i, and wp has been completed on a non-faulty process P_j, then* $\text{Cont}_i(wp) = \text{Cont}_j(wp)$.

Proof. Say wp has been validated on P_i and completed in P_j. Since wp has been validated on P_i, $\text{Cont}_i(wp) = v$ implies $\text{Cont}_i(wpq) = v$ for $(n+1) - t$ different processes $q \in \mathbb{Q}$. When P_j applies the completion rule on wp, we must have $\text{Cont}_j(wpq) = v$ for $(n+1) - 2t$ different processes $q \in \mathbb{G}$, as we have at most t faulty processes. Therefore, $\text{Cont}_i(wp) = \text{Cont}_j(wp)$.

If wp has been completed on all non-faulty processes, they all have identified t faulty processes, and the completion rule is performed over identical entries associated with non-faulty processes. Therefore, $\text{Cont}_i(wp) = \text{Cont}_j(wp)$ in this case as well. ◀

▶ **Definition 24.** We define a *pivotal subtree* as follows:

1. If there exists a subtree $T(g)$ with less than $\lceil t/k \rceil$ non-validated processes, define this subtree as pivotal;
2. Otherwise, we identified $k \cdot \lceil t/k \rceil \geq t$ Byzantine processes, so we apply the completion rule consistently to R-round values in $T(0)$, and define $T(0)$ as pivotal instead.

A pivotal subtree, therefore, must exist according to Definition 24. For that subtree, any sequence $p_1, (p_1p_2), \ldots, (p_1p_2 \ldots p_x)$, with $p_1 \neq \ldots \neq p_x$, has size $x < R = \lceil t/k \rceil$. As we see further ahead, this will allow us to suitably perform consensus over consistent values.

We first highlight that, essentially, our algorithm is separating the possible chains of unknown values across disjoint process groups, which either forces one of these chains to be smaller than $R = \lceil t/k \rceil$, or reveals all faulty processes, giving us the ability to perform the completion rule in a consistent way. This fundamental tradeoff underlies our algorithm, and ultimately explains *why* the $\lceil t/k \rceil$ connectivity bound is tight for relatively large numbers of n compared to t.

6.3.1 The Consensus Rule

Denote the set of processes in the word w as $\text{SetProc}(w)$. For any non-validated wb with $b \in \mathbb{P}(g)$ in a pivotal subtree $T(g)$, where $1 \leq |wb| < R$, we establish consensus on $\text{Cont}(wb)$. We apply the *consensus* rule: $\text{Cont}(wb) = v$ if the majority of processes in $\mathbb{P}(g) \setminus \text{SetProc}(wb)$

is such that $wbp = v$. This rule is applied first to entries labeled wb where $|wb| = R - 1$, and then moving upwards (please refer to Algorithm 1). Lemma 25 shows that the consensus rule indeed establishes consensus across non-faulty processes that identify $T(g)$ as the pivotal subtree.

▶ **Lemma 25.** *For any two-non-faulty processes P_i and P_j that applied the consensus rule on a pivotal subtree $T(g)$, with $0 \leq g < k$, we have that $\text{Cont}_i(p) = \text{Cont}_j(p)$ for any $p \in \mathbb{P}(g)$.*

Proof. Consider a non-faulty process P_i establishing the value of $\text{Cont}_i(wp)$ with the consensus rule. Define $\text{SetCons}(wp) = \mathbb{P}(g) \setminus \text{SetProc}(wp)$ for any $wp \in T(g)$ with $|wp| < R$, noting that $|\text{SetCons}(wp)| \geq 2t + 2$ as $|\mathbb{P}(g)| \geq 3t + 1$ and $|wp| < t$.

There are two possible cases:

1. If wp has been validated at a non-faulty process P_j with $\text{Cont}_j(wp) = v$, at most t values from $S_i = \text{Multiset}(\text{Cont}_i(wpq) : q \in \text{SetCons}(wp))$ will be different than v. Hence, there will always be a majority of values in S_i that will contain v, because $|S_i| \geq 2t + 2$.
2. Otherwise, if wp has not been validated at any non-faulty process, all $\text{Cont}(wp)$ values are being calculated over consistent values, by Lemma 23, which makes all non-faulty processes establish $\text{Cont}(wp)$ consistently with the consensus rule. ◀

▶ **Theorem 26.** *Algorithm 1 solves k-set agreement in $\lceil t/k \rceil + 1$ rounds for $n + 1 > k(3t + 1)$.*

Proof. Termination is trivial, as we execute exactly $R = \lceil t/k \rceil + 1$ rounds. By Lemma 25, each pivotal subtree yields a unique decision value. As we have at most k pivotal subtrees identified across non-faulty processes, up to k values are possibly decided across non-faulty processes. ◀

7 Conclusion

In Byzantine synchronous systems, the protocol complex can remain $(k - 1)$-connected for $\lceil t/k \rceil$ rounds, potentially *one* more round than in crash-failure systems. We conceive a combinatorial operator modeling the ability of Byzantine processes to *equivocate* without revealing their Byzantine nature, just after $\lfloor t/k \rfloor$ rounds of crash failures. We compose this operator with the regular crash-failure operators, extending $(k - 1)$-connectivity up to $\lceil t/k \rceil$ rounds. We tighten this bound, at least when n is relatively large compared to t, via a full-information protocol that solves a formulation of k-set agreement.

It may be surprising that Byzantine failures impose only *one* additional synchronous round over the crash-failure model, and *at most that* in our setting, where inputs are arbitrarily attributed to processes, and the number of processes is at least $k(3t + 1)$. In terms of solvability vs. number of rounds, the penalty for moving from crash to Byzantine failures can thus be *quite limited*. Previous work has hinted this possibility operationally, since

- **(i)** in synchronous systems where n is large enough compared to t, we can simulate crash failures on Byzantine systems with a 1-round delay [2]; and
- **(ii)** techniques similar to the reliable broadcast of [4, 25] deal with the problem of Byzantine equivocation, also with a 1-round delay.

This extra round is crucial – but enough – to limit the impact of Byzantine behavior in rather usual operational settings.

The algorithm that matches the connectivity bound was designed to separate chains of unresolved values, such that we suitably limit their size, or force the adversary to reveal the identity of all faulty processes. The prospect of an algorithm that applies similar ideas, however with better resilience, is a thought-provoking perspective for future work.

References

1 Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley Interscience, 2nd edition, March 2004.

2 Rida A. Bazzi and Gil Neiger. Simplifying fault-tolerance: Providing the abstraction of crash failures. *Journal of the ACM*, 48(3):499–554, May 2001. doi:10.1145/382780.382784.

3 A. Björner. Topological methods. In R. L. Graham, M. Grötschel, and L. Lovász, editors, *Handbook of Combinatorics*, volume 2, pages 1819–1872. MIT Press, Cambridge, MA, USA, December 1995. URL: http://dl.acm.org/citation.cfm?id=233228.233246.

4 G. Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75(2):130–143, November 1987.

5 Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer, 2 edition, February 2011.

6 Armando Castañeda, Yannai A. Gonczarowski, and Yoram Moses. Brief announcement: Pareto optimal solutions to consensus and set consensus. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, PODC'13, pages 113–115, New York, NY, USA, 2013. ACM. doi:10.1145/2484239.2484280.

7 Soma Chaudhuri. More choices allow more faults: set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132–158, July 1993.

8 Soma Chaudhuri, Maurice Herlihy, Nancy A. Lynch, and Mark R. Tuttle. Tight bounds for k-set agreement. *Journal of the ACM*, 47(5):912–943, September 2000. doi:10.1145/355483.355489.

9 Roberto de Prisco, Dahlia Malkhi, and Michael Reiter. On k-set consensus problems in asynchronous systems. *IEEE Transactions on Parallel and Distributed Systems*, 12(1):7–21, January 2001. doi:10.1109/71.899936.

10 Eli Gafni, Petr Kuznetsov, and Ciprian Manolescu. A generalized asynchronous computability theorem. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC'14, pages 222–231, New York, NY, USA, 2014. ACM. doi:10.1145/2611462.2611477.

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

12 Maurice Herlihy and Sergio Rajsbaum. Concurrent computing and shellable complexes. In Nancy Lynch and Alexander Shvartsman, editors, *Distributed Computing*, volume 6343 of *Lecture Notes in Computer Science*, pages 109–123. Springer Berlin / Heidelberg, 2010. doi:10.1007/978-3-642-15763-9_10.

13 Maurice Herlihy, Sergio Rajsbaum, and Mark Tuttle. An axiomatic approach to computing the connectivity of synchronous and asynchronous systems. *Electronic Notes in Theoretical Computer Science*, 230(0):79–102, March 2009. doi:10.1016/j.entcs.2009.02.018.

14 Maurice Herlihy, Sergio Rajsbaum, and Mark R. Tuttle. Unifying synchronous and asynchronous message-passing models. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, PODC'98, pages 133–142, New York, NY, USA, 1998. ACM. doi:10.1145/277697.277722.

15 Maurice Herlihy and Nir Shavit. The asynchronous computability theorem for t-resilient tasks. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, STOC'93, pages 111–120, New York, NY, USA, 1993. ACM. doi:10.1145/167088.167125.

16 Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858–923, November 1999.

17 Dimitry N. Kozlov. *Combinatorial Algebraic Topology*, volume 21 of *Algorithms and Computation in Mathematics*. Springer, 1st edition, October 2007. doi:10.1007/978-3-540-71962-5.

18 Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transaction on Programming Languages and Systems*, 4(3):382–401, July 1982. doi:10.1145/357172.357176.

19 Hammurabi Mendes, Christine Tasson, and Maurice Herlihy. Distributed computability in Byzantine asynchronous systems. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing*, STOC'14, pages 704–713, New York, NY, USA, 2014. ACM. doi:10.1145/2591796.2591853.

20 James Munkres. *Elements of Algebraic Topology*. Prentice Hall, 2nd edition, January 1984. URL: http://www.worldcat.org/isbn/0131816292.

21 James Munkres. *Topology*. Pearson, 2nd edition, January 2000.

22 Gil Neiger. Distributed consensus revisited. *Information Processing Letters*, 49(4):195–201, February 1994. doi:10.1016/0020-0190(94)90011-6.

23 Gil Neiger and Sam Toueg. Automatically increasing the fault-tolerance of distributed algorithms. *Journal of Algorithms*, 11(3):374–419, 1990. doi:10.1016/0196-6774(90)90019-B.

24 Michael Saks and Fotios Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, STOC'93, pages 101–110, New York, NY, USA, 1993. ACM. doi:10.1145/167088.167122.

25 T. Srikanth and S. Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2(2):80–94, June 1987.

A Appendix: Proofs for the Connectivity Arguments

Proof of Lemma 21. Consider a k-simplex $\alpha = \{u_0, \ldots, u_k\} \subseteq \{v_0, \ldots, v_d\}$ with $k+1$ different inputs. Let $\mathcal{I}_\beta = \Psi(\mathbb{P}, \beta)$ for any $\beta \subseteq \alpha$, and $\mathcal{I}_x = \bigcup_{\beta \in \mathrm{skel}^x(\alpha)} \Psi(\mathbb{P}, \beta)$. We construct a sequence of continuous maps $g_x : |\,\mathrm{skel}^x(\alpha)| \to |\mathcal{K}_x|$ where $\mathcal{K}^x$ is homeomorphic to $\mathrm{skel}^x(\alpha)$ in $|\,\mathrm{skel}^x(\mathcal{P}(\mathcal{I}_x))|$.

Base. Let g_0 map any vertex $v \in \alpha$ to a vertex in $\mathcal{K}_v = \mathcal{P}(\mathcal{I}_{\{v\}})$. We know that $\mathcal{K}_v$ is k-connected since $\dim(\mathcal{I}_{\{v\}}) = \dim(\mathcal{I})$ and $\mathcal{P}$ is a k-connected carrier map. We just constructed

$$g_0 : |\,\mathrm{skel}^0(\alpha)| \to |\mathcal{K}_0|,$$

where $\mathcal{K}^0$ is isomorphic to a $\mathrm{skel}^0(\alpha)$ in $|\,\mathrm{skel}^0(\mathcal{P}(\mathcal{I}_0))|$.

Induction Hypothesis. Assume $g_{x-1} : |\,\mathrm{skel}^{x-1}(\alpha)| \to |\mathcal{K}_{x-1}|$ for any $x \leq k$, where $\mathcal{K}_{x-1}$ is isomorphic to $\mathrm{skel}^{x-1}(\alpha)$ in $|\,\mathrm{skel}^{x-1}(\mathcal{P}(\mathcal{I}_{x-1}))|$. For any $\beta \in \mathrm{skel}^x(\alpha)$, we have that $\mathrm{skel}^x(\mathcal{P}(\mathcal{I}_\beta))$ is $(x-1)$-connected, hence the continuous image of the $(x-1)$-sphere in $\mathcal{P}(\mathcal{I}_\beta)$ can be extended to the continuous image of the x-disk in $\mathrm{skel}^x(\mathcal{P}(\mathcal{I}_\beta))$. We just constructed

$$g_x : |\,\mathrm{skel}^x(\alpha)| \to |\mathcal{K}_x|,$$

where $\mathcal{K}^x$ is isomorphic to $\mathrm{skel}^x(\alpha)$ in $|\,\mathrm{skel}^x(\mathcal{P}(\mathcal{I}_0))|$. In the end, we have $g_k : |\alpha| \to |\mathcal{K}_k|$ where $\mathcal{K}_k$ is isomorphic to α in $\mathrm{skel}^k(\mathcal{P}(\mathcal{I}_k))$.

Now suppose, for the sake of contradiction, that k-set agreement is solvable, so there must be a simplicial map $\delta : \mathcal{P}(\mathcal{I}) \to \mathcal{O}$ carried by Δ. Then, induce the continuous map $\delta_c : |\mathcal{K}_k| \to |\alpha|$ from δ such that $\delta_c(v) \in |\,\mathrm{views}(\delta(\mu))|$ if $v \in |\mu|$, for any $\mu \in \mathcal{K}_k$. Also, note that the composition of g_k with the continuous map δ_c induces another continuous map

$|\alpha| \to |\partial\,\alpha|$, since by assumption δ never maps a k-simplex of $\mathcal{K}_k$ to a simplex with $k+1$ different views (so δ_c never maps a point to $|\operatorname{Int}\alpha|$). We built a *continuous retraction* of α to its own border $\partial\,\alpha$, a contradiction (please refer to [20, 17]). Since our assumption was that there existed a simplicial map $\delta : \mathcal{P}(\mathcal{I}) \to \mathcal{O}$ carried by Δ, we conclude that k-set agreement is not solvable. ◀

Recovering Shared Objects Without Stable Storage*†

Ellis Michael[1], Dan R. K. Ports[2], Naveen Kr. Sharma[3], and Adriana Szekeres[4]

1 **University of Washington, Seattle, USA**
emichael@cs.washington.edu

2 **University of Washington, Seattle, USA**
drkp@cs.washington.edu

3 **University of Washington, Seattle, USA**
naveenks@cs.washington.edu

4 **University of Washington, Seattle, USA**
aaasz@cs.washington.edu

Abstract

This paper considers the problem of building fault-tolerant shared objects when processes can crash and recover but lose their persistent state on recovery. This Diskless Crash-Recovery (DCR) model matches the way many long-lived systems are built. We show that it presents new challenges, as operations that are recorded at a quorum may not persist after some of the processes in that quorum crash and then recover.

To address this problem, we introduce the notion of *crash-consistent quorums*, where no recoveries happen during the quorum responses. We show that relying on crash-consistent quorums enables a recovery procedure that can recover all operations that successfully finished. Crash-consistent quorums can be easily identified using a mechanism we term the *crash vector*, which tracks the causal relationship between crashes, recoveries, and other operations.

We apply crash-consistent quorums and crash vectors to build two storage primitives. We give a new algorithm for multi-writer, multi-reader atomic registers in the DCR model that guarantees safety under all conditions and termination under a natural condition. It improves on the best prior protocol for this problem by requiring fewer rounds, fewer nodes to participate in the quorum, and a less restrictive liveness condition. We also present a more efficient single-writer, single-reader atomic set – a *virtual stable storage* abstraction. It can be used to lift any existing algorithm from the traditional Crash-Recovery model to the DCR model. We examine a specific application, state machine replication, and show that existing diskless protocols can violate their correctness guarantees, while ours offers a general and correct solution.

1998 ACM Subject Classification E.1 Distributed Data Structures, H.3.4 Distributed Systems

Keywords and phrases asynchronous system, fault-tolerance, crash-recovery, R/W register, state machine replication

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.36

* An extended version of this paper is available as a technical report [27].

† This material is based upon work supported by the National Science Foundation under award CNS-1615102 and a Graduate Research Fellowship, and by gifts from Google and VMware.

© Ellis Michael, Dan R. K. Ports, Naveen Kr. Sharma, and Adriana Szekeres;
licensed under Creative Commons License CC-BY

1 Introduction

Today's distributed systems are key pieces of infrastructure that must remain available even though the servers that implement them are constantly failing. These systems are long-lived and must be able to tolerate nodes crashing and rejoining the system. In particular, nodes must be able to rejoin the system even after losing their disk state, a real concern for large-scale data centers where hard drive failures are a regular occurrence [7].

This paper addresses the problem of how to build recoverable shared objects even when processes lose their entire state. We consider the *Diskless Crash-Recovery* model: each process in the system may go down at any time; upon recovery, it loses all state it had before the crash except for its identity. However, processes can run a *recovery protocol* to reconstruct their state before deeming themselves operational again. This model matches the way that many distributed systems are built in practice.

The Diskless Crash-Recovery model (DCR) is more challenging than the traditional Crash-Stop model (CS) or the Crash-Recovery with Stable Storage model (CRSS). The main challenge is that an invariant that holds at one process may not hold on that process's next incarnation after recovery. This leads to the problem of *unstable quorums*: it is possible for a majority of processes to acknowledge a write operation, and yet processes can still subsequently lose that knowledge after crash and recovery.

We provide a general mechanism for building recoverable shared objects in the DCR model. We show that an operation can be made recoverable once it is stored by a *crash-consistent quorum*, which we informally define as one where no recoveries happen during the quorum responses. Crash-consistent quorums can be efficiently identified using a mechanism called the *crash vector*: a vector, maintained by each process, that tracks the latest known incarnation of each process. By including crash vectors in protocol messages, processes can identify the causal relationship between crash recoveries and other operations. This makes it possible to discard responses that are not part of a crash-consistent quorum. We show that this is sufficient to make storage mechanisms recoverable.

The crash-consistent quorum approach is a general strategy for making storage primitives recoverable. We give two concrete examples in this paper, both of which are always safe and guarantee liveness during periods of stability; other storage primitives are also possible:

- First, we build a *multi-writer, multi-reader atomic register* by extending the well-known ABD protocol [3] with crash vectors. This improves on the best prior protocol by Konwar et al. [17], $RADON_R^{(S)}$, for this problem: it requires fewer rounds (2 rather than 3), requires fewer nodes to participate in the protocol (a simple majority vs $3/4$), and has a less restrictive liveness condition.
- Second, we construct a *single-writer, single-reader atomic set*, which has weaker semantics yet permits a more efficient implementation, requiring only a single round of communication for writes. We refer to this algorithm as *virtual stable storage*, as it offers consistency semantics similar to a local disk. We show that the virtual stable storage protocol can be used to transform any protocol that operates in the traditional CS or CRSS models to one that operates in DCR.

We discuss the application of this work to state machine replication, a widely used distributed system technique. Recovering from disk failures is an important concern in practice, and recent replication protocols attempt to support recovery after complete loss of state. Surprisingly, we find that each of the three such protocols [7, 16, 22] can lose data. We identify a general problem: while these protocols go to great lengths to ensure that a recovering replica reconstructs the set of operations it previously processed, they fail to

recover critical *promises* the replica has previously made, e.g., to elect a new leader. This is due to the fact that these protocols rely on *unstable quorums* to persist these promises. This causes nodes to break important invariants upon recovery, causing the system to violate safety properties. Our approach provides a correct, general, and efficient solution.

To summarize, this paper makes the following contributions:

- It formalizes a *Diskless Crash-Recovery (DCR)* failure model in a way that captures the challenges of long-lived applications (Section 3).
- It introduces the notion of *crash-consistent quorums* and provides two communication primitives for reading from and writing to crash-consistent quorums (Section 4).
- It presents algorithms built on top of our communications primitives for two different shared objects in the DCR model: an atomic multi-writer, multi-reader register and an atomic single-writer, single-reader set. The former is a general purpose register which demonstrates the generality of our approach, while the latter provides a *virtual stable storage* interface that can be used to port any protocol in the CRSS model to one for the DCR model (Section 5).
- Finally, it examines prior protocols for state machine replication in the DCR failure model and demonstrates flaws in these protocols that lead to violations of safety properties. Our two communication primitives can provide correct solutions (Section 6).

2 Background and Related Work

Static Systems. A static system comprises a fixed, finite set of processes. Fault-tolerant protocols for reliable storage for static systems have been studied extensively in the Crash-Stop (CS) failure model, where processes that fail never rejoin the system, and the Crash-Recovery with Stable Storage model (CRSS). In the latter model, processes recover with the same state after a crash. Consensus and related problems, in particular, have been studied extensively in these settings [8, 12, 30]. In CRSS, a crashed and recovered node is no different than one which was temporarily unavailable; asynchronous algorithms that tolerate lossy networks are inherently robust to these types of failures [8].

Prior work on fault-tolerant shared objects and consensus without stable storage generally requires some subset of the processes to never fail [2, 11]. Aguilera et al. [2] showed an impossibility result for a crash-recovery model: even with certain synchrony assumptions, consensus cannot be solved *without at least one process that never crashes*. The main differentiator between that work and this paper is that in their model, the states of processes were binary – either "up" or "down." We overcome this limitation by adding an extra "recovering" state. As long as the number of processes which are "down" or "recovering" at any given time is bounded, certain problems can be solved even *without processes that never fail*.

Recently, Konwar et al. [17] presented a set of algorithms for implementing an atomic multi-writer, multi-reader (MWMR) register in a model similar to ours. We generalize and improve on this work using new primitives for crash-consistent quorums. Our techniques are applicable to other forms of shared objects as well, and our MWMR register is more efficient: it requires one fewer phase and a simple majority quorum (vs $^3/_4$).

Several recent practical state machine replication systems [7, 16, 22] incorporate ad hoc recovery mechanisms for nodes to recover from total disk loss. The common intuition behind these approaches is that a write to disk can be replaced with a write to a quorum of other nodes, recovering after a failure by performing a quorum read. However, we show that these protocols are not correct; they can lose data in certain failure scenarios. A more recent design, Replacement [13], provides a mechanism for replacing failed processes. Like our work

and the epoch vectors in JPaxos [16], it draws on concepts like version vectors [31] and vector clocks [9] to determine the causal dependencies between replacements and other operations. We build on these techniques to provide generic communication primitives in DCR.

Dynamic Systems. In a dynamic setting, processes may leave or join the system at will. Although we consider a static system, DCR may be viewed as a dynamic system with a finite concurrency level [26], i.e, where there is a finite bound on the maximum number of processes that are simultaneously active, over all runs. Here, a recovering process without state is equivalent to a newly joined process.

Many dynamic systems implement *reconfiguration* protocols [1, 10, 21–24, 33]. Reconfiguration allows one to change the set of members allowed to participate in the computation. This process allows both adding new processes and removing processes from the system. Reconfiguration is a more general problem than recovery: it can be used to handle disk failure by introducing a recovering node as a new member and removing its previous incarnation. However, general reconfiguration protocols are a blunt instrument, as they must be able to handle completely changing the membership to a disjoint set of processes . As a result, these protocols are costly. Most use consensus to agree on the order of reconfigurations, which delays the processing of concurrent operations [28]. DynaStore [1] is the first proposal which does not require consensus, but reconfigurations can still delay R/W operations [28]. SmartMerge [14] improves on DynaStore by offering a more expressive reconfiguration interface. Recovery is a special case of reconfiguration, where each recovering process replaces, and has the same identity as, a previously crashed process. As a result, it permits more efficient solutions.

Other protocols implement shared registers and other storage primitives in churn-prone systems [4–6, 15]. In these systems, processes are constantly joining and leaving the system, but at a bounded rate. These protocols remain safe only when churn remains within the specified bound, in contrast to our work which is always safe. Most of these protocols also require synchrony assumptions for correctness. However, under these assumptions they are able to provide liveness guarantees even during constant churn.

3 System Model

We begin by defining our failure model: *Diskless Crash-Recovery* (DCR), a variant of the classic Crash-Recovery model where processes lose their entire state upon crashing.

We consider an asynchronous distributed system which consists of a fixed set of n processes, Π. Each process has a unique name (identifier) of some kind; we assume processes are numbered $1, \ldots, n$ for simplicity. Each process executes a protocol (formally, it is an I/O automaton [25]) while it is up. An execution of a protocol proceeds in discrete time steps, numbered with $\mathbb{N}$, starting at $t = 0$. At each step, at most one process either processes an input action, processes a message, crashes, or restarts. If it *crashes*, the process stops receiving messages and input actions, loses its state, and is considered DOWN. A process that is DOWN can *restart* and transition back to the UP state. We make the following assumptions about a process that restarts: (1) it knows it is restarting, (2) it knows its unique name and the names of the other processes in the system (i.e., this information survives crashes), and (3) it can obtain an incarnation ID that is distinct from all the ones that it previously obtained. Note that the incarnation ID need only be unique among different incarnations of a specific process, not the entire system. These are reasonable assumptions to make for real-world systems: (1) and (2) are fixed for a given deployment, and (3) can be obtained, for example, from a source of randomness or the local processor clock.

Processes are connected by an asynchronous network. Messages can be duplicated a finite number of times or reordered arbitrarily – but not modified – by the network. We assume that if an incarnation of a process remains UP, sends a message, and an incarnation of the destination process stays UP long enough, that message will eventually be delivered.[1]

The unique incarnation ID makes it possible to distinguish different incarnations of the same process. Without unique incarnation IDs, processes are vulnerable to "replay attacks:"

▶ **Theorem 1.** *Any state reached by a process that has crashed, restarted, and taken steps without receiving an input action or crashing again will always be reachable by that process.*

Proof. Suppose process p has crashed, restarted, and taken some number of steps without crashing or receiving an input action. That is, suppose that after it restarted, p received some sequence of messages, $\mathcal{M}$. Because p is an I/O automaton without access to randomness or unique incarnation IDs, anytime p crashes and restarts, it restarts into the exact same state. Furthermore, if p crashes, restarts, and receives the same sequence of messages, $\mathcal{M}$, having been duplicated by the network, p will always end up in the same state. ◀

A corollary to Theorem 1 is that any protocol in the DCR model without unique incarnation IDs satisfying the safety properties of consensus – or even a simple shared object such as a register – can reach a state from which terminating states are not reachable (i.e., a state of deadlock). If all processes crash and recover before deciding a value or receiving a write, they can always return to this earlier state, so the protocol cannot safely make progress.

For simplicity of exposition, we assume that the incarnation ID increases monotonically. We explain in our technical report [27] how to eliminate this requirement.

A restarting process must recover parts of its state. To do so, it runs a distinct *recovery protocol*. This protocol can communicate with other processes to recover previous state. Once the recovery protocol terminates, the process declares recovery complete and resumes execution of its normal protocol. We describe a process that is UP as RECOVERING if it is running its recovery protocol and OPERATIONAL when it is running the initial automaton. A protocol in this model should satisfy *recovery termination*: a recovering process eventually becomes OPERATIONAL, as long as it does not crash again in the meantime. This precludes vacuous solutions where recovering process never again participate in the normal protocol.

Using a separate recovery protocol matches the design of existing protocols like Viewstamped Replication [22]. Importantly, the distinction between RECOVERING and OPERATIONAL allows failure bounds in terms of the number of OPERATIONAL processes, e.g., that fewer than half of the processes can be either DOWN or RECOVERING at any moment. This circumvents Aguilera et al.'s impossibility result for consensus [2], which does not make such a distinction (i.e., restarting processes are immediately considered OPERATIONAL).

4 Achieving Crash-Consistent Quorums

Making shared objects recoverable in the DCR model requires a new type of quorum to capture the idea of persistent, recoverable knowledge. A simple quorum does not suffice. We demonstrate the problem through a simple straw-man example, and introduce the concepts of *crash-consistent quorums* and *crash vectors* to solve the problem. We use these to build generic quorum communication and recovery primitives.

[1] This model is equivalent to one in which the network can drop any message a finite number of times, with the added stipulation that processes resend messages until they are acknowledged.

4.1 Unstable Quorums: Intuition

Consider an intentionally simple example: a fault-tolerant *safe* register that supports a single writer and multiple readers. A safe register [19] is the weakest form of register, as the behavior of READ operations is only defined when there are no concurrent WRITEs. We further constrain the problem by allowing the writer to only ever execute one WRITE operation. That is, the only safety requirement is that once the WRITE completes, all subsequent READs that return must return the value written.

In the Crash-Stop model, a trivial quorum protocol suffices: WRITE(val) broadcasts val and waits for acknowledgments from a quorum. Here, we consider majority quorums:

▶ **Definition 2.** A quorum Q is a set of processes such that $Q \in \mathcal{Q} = \{Q : Q \in 2^{\Pi} \wedge |Q| > n/2\}$.

A subsequent READ would then be implemented by reading from a quorum. The quorum intersection property (i.e., $\forall Q_1, Q_2 \in \mathcal{Q} \;\; Q_1 \cap Q_2 \neq \{\}$) guarantees that at least one process will return val for a READ that happens after the WRITE. It is easy to extend this protocol to the CRSS model simply by having each process log val to disk before replying to a WRITE.

Could we use this same quorum protocol in our DCR model, where processes that crash recover without stable storage, by augmenting it with a recovery protocol that satisfies recovery termination? In fact, for this particular protocol, there is *no* recovery protocol that both guarantees the safety requirement and recovery termination – even if there is a majority of processes which are OPERATIONAL at any instant! In order to tolerate the crashes of a minority of processes and satisfy recovery termination, any recovery protocol must be able to proceed after communicating with only a simple majority of processes. However, if a process crashes in the middle of the WRITE procedure – after acknowledging val – it may recover before a majority of processes have received val. No recovery procedure that communicates only with this quorum of processes can cause the process to relearn val.

We term the resulting situation an *unstable quorum*: the WRITE operation received responses from a quorum, and yet by the time it completes there may no longer exist a majority of processes that know val. It is thus possible to form a quorum of processes that either acknowledged val but then lost it during recovery, or never received the write request (delayed by the network). A subsequent READ could fail by reading from such a quorum.

Although this is a simple example, many important systems suffer from precisely this problem of unstable quorums. We show in Section 6 that essentially this scenario can cause three different state machine replication protocols to lose important pieces of state.

4.2 Crash-Consistent Quorums

We can avoid this problem – both for the straw-man problem above and in the general case – by relying not just on simple quorums of responses but *crash-consistent* ones.

Crash Consistency. We informally define a *crash-consistent quorum* to be one where no recoveries of processes in the quorum *happen during* the quorum responses. More precisely:

▶ **Definition 3.** Let $\mathcal{E}$ be the set of all events in an execution. A set of events, $E \subseteq \mathcal{E}$, is *crash-consistent* if $\forall e_1, e_2 \in E$ there is no $e_3 \in \mathcal{E}$ that takes place at a later incarnation of the same process as e_1 such that $e_3 \rightarrow e_2$. Here, $\rightarrow$ represents Lamport's happens-before relation [18].

In Section 4.3, we show how to build recoverable primitives using crash-consistent quorums, in which all quorum replies (i.e. the message send events at a quorum) are crash-consistent.

Crash Vectors. How does a process acquire a crash-consistent quorum of responses? The mechanism that allows us to ensure a crash-consistent quorum is the *crash vector*. This is a vector that contains, for each process, its latest known incarnation ID. Like a version vector, processes attach their crash vector to the relevant protocol messages and use incoming messages to update their crash vector. The crash vector thus tracks the causal relationship between crash recoveries and other operations. When acquiring a quorum on a WRITE operation, we check whether any of the crash vectors are inconsistent with each other, indicating that a recovery may have happened concurrently with one of the responses. We then discard any responses from previous incarnations of the recovering process, ensuring a crash-consistent quorum, and thus avoiding the aforementioned problem.

4.3 Communication Primitives in DCR

We now describe in detail two generic quorum communication primitives, one of which acquires a *crash-consistent quorum*, as well as a generic recovery procedure. These primitives require their users to implement an abstract interface: Read-State, which returns a representation of the state of the shared object; Update-State, which alters the current state with a specific value; and Rebuild-State, which is called during recovery and takes a set of state representations and combines them.

The Acquire-Quorum primitive writes a value to a crash-consistent quorum and returns the latest state. The Read-Quorum primitive returns a *fresh* – but possibly inconsistent – snapshot of the state as maintained at a quorum of processes. If Acquire-Quorum(val) succeeds, then any subsequent Read-Quorum will return at least one response from a process that *knows* (i.e., has previously updated its state with) val.

The detailed protocol implementing the two primitives and the recovery procedure is presented as pseudo-code in Algorithm 1. We present the algorithm using a modified I/O automaton notation. In our protocol, **procedures** are input actions that can be invoked at any time (e.g., in a higher level protocol); **functions** are simple methods; and **upon** clauses specify how processes handle external events (i.e., messages, system initialization, and recovery). We use **guards** to prevent actions from being activated under certain conditions. If the **guard** of a message handler or **procedure** is not satisfied, no action is taken, and the message is not consumed (i.e., it remains in the network undelivered).

Each of the n process in Π maintains a *crash vector*, v, with one entry for each process in the system. Entry i in this vector tracks the latest known incarnation ID of process i. During an incarnation, a process numbers its ACQUIRE and READ messages using the local variable c to match messages with replies. When a process recovers, it gets a new value from its local, monotonic clock and updates its incarnation ID in its own vector. When the recovery procedure ends, the process becomes OPERATIONAL and signals this through the *op* flag. A process's crash vector is updated whenever a process learns about a newer incarnation of another process. Crash vectors are partially ordered, and a join operation, denoted $\sqcup$, is defined over vectors, where $(v_1 \sqcup v_2)[i] = \max(v_1[i],\ v_2[i])$. Initially, each process's crash vector is $[\bot, \ldots, \bot]$, where $\bot$ is some value smaller than any incarnation ID.

The Acquire-Quorum function handles both writing values and recovering. Acquire-Quorum ensures the persistence of both the process's current crash vector – in particular the process's own incarnation ID in the vector – as well as the value to be written, val. It provides these guarantees by collecting responses from a quorum of processes and ensuring that those responses are *crash-consistent*. It uses crash vectors to detect when any process that previously replied *could have crashed* and thus could have "forgotten" the written value.

Algorithm 1 Communications primitives.

Permanent Local State:
$n \in \mathbb{N}^+$ ▷ Number of processes
$i \in [1, \ldots, n]$ ▷ Process number

Volatile Local State:
$v \leftarrow [\bot$ **for** $i \in [1, \ldots, n]]$ ▷ Crash vector
$op \leftarrow false$ ▷ Operational flag
$R \leftarrow \{\}$ ▷ Reply set
$c \leftarrow 0$ ▷ Message number

1: **upon** System-Initialize
2: $op \leftarrow true$
3: **end upon**

4: **upon** Recover
5: $v[i] \leftarrow$ Read-Clock
6: $\Sigma \leftarrow$ Acquire-Quorum($null$)
7: Rebuild-State(Σ)
8: $op \leftarrow true$
9: **end upon**

10: **function** Acquire-Quorum(val)
11: $R \leftarrow \{\}$
12: $c \leftarrow c + 1$
13: $m \leftarrow \langle$acquire$\rangle$
14: $m.c \leftarrow c$
15: $m.val \leftarrow val$
16: **for all** $j \in [1, \ldots, n]$ **do**
17: Send-Message(m, j)
18: **end for**
19: Wait until $|R| > n/2$
20: **return** $\{m.s : m \in R\}$
21: **end function**

22: **function** Read-Quorum
23: $R \leftarrow \{\}$
24: $c \leftarrow c + 1$
25: $m \leftarrow \langle$read$\rangle$
26: $m.c \leftarrow c$
27: **for all** $j \in [1, \ldots, n]$ **do**
28: Send-Message(m, j)
29: **end for**
30: Wait until $|R| > n/2$
31: **return** $\{m.s : m \in R\}$
32: **end function**

33: **function** Send-Message(m, j)
34: $m.f \leftarrow i$ ▷ Sender
35: $m.v \leftarrow v$
36: Send m to process j
37: **end function**

38: **upon** receiving $\langle$acquire$\rangle$, m
39: **guard:** op
40: $v \leftarrow v \sqcup m.v$
41: $m' \leftarrow \langle$acquire-rep$\rangle$
42: **if** $m.val \neq null$ **then**
43: Update-State($m.val$)
44: **end if**
45: $m'.s \leftarrow$ Read-State
46: $m'.c \leftarrow m.c$
47: Send-Message(m', $m.f$)
48: **end upon**

49: **upon** receiving $\langle$acquire-rep$\rangle$, m
50: **guard:** $m.v[i] = v[i] \wedge c = m.c$
51: $v \leftarrow v \sqcup m.v$
52: Add m to R
▷ Discard inconsistent, duplicate replies
53: **while** $\exists m' \in R$ **where**
54: $m'.v[m'.f] < v[m'.f]$ **do**
55: Remove m' from R
56: Resend $\langle$acquire$\rangle$ message to $m'.f$
57: **end while**
58: **while** $\exists m', m'' \in R$ **where**
59: $m'.f = m''.f \wedge m' \neq m''$ **do**
60: Remove m' from R
61: **end while**
62: **end upon**
63: **upon** receiving $\langle$read$\rangle$, m
64: **guard:** op
65: $v \leftarrow v \sqcup m.v$
66: $m' \leftarrow \langle$read-rep$\rangle$
67: $m'.s \leftarrow$ Read-State
68: $m'.c \leftarrow m.c$
69: Send-Message(m', $m.f$)
70: **end upon**

71: **upon** receiving $\langle$read-rep$\rangle$, m
72: **guard:** $m.v[i] = v[i] \wedge c = m.c$
73: $v \leftarrow v \sqcup m.v$
74: Add m to R
75: **end upon**

4.4 Correctness

We show that our primitives provide the same safety properties as writing and reading to simple quorums in the Crash-Stop model. First, we formally define quorum knowledge in the DCR context.

▶ **Definition 4** (Stable Properties)**.** A predicate on the history of an incarnation of a process (i.e., the sequence of events it has processed) is a *stable property* if it is monotonic (i.e., X being true of history h implies that X is true of any history with h as a prefix).

▶ **Definition 5.** If stable property X is true of some incarnation of a process, p, we say that incarnation of p *knows* X.

▶ **Definition 6** (Quorum Knowledge). We say that a quorum Q *knows* stable property X if, for all processes $p \in Q$, one of the following holds:
(1) p is DOWN,
(2) p is OPERATIONAL and knows X, or
(3) p is RECOVERING and either already knows X or will know X if and when it finishes recovery.

In our analysis of Algorithm 1, we are concerned with knowledge of two types of stable properties: knowledge of values and knowledge of incarnation IDs. An incarnation of a process knows value *val* if it has either executed UPDATE-STATE(*val*) or executed REBUILD-STATE with an ACQUIRE-REP message in the reply set sent by a process which knew *val*. Knowledge of a process's incarnation ID, i, is the stable property of having an entry in a crash vector for that process greater than or equal to i.

Next, we define crash-consistency on ACQUIRE-REP messages with crash vectors.

▶ **Definition 7** (Crash Consistency). A set of ACQUIRE-REP messages R is *crash-consistent* if $\forall s_1, s_2 \in R.\ s_1.v[s_2.f] \leq s_2.v[s_2.f]$.

Note that Definition 7, phrased in terms of crash vectors, is equivalent to the sending events of the ACQUIRE-REP messages being crash-consistent according to Definition 3.

▶ **Definition 8** (Quorum Promise). We say that a crash-consistent set of ACQUIRE-REP messages constitutes a *quorum promise* for stable property X if the set of senders of those messages is a quorum, and each sender knew X when it sent the message.

▶ **Definition 9.** If process p sent one of the ACQUIRE-REP message belonging to a quorum promise received by some process, we say that p *participated* in that quorum promise.

The post-condition of the loop on line 53 guarantees the crash-consistency of the reply set by discarding any inconsistent messages; the next loop guarantees that there is at most one message from each process in the reply set. Therefore, the termination of ACQUIRE-QUORUM (line 10) implies that the process has received a quorum promise showing that *val* was written and that every participant had a crash vector greater than or equal to its own vector *when it sent the* ACQUIRE *message*. This implies that whenever a process finishes recovery, it must have received a quorum promise showing that the participants in its recovery had that process's latest incarnation ID in their crash vectors.

Unlike having a stable property, that a process *participated* in a quorum promise holds across failures and recoveries. That is, we say that a process, not a specific incarnation of that process, participated in a quorum promise. Also note that only OPERATIONAL processes ever participate in a quorum promise, guaranteed by the guard on the ACQUIRE message handler.

4.4.1 Safety

Finally, we are ready to state the main safety properties of our generic read/write primitives.

▶ **Theorem 10** (Persistence of Quorum Knowledge). *If at time t, some quorum, Q, knows stable property X, then for all times $t' \geq t$, Q knows X.*

Proof. We prove by strong induction on t' that the following invariant, I, holds for all $t' \geq t$: For all p in Q: (1) p is OPERATIONAL and knows X, (2) p is RECOVERING, or (3) p is DOWN. In the base case at time t, Q knows X by assumption, so I holds.

Now, assuming I holds at all times $t' - 1 \geq t$, we show that I holds at time t'. The only step any process $p \in Q$ could take to falsify I is finishing recovery. If recovery began at or before time t, then because Q knew X, p must know X now that it has finished recovering. Otherwise, if it began after time t, then p must have received some set of ACQUIRE-REP messages from a quorum, all of which were sent after time t. By quorum intersection, one of these messages must have come from some process in Q. Call this process q. Since q's ACQUIRE-REP message, m, was sent after time t and before t', by the induction hypothesis, q must have known X when it sent m. Therefore, p must know X upon finishing recovery since it updates its crash vector and rebuilds its state using m.

Since I holds for all times $t' \geq t$, this implies the theorem. ◀

▶ **Theorem 11** (Acquisition of Quorum Knowledge). *If process p receives a quorum promise for stable property X from quorum Q, then Q knows X.*

Proof. We again prove this theorem by (strong) induction, showing that the following invariant, I, holds for all times, t:

1. If a process receives a quorum promise for stable property X from quorum Q, then Q knows X.
2. If process p ever participated in a quorum promise for X at or before time t, and p is OPERATIONAL, then p knows X.

I holds vacuously at $t = 0$. We show that if I holds at time $t - 1$, it holds at time t:

First, we consider part 1 of I. If p has received a quorum promise, R, from quorum Q for X, then because R is crash-consistent, we know that *at the time they participated in* R no process in Q had participated in the recovery of any later incarnation of any other process in Q than the one that participated in R. If they had, then by the induction hypothesis, such a process would have known the recovered process's new incarnation ID when it participated in R, and R would not have been crash-consistent.

Given that fact, we will use a secondary induction to show that for all times, t', all of the processes in Q either:

(1) haven't yet participated in R,
(2) are DOWN,
(3) are RECOVERING, or
(4) are OPERATIONAL and know X.

In the base case, no process in Q has yet participated in R. For the inductive step, note that the only step any process q could take that would falsify our invariant is transitioning from RECOVERING to OPERATIONAL after having participated in R. If q finished recovering, it must have received a quorum promise showing that the senders knew its new incarnation ID. By quorum intersection, at least one of these came from some process $r \in Q$. We already know r couldn't have participated in q's recovery before participating in R. So by the induction hypothesis, r knew X at the time it participated in q's recovery. Because knowledge of values and incarnation IDs is transferred through ACQUIRE-REP messages, q knows X, completing this secondary induction.

Finally, we know that since p has received R at time t, all of the process in Q have already participated in R, so all of the processes in Q are either DOWN, RECOVERING (and will know X upon finishing recovery), or are OPERATIONAL and know X. Therefore, Q knows X, and this completes the proof that part 1 of I holds at time t.

Now, we consider part 2 of I. Suppose, for the sake of contradiction, that p is OPERATIONAL at time t and doesn't know X, but participated in quorum promise R for X at or before

time t. Let Q be the set of processes participating in R. Since p does not know X, p must have crashed and recovered since participating in R. Consider p's most recent recovery, and let the quorum promise it received showing that the senders knew p's new incarnation ID (or a greater one) be R'. Let the set of participants in R' be Q'. By quorum intersection, there exists some $r \in Q \cap Q'$.

It must be the case that r participated in R' before R; otherwise by induction, when r participated in R', it would have known X, and then transferred that knowledge to the current incarnation of p (at time t). r couldn't have participated in R before time t, because then by part 2 of I, it would have known p's latest incarnation ID when participating in R, violating the consistency of R. However, r cannot participate in R at or after time t, either. Because p has received a quorum promise for its new incarnation ID at or before time t, by part 1 of I, Q' knows p's new incarnation ID. By Theorem 10, Q' continues to know this at all later times. Because $r \in Q'$, it must know p's incarnation ID, and thus cannot participate in R without violating its crash-consistency. This contradicts the fact that r participates in R and completes the proof that part 2 holds at time t. ◀

Since ACQUIRE-QUORUM(val) obtains a quorum promise for val, Theorem 11 implies quorum knowledge of val, and Theorem 10 shows that that knowledge will persist for all future time, subsequent ACQUIRE-QUORUMs and READ-QUORUMs will get a response from a process which knows val.

4.4.2 Liveness

ACQUIRE-QUORUM and READ-QUORUM terminate if there is some quorum of processes that all remain OPERATIONAL for a sufficient period of time.[2] This is easy to see since a writing or recovering process will eventually get an ACQUIRE-REP from each of these OPERATIONAL processes, and those replies must be crash-consistent. Note that the termination of ACQUIRE-QUORUM implies the termination of the recovery procedure, RECOVER. Therefore, the same liveness conditions are required for ACQUIRE-QUORUM and for recovery termination.

We define a sufficient liveness condition, LC, below. It is a slightly weaker version of the network stability condition N_2 from [17]: the period in which processes must remain OPERATIONAL is shorter.

▶ **Definition 12** (Liveness Condition (LC)). Consider a process p executing either the ACQUIRE-QUORUM or READ-QUORUM function, ϕ, and consider the following statements:

1. There exists a quorum of processes, Q, all of which consume their respective messages sent from ϕ.
2. Every process in Q either
   a. remains OPERATIONAL during the interval $[T_1, T_2]$, where T_1 is the point in time at which ϕ was invoked and T_2 the earliest point in time at which p completes the consumption of all the responses sent by the processes in Q or
   b. becomes DOWN and remains DOWN during the same interval after p consumed its response.

If these two statements are true for *every* invocation of a ACQUIRE-QUORUM or READ-QUORUM function, then we say that LC is satisfied.

[2] We assume that the application-provided READ-STATE, UPDATE-STATE, and REBUILD-STATE functions execute entirely locally and do not block.

Our protocol implementing the group communication primitives is live if LC is satisfied. For LC to be satisfied, it is necessary that at most a minority of processes are DOWN *at any given time*. Otherwise, no process can ever receive replies from a quorum again.

5 Recoverable Shared Objects in DCR

In this section we demonstrate the benefits of our quorum communication primitives for DCR: generality and efficiency. We present protocols for two different shared objects: a multi-writer, multi-reader (MWMR) atomic register and a single-writer, single-reader (SWSR) atomic set. In both protocols, READ and WRITE are intended to be invoked serially.

The first protocol implements a shared, fault-tolerant MWMR atomic register in DCR. It is more efficient and has better liveness conditions than prior work. The second protocol implements a weaker abstraction – a shared, fault-tolerant SWSR atomic set. We use this set as a basic storage primitive to provide processes with access to their own *virtual stable storage* (VSS), an emulation of a local disk. This enables easy migration of protocols to DCR.

5.1 Multi-writer, Multi-reader Atomic Register

We present a protocol for implementing a fault-tolerant, recoverable multi-writer, multi-reader (MWMR) atomic register in DCR, which guarantees the linearizability of READs and WRITEs [19]. Our protocol is similar to the ABD protocol [3] but augments it with a recovery procedure. Its pseudo-code is presented in Algorithm 2. Timestamps are used for version control, as in the original protocol. A timestamp is defined as a triple $(z, i, v[i])$, where $z \in N$, $i \in [1..n]$ is the ID of the writing process, and $v[i]$ is the incarnation ID of that process. Timestamps are ordered lexicographically. By replacing each quorum write phase in the original protocol with our ACQUIRE-QUORUM function and each quorum read phase with READ-QUORUM, we guarantee that every successful write phase is visible to subsequent read phases, despite concurrent crashes and recoveries, thus preserving safety in DCR. The REBUILD-STATE function reconstructs a value of the register at least as new as the one of the last successful write that finished before the process crashed.

Discussion. The most recent protocol for fault-tolerant, recoverable, MWMR atomic registers is *RADON* [17]. The always-safe version of *RADON*, $RADON_R^{(S)}$, introduces an additional communication phase after each quorum write to check whether any of the processes that acknowledged the write crashed in the meantime. This increases the latency of both the READ and WRITE procedures. Also, our liveness conditions are weaker: our protocol is live if any majority of processes do not crash for a sufficient period of time, while $RADON_R^{(S)}$ requires a supermajority (3/4) of processes to not crash.

5.2 Virtual Stable Storage

Algorithm 3 presents a protocol for a fault-tolerant, recoverable, SWSR set, where the reader is the same as the writer. It guarantees that the values written by completed WRITEs and those returned in READs are returned in subsequent READs. Given the group communication primitives, its implementation is straightforward; the only additional detail is that values read during recovery should be written back to ensure atomicity (line 17).

Discussion. We can use this set to provide a *virtual stable storage* abstraction. It is well known that any correct protocol in CS can be transformed into a correct protocol in the

Algorithm 2 Multi-writer, multi-reader atomic register in Diskless Crash-Recovery.

```
     Volatile Local State:
         (t, d) ← (t_0, d_0) ▷ Value of register

 1:  procedure WRITE(d_new)
 2:  guard: op
     ▷ Get latest timestamp
 3:      Σ ← READ-QUORUM
 4:      (t_max, d_max) ← max(Σ)
     ▷ Write value
 5:      t_new ← (t_max.z + 1, i, v[i])
 6:      ACQUIRE-QUORUM((t_new, d_new))
 7:  end procedure

 8:  procedure READ
 9:  guard: op
     ▷ Get latest register value
10:      Σ ← READ-QUORUM
11:      (t_max, d_max) ← max(Σ)
     ▷ Write latest register value
12:      ACQUIRE-QUORUM((t_max, d_max))
13:      return d_max
14:  end procedure

15:  function UPDATE-STATE(val)
16:      if val.t > t then
17:          (t, d) ← val
18:      end if
19:  end function

20:  function READ-STATE
21:      return (t, d)
22:  end function

23:  function REBUILD-STATE(Σ)
24:      (t, d) ← max(Σ)
25:  end function
```

Algorithm 3 Single writer, single reader atomic set in Diskless Crash-Recovery.

```
     Permanent Local State:
         owner ▷ Owner of set flag

     Volatile Local State:
         S ← {}          ▷ Local set

 1:  procedure WRITE(s)
 2:  guard: op ∧ owner
 3:      ACQUIRE-QUORUM({s})
 4:      S ← S ∪ {s}
 5:  end procedure

 6:  procedure READ
 7:  guard: op ∧ owner
 8:      return S
 9:  end procedure

10:  function UPDATE-STATE(val)
11:      S ← S ∪ val
12:  end function

13:  function READ-STATE
14:      return S
15:  end function

16:  function REBUILD-STATE(Σ)
17:      ACQUIRE-QUORUM(Σ)
18:      S ← ⋃ Σ
19:  end function
```

CRSS model by having processes write every message they receive (or the analogous state update) to their local disk before sending a reply. By equipping each process with VSS, any correct protocol in the CRSS model can then be converted into a safe protocol in the DCR model, wherein processes write to crash-consistent quorums instead of stable storage.

6 Recoverable Replicated State Machines in DCR

We further extend our study of DCR to another specific problem: state machine replication (SMR). SMR is a classic approach for building fault-tolerant services [18, 32] that calls for the service to be modeled as a deterministic state machine, replicated over a group of replicas. System correctness requires each replica to execute the same set of operations in the same order, even as replicas and network links fail. This is typically achieved using a consensus-based replication protocol such as Multi-Paxos [20] or Viewstamped Replication [22, 29] to establish a global order of client requests.

We examined three diskless recovery protocols for SMR: Viewstamped Replication [22], Paxos Made Live [7], and JPaxos [16]. We found that each of these protocols suffers from the problem illustrated in the example at the beginning of Section 4: they use regular quorums of responses (instead of crash-consistent ones) when persisting critical data, which could violate their invariants. This can lead to operations being lost, or different operations being executed at different replicas, both serious correctness violations. We provide here brief explanations of the problems in each of these protocols. For more details on how the protocols work and complete traces, see our technical report [27].

Viewstamped Replication [29] is the first consensus-based SMR protocol. The original version of the protocol requires a single write to disk, during a view change. A recent VR variant [22] replaces the write to disk with a write to a quorum of replicas, in an attempt to eliminate the necessity for disks. However, it uses simple quorum responses, allowing the recovering replica to violate an important invariant: once a replica committed to take part in a new view, it will never operate in a lower view. As a result, an operation can complete successfully and then be lost after a view change.

Paxos Made Live [7] is Google's Multi-Paxos implementation. To handle corrupted disks, it lets a replica rejoin the system without its previous state and runs an (unspecified) recovery protocol to restore the application state. The replica must then wait to observe a full instance of successful consensus before participating. This successfully prevents the replica from accepting multiple values for the same instance (e.g., one before and one after the crash). However, it does *not* prevent the replica from sending different promises (i.e., leader change commitments) to potential new leaders, which can lead to a new leader deciding a new value for a prior successful instance of consensus.

JPaxos [16], a hybrid of Multi-Paxos and VR, provides a variety of deployment options, including a diskless one. Nodes in JPaxos maintain an *epoch vector* that tracks which nodes have crashed and recovered to discard lost promises made by prior incarnations of recovered nodes. However, like VR and PML, certain failures during node recovery can cause the system to lose state and violate safety properties.

All of these protocols can be correctly migrated to DCR, with little effort, using VSS write operations, as explained in Section 5.2. This approach is straightforward, efficient, and requires no invasive protocol modifications.

7 Conclusion

This paper examined the Diskless Crash-Recovery model, where process can crash and recover but lose their state. We show how to provide persistence guarantees in this model using new quorum primitives that write to and read from *crash-consistent quorums*. These general primitives allow us to construct shared objects in the DCR model. In particular, we show a MWMR atomic register protocol requiring fewer communication rounds and weaker liveness assumptions than the best prior work. We also build a SWSR atomic set that can be used to provide each process with *virtual stable storage*, which can be used to easily migrate any protocol from traditional Crash-Recovery models to DCR.

Acknowledgments. We thank Marcos K. Aguilera for his comments on early drafts of this work, as well as Irene Zhang, the anonymous reviewers, and Jennifer L. Welch for their helpful feedback.

References

1 Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic atomic storage without consensus. *J. ACM*, 58(2):7:1–7:32, April 2011.

2 Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Failure detection and consensus in the crash-recovery model. In *Proc. of DISC*, 1998.

3 Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. of the ACM*, 42(1):124–142, January 1995.

4 Hagit Attiya, Hyun Chul Chung, Faith Ellen, Saptaparni Kumar, and Jennifer L. Welch. Simulating a shared register in an asynchronous system that never stops changing. In *Proc. of DISC*, 2015.

5 Roberto Baldoni, Silvia Bonomi, Anne-Marie Kermarrec, and Michel Raynal. Implementing a register in a dynamic distributed system. In *Proc. of ICDCS*, 2009.

6 Roberto Baldoni, Silvia Bonomi, and Michel Raynal. Implementing a regular register in an eventually synchronous distributed system prone to continuous churn. *IEEE Trans. Parallel Distrib. Syst.*, 23(1):102–109, January 2012.

7 Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *Proc. of PODC*, 2007.

8 Danny Dolev, Roy Friedman, Idit Keidar, and Dahlia Malkhi. Failure detectors in omission failure environments. In *Proc. of PODC*, 1997.

9 Colin J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proc. of ACSC*, 1988.

10 Eli Gafni and Dahlia Malkhi. Elastic configuration maintenance via a parsimonious speculating snapshot solution. In *Proc. of DISC*, 2015.

11 Rachid Guerraoui, Ron R. Levy, Bastian Pochon, and Jim Pugh. The collective memory of amnesic processes. *ACM Trans. Algorithms*, 4(1):12:1–12:31, March 2008.

12 Michel Hurfin, Achour Mostéfaoui, and Michel Raynal. Consensus in asynchronous systems where processes can crash and recover. In *Proc. of SRDS*, 1998.

13 Leander Jehl, Tormod Erevik Lea, and Hein Meling. Replacement: Decentralized failure handling for replicated state machines. In *Proc. of SRDS*, 2015.

14 Leander Jehl, Roman Vitenberg, and Hein Meling. SmartMerge: A new approach to reconfiguration for atomic storage. In *Proc. of DISC*, 2015.

15 Andreas Klappenecker, Hyunyoung Lee, and Jennifer L. Welch. Dynamic regular registers in systems with churn. *Theor. Comput. Sci.*, 512:84–97, November 2013.

16 Jan Kończak, Nuno Santos, Tomasz Żurkowski, Paweł T. Wojciechowski, and André Schiper. JPaxos: State machine replication based on the Paxos protocol. Technical Report EPFL-REPORT-167765, 2011.

17 Kishori M. Konwar, N. Prakash, Nancy A. Lynch, and Muriel Médard. RADON: Repairable atomic data object in networks. In *Proc. of OPODIS*, 2016.

18 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of the ACM*, 1978.

19 Leslie Lamport. On interprocess communication. *Distributed Computing. Parts I and II*, 1986.

20 Leslie Lamport. Paxos made simple. *ACM SIGACT News 32*, 2001.

21 Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Reconfiguring a state machine. *SIGACT News*, 41(1):63–73, March 2010.

22 Barbara Liskov and James Cowling. Viewstamped Replication revisited. Technical report, MIT, July 2012.

23 Jacob R. Lorch, Atul Adya, William J. Bolosky, Ronnie Chaiken, John R. Douceur, and Jon Howell. The SMART way to migrate replicated stateful services. In *Proc. of EuroSys*, 2006.

24 Nancy A. Lynch and Alexander A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Proc. of DISC*, 2002.

25 Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.

26 Michael Merritt and Gadi Taubenfeld. Computing with infinitely many processes under assumptions on concurrency and participation. In *Proc. of DISC*, 2000.

27 Ellis Michael, Dan R. K. Ports, Naveen Kr. Sharma, and Adriana Szekeres. Recovering shared objects without stable storage [extended version]. Technical Report UW-CSE-17-08-01, University of Washington CSE, August 2017.

28 Peter Musial, Nicolas Nicolaou, and Alexander A. Shvartsman. Implementing distributed shared memory for dynamic networks. *Comm. of the ACM*, 57(6), June 2014.

29 Brian M. Oki and Barbara H. Liskov. Viewstamped Replication: A new primary copy method to support highly-available distributed systems. In *Proc. of PODC*, 1988.

30 R.C. Oliveira, R. Guerraoui, and A. Schiper. Consensus in the crash recover model. Technical Report TR-97/239, EPFL, Lausanne, Switzerland, 1997.

31 D.S. Parker, G.J. Popek, G. Rudisin, A. Stoughton, B.J. Walker, E. Walton, J.M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. In *IEEE Trans. on Software Engineering*, 1983.

32 Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 1990.

33 Alexander Shraer, Benjamin Reed, Dahlia Malkhi, and Flavio P. Junqueira. Dynamic reconfiguration of primary/backup clusters. In *Proc. of USENIX ATC*, 2012.

Dalí: A Periodically Persistent Hash Map

Faisal Nawab*[1], Joseph Izraelevitz*[2], Terence Kelly*[3], Charles B. Morrey III*[4], Dhruva R. Chakrabarti*[5], and Michael L. Scott[6]

1 **University of California, Santa Barbara, CA, USA**
nawab@cs.ucsb.edu
2 **University of Rochester, NY, USA**
jhi1@cs.rochester.edu
3 **Palo Alto, CA, USA**
4 **Woodside, CA, USA**
5 **Palo Alto, CA, USA**
dhruvac@gmail.com
6 **University of Rochester, NY, USA**
scott@cs.rochester.edu

Abstract

Technology trends suggest that byte-addressable nonvolatile memory (NVM) will supplant many uses of DRAM over the coming decade, raising the prospect of inexpensive recovery from power failures and similar faults. Ensuring the consistency of persistent state remains nontrivial, however, in the presence of volatile caches; cached values can "leak" back to persistent memory in arbitrary order. To ensure consistency, existing persistent memory algorithms use expensive, explicit write-back instructions to force each value back to memory before performing a dependent write, thereby incurring significant run-time overhead.

To reduce this overhead, we present a new design paradigm that we call *periodic persistence*. In a periodically persistent data structure, updates are made "in place," but can safely leak back to memory *in any order*, because only those updates that are known to be valid will be heeded during recovery. To guarantee forward progress, we periodically force a write-back of all dirty data in the cache, ensuring that all "sufficiently old" updates have indeed become persistent, at which point they become semantically visible to the recovery process.

As an example of periodic persistence, we present a transactional hash map, Dalí, together with an informal proof of safety (buffered durable linearizability). Experiments with a prototype implementation suggest that periodic persistence can offer substantially better performance than either file-based or incrementally persistent (per-access write-back) alternatives.

1998 ACM Subject Classification D.3.3 Concurrent Programming Structures

Keywords and phrases data structure, nonvolatile memory, durable linearizability

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.37

1 Introduction

For decades, programmers have been accustomed to partitioning program state into *memory*, which is transient – used during a single program run – and *storage*, which is persistent –

* This work was supported in part by the US Department of Energy under Cooperative Agreement no. DE-SC0012199 while the indicated authors were members of Hewlett Packard Labs. At the University of Rochester, the work was supported in part by NSF grants CNS-1319417, CCF-1337224, and CCF-1422649, and by a Google Faculty Research award.

© Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott;
licensed under Creative Commons License CC-BY

intended for use across program runs and even system crashes. The design of data structures is rooted in the use of memory; data in storage is typically relegated to a file or database.

Since the 1970s, memory has been virtually synonymous with DRAM, accessed (since the 1980s) through a rich hierarchy of caches. Storage has been the province of magnetic disks or, more recently, flash drives. Several new memory technologies, however, promise to provide byte-addressable nonvolatile memory (NVM) with access latencies and costs comparable to those of DRAM. These technologies provide the opportunity to re-think the memory–storage divide, and to entertain the possibility of maintaining traditional in-memory data structures across program runs and crashes.

We are particularly interested in crashes, as they present unique consistency challenges. For simplicity, and in keeping with the real-world common case, we assume a "whole system crash" failure model (caused, for example, by power failure or an OS kernel panic). We wish to ensure, in the wake of a crash, that data in memory are consistent. At first blush, it is tempting to model this goal as a conventional concurrency problem: "normal" execution entails one or more threads performing atomic updates to the data; a *recovery procedure* runs in the wake of a crash and (since the crash can occur at any time) functions as if it were merely an additional concurrent thread (with the possible simplifying assumption that it runs in isolation).

The problem with this model is that the recovery procedure does not have access to the view of memory shared by threads during normal execution. Caches are likely to remain volatile, at least for the foreseeable future, so what the recovery procedure sees is whatever has been written back to nonvolatile memory prior to the crash. Unfortunately, hardware capacity and associativity constraints require that caches be permitted to perform their writes-back in essentially arbitrary order. When this order differs from the happens-before order of the running program, the values that happen to have "leaked back" to memory at any particular time may be mutually inconsistent. If, for example, a program creates an object and then aims a pointer at it, it is possible for the pointer to reach memory before the object to which it points. Persistent data structures must be carefully designed to avoid this sort of problem.

In current real-world processors, instructions to control the ordering, timing, and granularity of writes-back from caches to memory are rather limited. On Intel processors, for example, the CLFLUSH instruction [16] takes an address as argument, and blocks until the cache line containing the address has been both evicted from the cache and written back to the memory controller. When combined with an MFENCE instruction to prevent compiler and processor instruction reordering, CLFLUSH allows the programmer to force a write-back that is guaranteed to *persist* (reach nonvolatile memory) before any subsequent store. The overhead is substantial, however – on the order of hundreds of cycles. Future processors may provide less expensive persistence instructions, such as the `pwb`, `pfence`, and `psync` assumed in our earlier work [17], or the `ofence` and `dfence` of Nalli et al. [21]. Even in the best of circumstances, however, "persisting" an individual store (and ordering it relative to other stores) is likely to take time comparable to a memory consistency fence on current processors – i.e., tens of cycles. Due to power constraints [8], we expect that writes and flushes into NVM will be guaranteed to be failure-atomic only at increments of eight bytes – not across a full 64-byte cache line.

We use the term *incremental persistence* to refer to the strategy of persisting store w_1 before performing store w_2 whenever w_1 occurs before w_2 in the happens-before order of the program during normal execution (i.e., when $w_1 <_{hb} w_2$). Given the expected latency of

even an optimized persist, this strategy seems doomed to impose significant overhead on the operations (method calls) of any data structure intended to survive program crashes.

As an alternative, we introduce a strategy we refer to as *periodic persistence*. The key to this strategy is to design a data structure in such a way that modifications can safely leak into persistence *in any order*, removing the need to persist locations incrementally and explicitly as an operation progresses. To ensure that an operation's stores eventually become persistent, we periodically execute a *global fence* that forces all cached data to be written back to memory. The interval between global fences bounds the amount of work that can ever be lost in a crash (though some work may be lost). To avoid depending on the fine-grain ordering of writes-back, we arrange for "leaked" lines to be ignored by any recovery procedure that executes before a subsequent global fence. After the fence, however, a known set of cache lines will have been written back, making their contents safe to read. Like naive uninstrumented code, periodic persistence allows stores to persist out of order. It guarantees, however, that the recovery procedure will never use a value v from memory unless it can be sure that all values on which v depends have also safely persisted.

In contrast to checkpointing, which creates a consistent *copy* of data in nonvolatile memory, periodic persistence maintains a *single* instance of the data for both the running program and the recovery procedure. This single instance is designed in such a way that recent updates are nondestructive, and the recovery procedure knows which parts of the data structure it can safely use.

In some sense, periodically persistent structures can be seen as an adaptation of traditional *persistent data structures* [12] (in a different sense of the word "persistent") or of multiversion transactional memory systems [3], both of which maintain a history of data structure changes over time. In our case, we can safely discard old versions that predate the most recent global fence, so the overall impact on memory footprint is minimal. At the same time, we must ensure not only that the recovery procedure ignores the most recent updates but also that it is never confused by their potential structural inconsistencies.

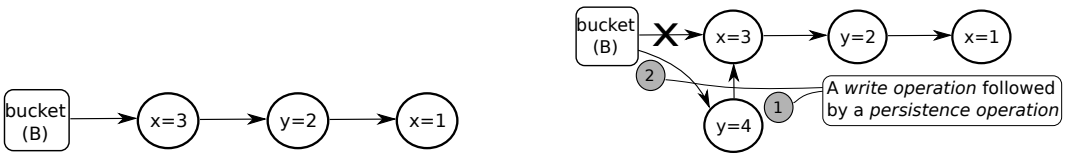
As an example of periodic persistence, we introduce Dalí,[1] a transactional hash map for nonvolatile memory. Dalí demonstrates the feasibility of using periodic persistence in a nontrivial way. Experience with a prototype implementation confirms that Dalí can significantly outperform alternatives based on either incremental or traditional file-system-based persistence. Our prototype implements the global fence by flushing (writing back and invalidating) all on-chip caches. Performance results would presumably be even better with hardware support for whole-cache write-back without invalidation.

The remainder of this paper is organized as follows: Section 2 elaborates on the motivation for our work in the context of persistent hash maps. We describe Dalí's design in Section 3 and prove its correctness in Section 4. Section 5 then presents experimental results. Section 6 reviews related work. Section 7 summarizes our conclusions.

2 Motivation

As a motivating example, consider the construction of a persistent hash map, beginning with the nonblocking structure of Schwalb et al. [24]. To facilitate transactional update of entries in multiple buckets, we switch to a blocking design with a lock in each bucket, enabling the use of two-phase locking (and, for atomicity in the face of crashes, undo logging).

[1] The name is inspired by Dalí's painting *The Persistence of Memory*.



(a) A bucket containing three records.

(b) An example of the write-ordering overhead entailed in updating a data object.

Figure 1 A hash map data structure that demonstrates the overhead of write ordering.

This hash map, which is incrementally persistent, consists of an array of buckets, each of which points to a singly-linked list of *records*. Each record is a key-value pair. Figure 1a shows a bucket with three records. For the sake of simplicity, each list is prepend-only: records closer to the head are more recent. It is possible that multiple records exist for the same key – the figure shows two records for the key x, for instance, but only the most recent record is used. Deletions are handled by inserting a "not present" record. Garbage collection / compaction can be handled separately; we omit the description here.

Figure 1b shows an update to change the value of y to 4. The update comprises several steps: (1a) A record, r_{new} with the new key-value pair is written. The record points to the current head of the list. (1b) A persist of r_{new} serves to push its value from cache to NVM. (2a) The bucket list head pointer, B, is overwritten to point to r_{new}. (2b) A second persist pushes B to NVM. The first persist must complete before the store to B: it prevents the incorrect recovery state in which r_{new} is not in NVM and B is a dangling pointer. The second persist must complete before the operation that updates y returns to the application program: it prevents misordering with respect to subsequent operations.

On current hardware, a persist operation waits hundreds of cycles for a full round trip to memory. On future machines, hardware support for ordered (queued) writes-back might reduce this to tens of cycles. Even so, incremental persistence can be expected to increase the latency of simple operations several-fold. The key insight in Dalí is that when enabled by careful data structure design, periodic persistence can eliminate fine-grain ordering requirements, replacing a very large number of single-location fences with a much smaller number of global fences, for a large net win in performance, at the expense of possible lost work. In practice, we would expect the frequency of global fences to reflect a trade-off between overhead and the amount of work that may be lost on a crash. Fencing once every few milliseconds strikes us as a good initial choice.

3 Dalí

Dalí is our prepend-only transactional hash map designed using periodic persistence. It can be seen as the periodic persistence equivalent of the incrementally persistent hash map of Section 2 and Figure 1. As a transactional hash map, Dalí supports the normal `get`, `set`, `delete`, and `replace` methods. It also supports ACID transactions comprising any number of the above methods.

Dalí updates or inserts by prepending a record to the appropriate bucket; the most recent record for a key is the one closest to the head of the list (duplicates may exist, but only the most recent record matters). Records in a bucket are from time to time consolidated to remove obsolete versions. Dalí employs per-bucket locks (mutexes) for isolation. A variant of strong strict two-phase locking (SS2PL) is used to implement transactions.

```
class node:
  key k; val v
  node* next
class bucket:
  mutex lock
  int stat<a, f, c, ss> // 2/2/2/58 bits
  node* ptrs[3]
class dali:
  bucket buckets[N_BUCKTS]
  int list flist
  int epoch
```

Figure 2 Dalí globals and data types.

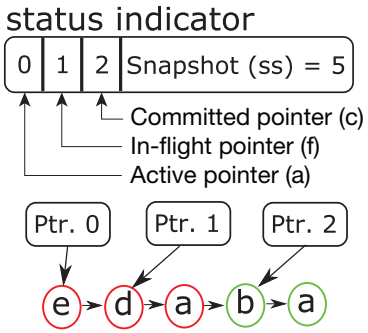


Figure 3 The structure of a bucket.

3.1 Data Structure Overview

As mentioned above, Dalí uses a periodic global fence to guarantee that changes to the data structure have become persistent. The fence is invoked by a special worker thread in parallel with normal operation by application threads. We say that the initiation points of the global fences divide time into *epochs*, which are numbered monotonically from the beginning of time (the numbers do not reset after a crash). Each update (or transactional set of updates) is logically confined to a single epoch, and the fence whose initiation terminates epoch E serves to persist all updates that executed in E. The execution of the fence, however, may overlap the execution of updates in epoch $E+1$. The worker thread does not initiate a global fence until the previous fence has completed. As a result, in the absence of crashes, we are guaranteed during epoch $E+1$ that any update executed in epoch $E-1$ has persisted. If a crash occurs in epoch F, however, updates from epochs F and $F-1$ cannot be guaranteed to be persistent, and should therefore be ignored. We refer to epochs F and $F-1$ as *failed epochs*, and revise our invariant in the presence of crashes to say that during a given epoch E, all updates performed in a non-failed epoch prior to $E-1$ have persisted. Failed epoch numbers are maintained in a persistent *failure list* that is updated during the recovery procedure.

In Dalí, hash map records are classified according to their persistence status. Assume that we are in epoch E. *Committed records* are ones that were written in a non-failed epoch at or before epoch $E-2$. *In-flight records* are ones that were written in epoch $E-1$ if it is not a failed epoch. *Active records* are ones that were written during the current epoch E. Records that were written in a failed epoch are called *failed records*. By steering application threads around failed records, Dalí ensures consistency in the wake of a crash.

Dalí's hash map buckets are similar in layout to those of the incrementally persistent hash map presented in Figure 1. Dalí adds metadata to each bucket, however, to track the persistence status of the bucket's records. The metadata in turn allows us to avoid persisting records incrementally. Specifically, a Dalí bucket contains not only a singly-linked list of records, but also a 64-bit *status indicator* and, in lieu of a head pointer for the list of records, a set of three *list pointers* (see pseudocode in Figure 2 and illustration in Figure 3). The status indicator comprises a *snapshot* (*SS*) field, denoting the epoch in which the most recent record was prepended to the bucket, and three 2-bit *role IDs*, which indicate the roles of the three list pointers. A single STORE suffices to atomically update the status indicator on today's 64-bit machines.[2]

[2] With 6 bits devoted to role IDs, 58 bits remain for the epoch number. If we start a new epoch every millisecond, roll-over will not happen for 9 million years.

Each of the three list pointers identifies a record in the bucket's list (or NULL). The pointers assume three roles, which are identified by storing the pointer number (0, 1, or 2) in one the three role ID fields of the status indicator. Roles are fixed for the duration of an epoch but can change in future epochs. The roles are:

Active pointer (a): provided that epoch SS has not failed, identifies the most recently added record (which must necessarily have been added in SS). Each record points to the record that was added before it. Thus, the active pointer provides access to the entire list of records in the bucket.

In-flight pointer (f): provided that epochs SS and $SS-1$ have not failed, identifies the most recent record, if any, added in epoch $SS-1$. If no such record exists, the in-flight role ID is set to invalid ($\perp$).

Committed pointer (c): identifies the most recent record added in a non-failed epoch equal to or earlier than $SS-2$.

To establish these invariants at start-up, we initialize the global `epoch` counter to 2 and, in every bucket, set SS to 0, all pointers to NULL, the in-flight role ID to $\perp$, and the active and committed IDs to arbitrary values.

Figure 3 shows an example bucket. In the figure SS is equal to 5, which means that the most recent record was prepended during epoch 5. The active pointer is Pointer 0. It points to record e, which means that e was added in epoch 5, even if we are reading the status indicator during a later epoch. Pointer 1 is the in-flight pointer, which makes d the most recently added record in epoch 4. Because a record points only to records that were added before it, by transitivity, records a, b, and the prior a were added before or during epoch 4. Finally, Pointer 2 is the committed pointer. This makes record b the most recently added record before or during epoch 3. By transitivity, the earlier record a was also added before or during epoch 3. Both record b and the earlier record a are therefore guaranteed persistent (shown in green) as of the most recent update (the time at which e was added), while the remainder of the records may not be persistent (shown in red).

It is important to note that the status indicator reflects the bucket's state at SS (the epoch of the most recent update to the bucket) even if a thread inspects the bucket during a later epoch. For example, suppose that a thread in epoch 10 reads the bucket state shown in Figure 3. Given the status indicator, the thread will conclude that all records were written during or before epoch 5 and thus are all committed and persistent (assuming that epochs 4 and 5 are not in the failure list). If one or both epochs are on the failure list, the thread can navigate around their records using the in-flight or committed pointers.

3.2 Reads

The task of the read method is to return the value, if any, associated with a given key. A reader begins by using a hash function to identify the appropriate bucket for its key, and locks the bucket. It then consults the bucket's epoch number (SS) and the global failed epoch list to identify the most recent, yet valid, of the three potential pointers into the bucket's linked list (Figure 4). Call this pointer the *valid head*. If SS is not a failed epoch, the valid head will be the active pointer, which will identify the most recently added record (which may or may not yet be persistent). If SS is a failed epoch but $SS-1$ is not, the valid head will be the in-flight pointer. If SS and $SS-1$ are both failed epochs, the valid head will be the committed pointer.

Starting from the valid head, a reader searches records in order looking for a matching key. Because updates to the hash map are prepends, the most recent matching record will

```
// Bucket is assumed locked via SS2PL
val bucket::read(key k):
  node* valid_head =
    if ss ∉ flist then ptrs[a]
    elsif ss-1 ∉ flist && f ≠ ⊥ then ptrs[f]
    else ptrs[c]
  return search(k, valid_head)
```

Figure 4 Dalí read method.

```
// Bucket is assumed locked via SS2PL
void bucket::update(key k, val v):
  bool curr_fail = ss ∈ flist
  bool prev_fail =
      ss-1 ∈ flist || f == ⊥
  node* valid_head =
    if !curr_fail then ptrs[a]
    elsif !prev_fail then ptrs[f]
    else ptrs[c]
  node* n = new node(k, v, valid_head)

  // Get new pointer roles from table
  int new_stat = lookup(epoch,
      curr_fail, prev_fail, stat)
  ptrs[new_stat.a] = n
  stat = new_stat
```

Figure 5 Dalí update method.

	SS	SS ∈ flist	SS −1 ∈ flist or f = ⊥	new a	new f	new c
1	E	N/A	N/A	a	f	c
2	$E-1$	✗	✗	c	a	f
3	$E-1$	✗	✓	f	a	c
4	$E-1$	✓	N/A	a	⊥	c
5	$< E-1$	✗	N/A	c	⊥	a
6	$< E-1$	✓	✗	a	⊥	f
7	$< E-1$	✓	✓	a	⊥	c

Figure 6 Lookup table for pointer role assignments. Current epoch is E.

be found first. If the key has been removed, the matching value may be NULL. If the key is not found in the list, the value returned from the read will also be NULL.

3.3 Updates

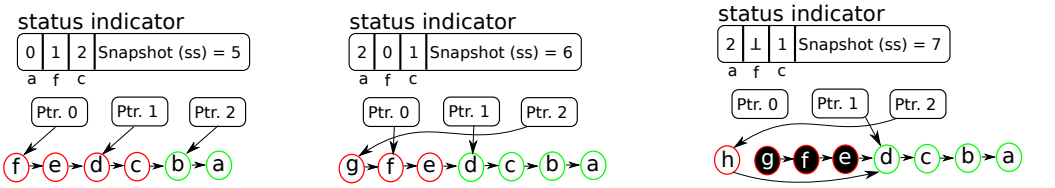
Updates in Dalí prepend a new version of a record, as in the incrementally persistent hash map of Section 2. Deletions / overwrites of existing keys and inserts of new keys are processed identically by a unified `update` method. Like the `read` method, `update` locks the bucket. An update to a Dalí bucket comprises several steps:

1. Determine the most recent, valid pointer (as in the `read` method).
2. Create a new record with the key and its new value (or NULL if a `remove`).
3. Determine the new pointer roles (if the new and old epochs are different).
4. Retarget the new active pointer to the new record node.
5. Update *SS* and the role IDs by overwriting the status indicator.

Pseudocode appears in Figure 5.

Step 3 is the most important part of the update algorithm, as it is the part that allows the update's component writes to be reordered. The problem to be addressed is the possibility that writes from neighboring epochs might be written back and become mixed in the persistent state. We might, for example, mix the snapshot indicator from the later epoch with the pointer values from the earlier epoch. Given any combination of update writes from bordering epochs, and an indication of epoch success or failure, the `read` procedure must find a correct and valid head, and the list beyond that head must be persistent.

The details of step 3 appear in Figure 6. They are based on the following three rules. First, the new committed pointer was last written at least two epochs prior, guaranteeing



(a) Initial state in epoch 5. **(b)** Adding record g in epoch 6. **(c)** Adding record h in epoch 7; epochs 5 and 6 have failed.

Figure 7 A sequence of Dalí updates.

that its value and target have become persistent (and would survive a crash in the current epoch). Second, the new active pointer was either previously invalid or pointed to an earlier record than the new committed pointer. In other words, according to both the old and new status indicators, the new active pointer will never be a valid head, so it is safe to reassign. Third, the new in-flight pointer is the most recent valid record set in the previous epoch, or $\perp$ if no such record exists. These rules are sufficient to enumerate all entries in the table.

Because each bucket is locked throughout the update method, there is no concern about simultaneous access by other active threads. We assume that each of the two key writes in an update – to a pointer and to the status indicator – is atomic with respect to crashes, but the order in which these two writes persist is immaterial: neither will be inspected in the wake of a crash unless the global epoch counter has advanced by 2.

Figure 7 displays two example updates. In Figure 7a, an update to the bucket has occurred in epoch 5. In Figure 7b, record g is added to the bucket in epoch 6. First, we initialize the new record to point to the most recent valid record, f. Then, we change the status indicator to update pointer roles and the epoch number. As we are in epoch 6, the most recent committed record was added in epoch 4 (the previous in-flight pointer). Therefore, pointer 1 is now the committed pointer. The new in-flight pointer is the one pointing to the most recent record added in the previous epoch (pointer 0). The remaining pointer, pointer 2, whose target is older than the new committed pointer, is then assigned the active role and is retargeted to point to the newly prepended record, g.

In Figure 7c, an additional record, h, is added to the bucket after a crash has occurred in epoch 6 (after the update of Figure 7b). Because of the crash, epochs 5 and 6 are on the failure list. Records e, f, and g are thus failed records, because they were added during these epochs and cannot be relied upon to have persisted. The new record, h, refers to the valid head d instead. Then, the status indicator is updated. The snapshot number SS becomes 7. The committed pointer is the one pointing to the most recent persistent record, d. Pointer 1, which points to d, is assigned the committed role. One currently invalid pointer (pointer 2) will point to the newly added record, h. Since the previous epoch is a failed one, there are no in-flight records, so we set the in-flight role as invalid. The net effect is to transform the state of the bucket in such a way that the failed records, e, f, and g, become unreachable.

3.4 Further Details

Global Routines. As noted in Section 3.1, our global fences are executed periodically by a special worker thread (or by a repurposed application thread that has just completed an operation). The worker first increments and persists the global epoch counter under protection of a sequence lock [19]. It then waits for all threads to exit any transaction in the previous epoch, thereby ensuring that every update occurs entirely within a single epoch. (The wait employs a global array, indexed by thread ID, that indicates the epoch of the

thread's current transaction, or 0 if it is not in a transaction.) Finally, the worker initiates the actual whole-cache write-back. In our prototype implementation, this is achieved with a custom system call that executes the Intel WBINVD instruction. This instruction has the side effect of invalidating all cache content. We hypothesize that future machines with persistent memory will provide an alternative instruction that avoids the invalidation.

Following a crash, a *recovery procedure* is invoked. This routine reads the value, F, of the global epoch counter and adds both F and $F-1$ to the failed epoch list (and persists these additions). The crashed epoch, F, is added because the fence that would have forced its writes-back did not start; the previous epoch, $F-1$, is added because the fence that would have forced its writes-back may not have finished. Significantly, the recovery procedure does not delete or modify failed records in the hash chains: as illustrated in Figure 7c, recovery is performed incrementally by application threads as they access data.

Transactions. Transactions are easily added on top of the basic Dalí design. Our prototype employs strong strict two-phase locking (SS2PL): to perform a transaction that includes multiple hash map operations, a thread acquires locks as it progresses, using timeout to detect (conservatively) deadlock with other threads. To preserve the ability to abort (when deadlock is suspected), it buffers its updates in transient state. When it has completed its code, including successful acquisition of all locks, it performs the buffered updates, as described in Section 3.3, and releases all its locks.

In-place Updates. A reader executing in epoch E is interested only in the most recent update of a given key k in E. If there are multiple records for k in E, only the most recent will be used. As a means of reducing memory churn, we modify our update routine to look for a previous entry for k in the current epoch, and to overwrite its associated value, atomically and in place, if it is found.

Multiversioning. Because historical versions are maintained, we can execute read-only operations efficiently, without the need for locking, by pretending that readers execute two epochs in the past, seeing the values that would persist after a crash. This optimization preserves serializability but not strict serializability. It improves throughput by preventing readers from interfering with concurrent update transactions. To ensure consistency, read-only transactions continue to participate in the global array that stalls updates in a new epoch until transactions from the previous epoch have completed.

Garbage Collection. Garbage collection recycles obsolete records that are no longer needed because newer persistent records with the same key exist; it operates at the granularity of a bucket. At the end of an update operation, before releasing the bucket's lock, a thread will occasionally peruse the committed records and identify any for which there exists a more recent committed record with the same key. Removal from the list entails a single atomic pointer update, which is safe as the bucket is locked. Once the removal is persistent (two epochs later), the record can safely be recycled. If memory pressure is detected, we can use incremental persistence to free the record immediately. Otherwise we keep the record on a "retired" list and reclaim it in the thread's first operation two epochs hence.

Because the retired list is transient, we must consider the possibility that records may be lost on a crash, thereby leaking memory. Similar concerns arise when bypassing failed records during an `update` operation, as illustrated in Figure 7b, and when updating the free list of the memory allocator itself. To address these concerns, we can end the recovery

procedure with a sweep of the heap that reclaims any node not found on a bucket list [2]. Since the amount of leakage is likely to be small, this need not occur on every crash.

4 Correctness

We here present an informal proof of Dalí's safety. Specifically, we argue that it satisfies *buffered durable linearizability* [17], an extension of traditional linearizability that accommodates whole-system crashes. For clarity of exposition (and for lack of space), we consider only `read` and `update` operations, omitting garbage collection, in-place updates, multiversioning, and transactions. We begin by arguing that a crash-free parallel history of Dalí is linearizable. We then show that the operations preserved at a crash represent a consistent cut of the history prior to the crash, so that when crashes and lost operations are removed from the history, what remains is still linearizable.

4.1 Linearizability

The code of Figures 4 and 5 defines a notion of `valid_head` for a Dalí bucket. Let us say that a bucket is *well formed* if `valid_head` points to a finite, acyclic list of nodes. We define the *valid content* of a well-formed bucket to comprise the initial occurrences of keys on this list, together with their associated values.

▶ **Theorem 1.** *In the absence of crashes, Dalí is a linearizable implementation of an unordered map.*

Proof. All Dalí operations on the same bucket acquire the bucket's lock; by excluding one another in time they trivially appear to take effect atomically at a point between their invocation and response. While the roles of the various pointers may rotate at epoch boundaries, inspection of the code in Figure 5 confirms that, in the absence of crashes, each newly created node in `update` links to `ptrs[a]` (which is always `valid_head`), and `ptrs[a]` is always updated to point to the new node. A trivial induction (starting with initially empty content) shows that this prepending operation preserves both well formedness and the desired sequential semantics. ◀

4.2 Buffered Durable Linearizability

Buffered durable linearizability [17] extends linearizability to accommodate histories with "full-system" crashes. Such crashes are said to divide a history into *eras*, with no thread executing in more than one era.[3] Information is allowed to be lost in a crash, but only in a consistent way. Specifically, if event e_1 happens before event e_2 ($e_1 <_{hb} e_2$ – e.g., e_1 is a store and e_2 is a load that sees its value), then e_1 cannot be lost unless e_2 is also.

Informally, a history is buffered durably linearizable (BDL) if execution in every era can be explained in terms of information preserved from the consistent cut of the previous era. More precisely, history H is BDL if, for every era ending in a crash, there exists a happens-before consistent cut of the events in that era such that for every prefix P of H, the history P' is linearizable, where P' is obtained from P by removing all crashes and, in all eras other than the last, all events that follow the cut. A concurrent object or system is BDL if all of its realizable histories are.

[3] With apologies to geologists, eras here are generally longer than epochs.

Our BDL proof for Dalí begins with the following lemma:

▶ **Lemma 2.** *An epoch boundary in Dalí represents a consistent cut of the happens-before relation on the hash map.*

Proof. Straightforward: The worker thread that increments the epoch number does so under protection of a sequence lock, and it doesn't release the lock until (a) no thread is still working in the previous epoch and (b) the new epoch number has persisted (so no thread will ever work in the previous epoch again). ◀

Suppose now that we are given a history H comprising read, update, and epoch boundary events, where some of the epoch boundaries are also marked as crashes. The two epochs immediately preceding a crash are said to have *failed*; the rest are *successful*. An update operation is said to be successful if it occurs in a successful epoch and to have failed otherwise. Let us define the "valid content" of bucket B at a point between events in H to mean "a singly linked chain of update records reflecting all and only the successful updates to B prior to this point in H." The following is then our key lemma:

▶ **Lemma 3.** *For any realizable history H of a Dalí bucket B, and any prefix P of H ending with a successful update u, `ptrs[a]` will refer to valid content immediately after u.*

Proof. By induction on successful updates. We can ignore the reads in H as they do not change state. As a base case, we adopt the convention that the initial state of B represents the result of a successful initialization "update." The lemma is trivially true for the history prefix consisting of only this single "update," at the end of which `ptrs[a]` is NULL.

Suppose now that for some constant k and all $0 \leq i < k$, the lemma is true for all prefixes P_i ending with the ith successful update, u_i. We want to prove that the lemma is also true for P_k. First consider the case in which there is no crash between the previous successful update, u_{k-1}, and u_k. By the same reasoning used in the proof of Theorem 1, u_k will prepend a new record onto the chain at `ptrs[a]`, preserving valid content.

If there *is* at least one crash between u_{k-1} and u_k, there must clearly be at least two failed epochs between them. This means that the valid content as of the end of u_{k-1} will have persisted as of the beginning of u_k – its chain will be intact. We wish to show that no changes to the pointers and status indicator that occur between u_{k-1} and u_k – caused by any number of completed or partial failed updates – can prevent u_k from picking up and augmenting u_{k-1}'s valid content. We do so by reasoning on the transitions enumerated in Figure 6.

Let E_{k-1} denote the epoch of u_{k-1} and E_k the epoch of u_k. We note that all failed updates between u_{k-1} and u_k occur in epochs numbered greater than E_{k-1}. Further, let v denote the value of `a` (0, 1, or 2) immediately after u_{k-1}. Any update that sees the state generated by u_{k-1} will use row 2, 3, or 5 of Figure 6, and will choose, as its "new `a`" a value *other* than v. Over the course of subsequent failed updates before u_k, `ptrs[v]`'s role may transition at most twice, from `a` to `f` to `c`. As a consequence, the code of Figure 5 will never change the value of `ptrs[v]` – that pointer will continue to reference u_{k-1}'s valid content until the beginning of u_k.

Reasoning more specifically about the ID roles, a status indicator change persisted by a failed update that happens in epoch $E_{k-1} + 1$ will, by necessity, make `ptrs[v]` the in-flight pointer. A subsequent update that sees this change in epoch $E_{k-1} + 2$ or later will by necessity make `ptrs[v]` the committed pointer. Alternatively, a failed update in epoch $E_{k-1} + 2$ or later, without having seen a previous failed update in epoch $E_{k-1} + 1$, will also make `ptrs[v]` the committed pointer. A subsequent update that sees this change will leave

`ptrs[v]`'s role alone. The net result of all these possibilities is that u_k will chose `ptrs[v]` as the `valid_head` regardless of which failed update's status indicator is read. It will then copy this value to the `next` field of its new node and point `ptrs[a]` at that node, preserving valid content. ◀

▶ **Theorem 4.** *Dalí is a buffered durably linearizable implementation of an unordered map.*

Proof. Straightforward: Given history H, containing crashes, we choose as our cut in each era the end of the last successful epoch. In the era that follows a crash, the visible content of each bucket (the records that will be seen by an initial `read` or `update`) will be precisely the valid content of that bucket. ◀

5 Experiments

We have implemented a prototype version of Dalí in C/C++ with POSIX threads. As described in Section 3.4, we implemented the global fence by exposing the privileged WBINVD instruction to user code using a syscall into a custom kernel module. Since non-volatile memory is not yet widely available, we simulated NVM by memory mapping a `tmpfs` file into Dalí's address space. This interface is consistent with industry projections for NVM [25].

As a representative workload for a hash map, we chose the transactional version of the Yahoo! Cloud Serving Benchmark (YCSB) [9, 11]. Each thread in this benchmark performs transactions repeatedly, for a given period of time. Keys are 8 bytes in length, and are drawn randomly from a uniform distribution of 100 million values. Values are 1000 bytes in length. We initialize the map with all keys in the key range.

The tested version of **Dalí** uses both mentioned optimizations (in-place updates and multiversioning) and our prototype SS2PL transaction processing system. Garbage collection is enabled. Epoch duration is a configurable parameter in Dalí; our experiments use a duration of 100 ms. We compared Dalí with three alternative maps: Silo [26], FOEDUS [18], and an incrementally persistent hash map (IP).

Silo [26] is an open source in-memory database for large multi-core machines.[4] It is a log-based design that maintains both an in-memory and a disk-resident copy. A decentralized log, maintained by designated logging threads, is used to commit transactions. We configured Silo to use NVM for persistent storage – i.e., Silo writes logs to main memory instead of disk.

FOEDUS [18] is an online transaction processing (OLTP) engine, available as open source.[5] The engine is explicitly designed for heterogeneous machines with both DRAM and NVM. Like Silo, FOEDUS is a log-based system with both an transient and persistent copy of the data. Unlike Silo, FOEDUS adopts a dual paging strategy in which a logical page may exist in two physical forms: a mutable volatile page in DRAM and an immutable snapshot page in NVM. FOEDUS commits transactions with the aid of a decentralized logging scheme similar to Silo. FOEDUS offers both key-ordered and unordered storage, based respectively on a B-tree variant and a hash map; our experiments use the latter. Like Dalí, both Silo and FOEDUS may lose recent transactions on a crash (their decentralized logs are reaped into persistence in the background).

We also implemented a data store called **IP**, an incrementally persistent hash map [24], as described in Section 2. As in Dalí, transactions in IP are implemented using SS2PL. To

[4] `https://github.com/stephentu/silo`
[5] `https://github.com/HewlettPackard/foedus`

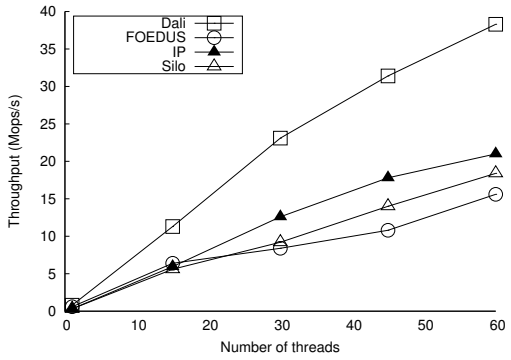


Figure 8 Scalability (75% reads).

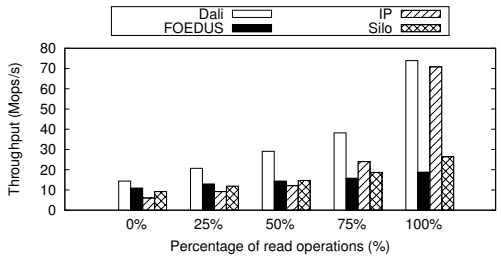


Figure 9 Impact of read:write ratio.

ensure correct recovery, per-thread undo logging is employed. In contrast to Dalí, Silo, and FOEDUS, transactions are immediately committed to persistence.

We benchmarked all four systems on a server-class machine with four Intel Xeon E7-4890 v2 processors, each with 15 cores, running Red Hat Enterprise Linux Server version 7.0. The machine has 3 TB of DRAM main memory. Each processor has a 37.5 MB shared L3 cache, and per-core private L2 and L1 caches of 256 KB and 32 KB, respectively.

Figure 8 shows the transaction throughput of Dalí and the comparison systems while varying the number of worker threads from 1 to 60; transactions here comprise three reads and one write. Dalí achieves a throughput improvement of 2–3× over Silo and FOEDUS across the range of threads. The removal of write-ordering overhead in Dalí reduces the time spent blocking per transaction, thereby improving throughput.

Figure 9 shows experiments that vary the read-to-write ratio at 60 threads across transactions containing four operations. Dalí's performance advantages compared to Silo and FOEDUS are larger for workloads with more reads due to the multiversioning optimization, whereas IP's advantage lies in the reduction in persist instructions at high read percentages.

6 Related Work

Dalí builds upon years of research on in-memory and NVM-centric designs, and upon decades of research on traditional database and multiversioning algorithms. As the promise of NVM is fast and fine-grained durable storage, tailored NVM systems have focused on specific types of applications: namely transactional memory and data storage.

Transactional memory systems are a natural fit for NVM, since a common challenge is to ensure consistent persistent state. The transaction-based NV-Heaps [7] and REWIND [5] and the lock-based Atlas [4] use undo logs to track writes to persistent state as they occur; on system crash, changes are rolled back. In contrast, the redo-logging Mnemosyne [27] redirects writes of persistent state to a thread-private location; on transaction commit, it copies changes to the shared state. All these systems are fine-grained "incrementally persistent" designs. A more novel design is SoftWrAP, which uses aliasing to keep both a transient and a persistent copy of data, thus avoiding inconsistencies caused by leaking cache lines [13].

Other authors have built intricate NVM data structures for data storage and transaction processing. Several projects use custom NVM-adapted trees that support atomic and durable updates [5, 6, 23, 28]. Schwalb et al. present a lock-free NVM hash map [24] similar to the incrementally persistent design of Section 2. These data structures all use incremental persistence, either within individual updates or in transaction logging.

Recent research on in-memory databases has also investigated NVM-based durability. Both DeBrabant et al. [10] and Arulraj et al. [1] explore how traditional database designs can be adapted for architectures with NVM, while Kimura's FOEDUS [18] builds a custom DBMS for NVM from the ground up.

Like Dalí, traditional disk-resident databases maintain a single persistent copy of the data (traditionally on disk, but for Dalí in NVM) and must move data into transient storage (traditionally DRAM, but for Dalí CPU caches) in order to modify it. Viewed in this light, CPU caches in Dalí resemble a database's STEALING, FORCEABLE buffer cache [15]. The updating algorithm of the incrementally persistent hash map is similar to traditional shadow paging [14, 29], but at a finer granularity. To the best of our knowledge, no prior art in this space has allowed writes to be reordered within an update or transaction, as Dalí does.

The prepend-only buckets of Dalí resemble several structures designed for RCU [20]. Dalí also resembles work on *persistent* data structures, where "persistent" here refers to the data structure's ability to preserve its own history [12]. Data structures of this sort are widely used in functional programming languages, where their ability to share space among multiple versions provides an efficient alternative to mutating a single version [22]. In the notation of this field, Dalí resembles a *partially persistent* data structure – one in which earlier versions can be read but only the most recent state can serve as the basis for new versions [12].

7 Conclusion

We have introduced *periodic persistence* as an alternative to the incremental persistence employed by most previous data structures designed for nonvolatile memory. Dalí, our periodically persistent hash map, executes neither explicit writes-back nor persistence fences within updates; instead, it tracks the recent history of the map and relies on a *periodic global fence* to force recent changes into persistence. Experiments with a prototype implementation suggest that Dalí can provide nearly twice the throughput of file-based or incrementally persistent alternatives. We speculate other data structures could be adapted to periodic persistence, and that the paradigm might be adaptable to traditional disk based architectures.

Acknowledgments. The authors sincerely thank Hideaki Kimura and Tianzheng Wang for their helpful suggestions and assistance.

References

1 Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. Let's talk about storage: Recovery methods for non-volatile memory database systems. In *SIGMOD*, Melbourne, Australia, 2015.

2 Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. Makalu: Fast recoverable allocation of non-volatile memory. In *OOPSLA*, Amsterdam, Netherlands, 2016.

3 João Cachopo and António Rito-Silva. Versioned boxes as the basis for memory transactions. *Science of Computer Programming*, 63(2):172–185, December 2006.

4 Dhruva Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Leveraging locks for NVM consistency. In *OOPSLA*, Portland, OR, USA, 2014. doi:10.1145/2660193.2660224.

5 Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Viglas. Recovery write-ahead system for in-memory non-volatile data-structures. *Proc. VLDB Endow.*, 8(5), January 2015. doi:10.14778/2735479.2735483.

6 Shimin Chen and Qin Jin. Persistent B+-trees in non-volatile main memory. *Proc. VLDB Endow.*, 8(7), 2015.

7 Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. Making persistent objects fast and safe with NVM. In *ASPLOS*, Newport Beach, CA, USA, 2011. doi:10.1145/1950365.1950380.

8 Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin C. Lee, Doug Burger, and Derrick Coetzee. Better I/O through byte-addressable, persistent memory. In *SOSP*, Big Sky, MT, USA, 2009.

9 Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *SOCC*, Indianapolis, IN, USA, 2010.

10 Justin DeBrabant, Joy Arulraj, Andrew Pavlo, Michael Stonebraker, Stan Zdonik, and Subramanya R. Dulloor. A prolegomenon on OLTP database systems for non-volatile memory. *Proc. VLDB Endow.*, 7(14), 2014.

11 Anamika Dey, Alan Fekete, Raghunath Nambiar, and Uwe Rohm. YCSB+T: Benchmarking web-scale transactional databases. In *ICDEW*, Chicago, IL, USA, 2014.

12 James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. In *STOC*, Berkeley, CA, USA, 1986.

13 Eric R. Giles, Kshitij Doshi, and Peter Varman. Softwrap: A lightweight framework for transactional support of storage class memory. In *MSST*, Santa Clara, CA, USA, 2015.

14 Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, and Irving Traiger. The recovery manager of the System R database manager. *ACM Computing Survey*, 13(2):223–242, June 1981.

15 Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Survey*, 15(4):287–317, December 1983.

16 Intel Corp. Intel architecture instruction set extensions programming reference. Technical Report 319433-022, Intel Corp., October 2014.

17 Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *DISC*, Paris, France, 2016.

18 Hideaki Kimura. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *SIGMOD*, Melbourne, Australia, 2015.

19 Christoph Lameter. Effective synchronization on Linux/NUMA systems. In *Proc. of the Gelato Federation Meeting*, San Jose, CA, USA, 2005.

20 Paul E. McKenney, Dipankar Sarma, Andrea Arcangeli, Andi Kleen, Orran Krieger, and Rusty Russell. Read copy update. In *Ottawa Linux Symposium*, Ottowa, Canada, 2002.

21 Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. An analysis of persistent memory use with WHISPER. In *ASPLOS*, Xi'an, China, 2017.

22 Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

23 Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *SIGMOD*, San Francisco, CA, USA, 2016.

24 David Schwalb, Markus Dreseler, Matthias Uflacker, and Hasso Plattner. NVC-Hashmap: A persistent and concurrent hashmap for non-volatile memories. In *IMDM*, Kohala Coast, HI, USA, 2015.

25 Storage Networking Industry Association (SNIA) Non-Volatile Memory Programming Model. http://www.snia.org/tech_activities/standards/curr_standards/npm.

26 Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, Farmington, PA, USA, 2013.

27 Haris Volos, Andres Jaan Tack, and Michael M. Swift. Lightweight persistent memory. In *ASPLOS*, Newport Beach, CA, USA, 2011. doi:10.1145/1950365.1950379.

28 Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. NV-Tree: Reducing consistency cost for NVM-based single level systems. In *FAST*, Santa Clara, CA, USA, 2015.

29 Tatu Ylönen. Concurrent shadow paging: A new direction for database research. Technical Report 1992/TKO-B86, Helsinki University of Technology, Helsinki, Finland, 1992.

Revision Notice

This is a revised version of the eponymous paper appeared in the proceedings of DISC 2017 (LIPIcs, volume 91, http://www.dagstuhl.de/dagpub/978-3-95977-053-8, published in October, 2017), in which a missing funding acknowledgment has been included.

Dagstuhl Publishing – July 19, 2018.

Symmetry Breaking in the Congest Model: Time- and Message-Efficient Algorithms for Ruling Sets*

Shreyas Pai[†1], Gopal Pandurangan[‡2], Sriram V. Pemmaraju[§3], Talal Riaz[¶4], and Peter Robinson[5]

1 **Department of Computer Science, The University of Iowa, Iowa City, IA, USA**
shreyas-pai@uiowa.edu
2 **Department of Computer Science, University of Houston, TX 77204, USA**
gopalpandurangan@gmail.com
3 **Department of Computer Science, The University of Iowa, Iowa City, IA, USA**
sriram-pemmaraju@uiowa.edu
4 **Department of Computer Science, The University of Iowa, Iowa City, IA, USA**
talal-riaz@uiowa.edu
5 **Department of Computer Science, Royal Holloway, University of London, UK**
peter.robinson@rhul.ac.uk

Abstract

We study local symmetry breaking problems in the Congest model, focusing on ruling set problems, which generalize the fundamental Maximal Independent Set (MIS) problem. The *time (round) complexity* of MIS (and ruling sets) have attracted much attention in the Local model. Indeed, recent results (Barenboim et al., FOCS 2012, Ghaffari SODA 2016) for the MIS problem have tried to break the long-standing $O(\log n)$-round "barrier" achieved by Luby's algorithm, but these yield $o(\log n)$-round complexity only when the maximum degree Δ is somewhat small relative to n. *More importantly, these results apply only in the* Local *model.* In fact, the best known time bound in the Congest model is still $O(\log n)$ (via Luby's algorithm) even for moderately small Δ (i.e., for $\Delta = \Omega(\log n)$ and $\Delta = o(n)$). Furthermore, *message complexity* has been largely ignored in the context of local symmetry breaking. Luby's algorithm takes $O(m)$ messages on m-edge graphs and this is the best known bound with respect to messages. Our work is motivated by the following central question: can we break the $\Theta(\log n)$ time complexity barrier and the $\Theta(m)$ message complexity barrier in the Congest model for MIS or closely-related symmetry breaking problems?

This paper presents progress towards this question for the distributed ruling set problem in the Congest model. A *β-ruling set* is an independent set such that every node in the graph is at most β hops from a node in the independent set. We present the following results:

- *Time Complexity:* We show that we can break the $O(\log n)$ "barrier" for 2- and 3-ruling sets. We compute 3-ruling sets in $O\left(\frac{\log n}{\log\log n}\right)$ rounds with high probability (whp). More generally we show that 2-ruling sets can be computed in $O\left(\log\Delta \cdot (\log n)^{1/2+\varepsilon} + \frac{\log n}{\log\log n}\right)$ rounds for any $\varepsilon > 0$, which is $o(\log n)$ for a wide range of Δ values (e.g., $\Delta = 2^{(\log n)^{1/2-\varepsilon}}$). These are the first 2- and 3-ruling set algorithms to improve over the $O(\log n)$-round complexity of Luby's algorithm in the Congest model.

* A short version of this paper appeared as a 3-page "Brief Announcement" in PODC 2017. A full version of this paper appears on arxiv [18], https://arxiv.org/abs/1705.07861.

† Supported in part by NSF grant CCF-1318166.
‡ Supported, in part, by NSF grants CCF-1527867, CCF-1540512, IIS-1633720, and CCF-1717075.
§ Supported in part by NSF grant CCF-1318166.
¶ Supported in part by NSF grant CCF-1318166.

© Shreyas Pai, Gopal Pandurangan, Sriram V. Pemmaraju, Talal Riaz, and Peter Robinson;
licensed under Creative Commons License CC-BY

- *Message Complexity:* We show an $\Omega(n^2)$ lower bound on the message complexity of computing an MIS (i.e., 1-ruling set) which holds also for randomized algorithms and present a contrast to this by showing a randomized algorithm for 2-ruling sets that, whp, uses only $O(n \log^2 n)$ messages and runs in $O(\Delta \log n)$ rounds. This is the first message-efficient algorithm known for ruling sets, which has message complexity nearly linear in n (which is optimal up to a polylogarithmic factor).

1998 ACM Subject Classification C.2.4 Distributed Systems, F.1.2 Modes of Computation, F.2.2 Nonnumerical Algorithms and Problems, G.2.2 Graph Theory

Keywords and phrases Congest model, Local model, Maximal independent set, Message complexity, Round complexity, Ruling sets, Symmetry breaking

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.38

1 Introduction

The *maximal independent set (MIS)* problem is one of the fundamental problems in distributed computing because it is an elegant abstraction of "local symmetry breaking," an issue that arises repeatedly in distributed computing. About 30 years ago Alon, Babai, and Itai [1] and Luby [17] presented a randomized algorithm for MIS, running on n-node graphs in $O(\log n)$ rounds with high probability (whp)[1]. Since then the MIS problem has been studied extensively and recently, there has been some exciting progress in designing faster MIS algorithms. For n-node graphs with maximum degree Δ, Ghaffari [10] presented an MIS algorithm running in $O(\log \Delta) + 2^{O(\sqrt{\log \log n})}$ rounds, improving over the algorithm of Barenboim et al. [5] that runs in $O(\log^2 \Delta) + 2^{O(\sqrt{\log \log n})}$ rounds. Ghaffari's MIS algorithm is the first MIS algorithm to improve over the round complexity of Luby's algorithm when $\Delta = 2^{o(\log n)}$ and Δ is bounded below by $\Omega(\log n)$.[2]

While the results of Ghaffari and Barenboim et al. constitute a significant improvement in our understanding of the round complexity of the MIS problem, it should be noted that both of these results are in the Local model. The Local model [20] is a synchronous, message-passing model of distributed computing in which *messages can be arbitrarily large*. Luby's algorithm, on the other hand, is in the Congest model [20] and uses small messages, i.e., messages that are $O(\log n)$ bits or $O(1)$ words in size. In fact, to date, *Luby's algorithm is the fastest known MIS algorithm in the* Congest *model*; this is the case even when Δ is between $\Omega(\log n)$ and $2^{o(\log n)}$. For example, for the class of graphs with $\Delta = 2^{O(\sqrt{\log n})}$, Ghaffari's MIS algorithm runs in $O(\sqrt{\log n})$ rounds whp in the Local model, but we don't know how to compute an MIS for this class of graphs in $o(\log n)$ rounds in the Congest model. It should be further noted that the MIS algorithms of Ghaffari and Barenboim et al. use messages of size $O(\text{poly}(\Delta) \log n)$ (see Theorem 3.5 in [5]), which can be much larger than the $O(\log n)$-sized messages allowed in the Congest model; in fact these algorithms do not work even if messages of size $O(\text{poly}(\log n))$ were allowed. Furthermore, large messages arise in these algorithms from a topology-gathering step in which cluster-leaders gather the

[1] Throughout, we use "with high probability (whp)" to mean with probability at least $1 - 1/n^c$, for some $c \geqslant 1$.

[2] For $\Delta = o(\log n)$, the deterministic MIS algorithm of Barenboim, Elkin, and Kuhn [3] that runs $O(\Delta + \log^* n)$ rounds is faster than Luby's algorithm.

entire topology of their clusters in order to compute an MIS of their cluster – this step seems fundamental to these algorithms and there does not seem to be an efficient way to simulate this step in the CONGEST model.

Ruling sets are a natural generalization of MIS and have also been well-studied in the LOCAL model. An (α, β)-*ruling set* [11] is a node-subset T such that (i) any two distinct nodes in T are at least α hops apart in G and (ii) every node in the graph is at most β hops from some node in T. A $(2, \beta)$-ruling set is an independent set and since such ruling sets are the focus of this paper, we use the shorthand β-*ruling sets* to refer to $(2, \beta)$-ruling sets. (Using this terminology an MIS is just a 1-ruling set.) The above mentioned MIS results [5, 10] have also led to the *sublogarithmic*-round algorithms for β-ruling sets for $\beta \geqslant 2$. The earliest instance of such a result was the algorithm of Kothapalli and Pemmaraju [14] that computed a 2-ruling set in $O(\sqrt{\log \Delta} \cdot (\log n)^{1/4})$ rounds by using an earlier version of the Barenboim et al. [4] MIS algorithm. There have been several further improvements in the running time of ruling set algorithms culminating in the $O(\beta \log^{1/\beta} \Delta) + 2^{O(\sqrt{\log \log n})}$ round β-ruling set algorithm of Ghaffari [10]. This result is based on a recursive sparsification procedure of Bisht et al. [7] that reduces the β-ruling set problem on graphs with maximum degree Δ to an MIS problem on graphs with degree much smaller. Ghaffari's β-ruling set result is also interesting because it identifies a separation between 2-ruling sets and MIS (1-ruling sets). This follows from the lower bound of $\Omega\left(\min\left\{\sqrt{\frac{\log n}{\log \log n}}, \frac{\log \Delta}{\log \log \Delta}\right\}\right)$ for MIS due to Kuhn et al. [15]. Again, we emphasize here that all of these improvements for ruling set algorithms are *only in the* LOCAL *model* because these ruling set algorithms rely on LOCAL-model MIS algorithms to "finish off" the processing of small degree subgraphs. As far as we know, prior to the current work there has been no $o(\log n)$-round, β-ruling set algorithm in the CONGEST model for any $\beta = O(1)$.

The focus of all the above results has been on the time (round) complexity. *Message complexity*, on the other hand, has been largely ignored in the context of local symmetry breaking problems such as MIS and ruling sets. For a graph with m edges, Luby's algorithm uses $O(m)$ messages in the CONGEST model and until now there has been no MIS or ruling set algorithm that uses $o(m)$ messages. We note that the ruling set algorithm of Goldberg et al. [11] which can be implemented in the CONGEST model [12] also takes at least $\Omega(m)$ messages.

The focus of this paper is symmetry breaking problems in the CONGEST model and the specific question that motivates our work is whether we can go beyond Luby's algorithm in the CONGEST model for MIS or any closely-related symmetry breaking problems such as *ruling sets*. In particular, *can we break the* $\Theta(\log n)$ *time complexity barrier and the* $\Theta(m)$ *message complexity barrier, in the* CONGEST *model for MIS and ruling sets?* In many applications, especially in resource-constrained communication networks and in distributed processing of large-scale data [13], it is important to design distributed algorithms that have low time complexity as well as message complexity.

We present two sets of results, one set focusing on time (round) complexity and the other on message complexity.

1. **Time complexity:** (cf. Section 2) We first show that 2-ruling sets can be computed in the CONGEST model in $O\left(\log \Delta \cdot (\log n)^{1/2+\varepsilon} + \frac{\log n}{\log \log n}\right)$ rounds whp for n-node graphs with maximum degree Δ and for any $\varepsilon > 0$. This is the first algorithm to improve over Luby's algorithm, by running in $o(\log n)$ rounds in the CONGEST model, for a wide range of values of Δ. Specifically our algorithm runs in $o(\log n)$ rounds for Δ bounded above by $2^{(\log n)^{1/2-\varepsilon}}$ for any value of $\varepsilon > 0$. In the full version [18], we show how to compute

3-ruling sets (for any graph) in $O\left(\frac{\log n}{\log \log n}\right)$ rounds whp in the Congest model, using our 2-ruling set algorithm as a subroutine. We also present a simple 5-ruling set algorithm based on Ghaffari's MIS algorithm that runs in $O(\sqrt{\log n})$ rounds in the Congest model.

2. **Message complexity:** (cf. Sections 3 and 4) We show that $\Omega(n^2)$ is a fundamental lower bound for computing an MIS (i.e., 1-ruling set) by showing that there exists graphs (with $m = \Theta(n^2)$ edges) where any distributed MIS algorithm needs $\Omega(n^2)$ messages. In contrast, we show that 2-ruling sets can be computed using significantly smaller message complexity. In particular, we present a randomized 2-ruling set algorithm that, whp, uses $O(n \log^2 n)$ messages and runs in $O(\Delta \log n)$ rounds. This is the first $o(m)$-message algorithm known for ruling sets, which takes near-linear (in n) message complexity. This message bound is tight up to a polylogarithmic factor, since we show that any $O(1)$-ruling set (randomized) algorithm that succeeds with probability $1 - o(1)$ requires $\Omega(n)$ messages in the worst case. We also present a simple 2-ruling set algorithm that uses $O(n^{1.5} \log n)$ messages, but runs faster – in $O(\log n)$ rounds.

Our results make progress towards understanding the complexity of symmetry breaking, in particular with respect to ruling sets, in the Congest model. With regards to time complexity, our results, for the first time, show that one can obtain $o(\log n)$ round algorithms for ruling sets in the Congest model. With regards to message complexity, our results are (essentially) tight: while MIS needs quadratic (in n) messages in the worst case, 2-ruling sets can be computed using near-linear (in n) messages. Other related work and omitted proofs can be found in the full version [18].

1.1 Distributed Computing Model

We consider the standard synchronous Congest model [20] described as follows.

We are given a distributed network of n nodes, modeled as an undirected graph G. Each node hosts a processor with limited initial knowledge. We assume that nodes have unique `ID`s (this is not essential, but simplifies presentation), and at the beginning of the computation each node is provided its `ID` as input. Thus, a node has only *local* knowledge[3]. Specifically we assume that each node has ports (each port having a unique port number); each incident edge is connected to one distinct port. This model is referred to as the *clean network model* in [20] and is also sometimes referred to as the KT_0 model, i.e., the initial (K)nowledge of all nodes is restricted (T)ill radius 0 (i.e., just the local knowledge) [2].

Nodes are allowed to communicate through the edges of the graph G and it is assumed that communication is synchronous and occurs in discrete rounds (time steps). In each round, each node can perform some local computation including accessing a private source of randomness, and can exchange (possibly distinct) $O(\log n)$-bit messages with each of its neighboring nodes. This model of distributed computation is called the Congest$(\log n)$ model or simply the Congest model [20].

[3] Our near-linear message-efficient algorithm (Section 3) does not require knowledge of n or Δ, whereas our time-efficient algorithms (Section 2) assume knowledge of n and Δ (otherwise it will work up to a given Δ).

1.2 Technical Overview

1.2.1 Time Bounds

The MIS algorithms of Barenboim et al. [5] and Ghaffari [10] use a 2-phase strategy, attributed to Beck [6], who used it in his algorithmic version of the Lovász Local Lemma. In the first phase, some number of iterations of a Luby-type "base algorithm" are run (in the CONGEST model). During this phase, some nodes join the MIS and these nodes and their neighbors become inactive. The first phase is run until the graph is "shattered", i.e., the nodes that remain active induce a number of "small" connected components. Once the graph is "shattered", the algorithm switches to the second, deterministic phase to "finish off" the problem in the remaining small components. It is this second phase that relies critically on the use of the LOCAL model in order to run fast.

In general, in the CONGEST model it is not clear how to take advantage of low degree or low diameter or small size of a connected component to solve symmetry-breaking problems (MIS or ruling sets) faster than the $O(\log n)$-round bound provided by Luby's algorithm. In both Barenboim et al. [5] and Ghaffari [10], a key ingredient of the second "finish-off" phase is the deterministic network decomposition algorithm of Panconesi and Srinivasan [19] that can be used to compute an MIS in $O(2^{\sqrt{\log s}})$ rounds on a graph with s nodes in the LOCAL model. If one can get connected components of size $O(\text{poly}(\log n))$ then it is possible to finish the rest of the algorithm in $2^{O(\sqrt{\log \log n})}$ rounds and this is indeed the source of the "$2^{O(\sqrt{\log \log n})}$" term in the round complexity of these MIS algorithms. In fact, the Panconesi-Srinivasan network decomposition algorithm itself runs in the CONGEST model, but once the network has been decomposed into small diameter clusters then algorithms simply resort to gathering the entire topology of a cluster at a cluster-leader and this requires large messages. Currently, there seem to be no techniques for symmetry breaking problems in the CONGEST model that are able to take advantage of the diameter of a network being small. As far as we know, there is no $o(\log n)$-round $O(1)$-ruling set algorithm in the CONGEST model *even for constant-diameter graphs*, for any constant larger than 1. To obtain our sublogarithmic β-ruling set algorithms (for $\beta = 2, 3, 5$), we use simple *greedy* MIS and 2-ruling set algorithms to process "small" subgraphs in the final stages of algorithm. These greedy algorithms just exchange $O(\log n)$-bit `ID`s with neighbors and run in the CONGEST model, but they can take $\Theta(s)$ rounds in the worst case, where s is the length of the *longest* path in the subgraph. So our main technical contribution is to show that it is possible to do a randomized shattering of the graph so that none of the fragments have any long paths.

1.2.2 Message Bounds

As mentioned earlier, our message complexity lower bound for MIS and the contrasting upper bound for 2-ruling set show a clear separation between these two problems. At a high-level, our lower bound argument exploits the idea of "bridge crossing" (similar to [16]) whose intuition is as follows. We consider two types of related graphs: (1) a *complete bipartite graph* and (2) a *random bridge graph* which consists of a two (almost-)complete bipartite graphs connected by two "bridge" edges chosen randomly (see Figure 1 and Section 4 for a detailed description of the construction). Note that the MIS in a complete bipartite graph is exactly the set of all nodes belonging to one part of the partition. The crucial observation is that if no messages are sent over bridge edges, then the bipartite graphs on either side of the bridge edges behave identically which can result in choosing adjacent nodes in MIS, a violation. In particular, we show that if an algorithm sends $o(n^2)$ messages, then with

probability at least $1 - o(1)$ that there will be *no* message sent over the bridge edges and by symmetry, with probability at least $1/2$, two nodes that are connected by the bridge edge will be chosen to be in the MIS.

Our 2-ruling set algorithm with low-message-complexity crucially uses the fact that, unlike in an MIS, in a 2-ruling set there are 3 categories of nodes: CATEGORY-1 (nodes that are in the independent set), CATEGORY-2 (nodes that are neighbors of CATEGORY-1) and CATEGORY-3 nodes (nodes that are neighbors of CATEGORY-2, but not neighbors of CATEGORY-1). Our algorithm, inspired by Luby's MIS algorithm, uses three main ideas. First, CATEGORY-2 and CATEGORY-3 nodes don't initiate messages; only undecided nodes (i.e., nodes whose category are not yet decided) initiate messages. Second, an undecided node does "checking sampling" (cf. Algorithm 3) first before it does local broadcast, i.e., it samples a few of its neighbors to see if there are any CATEGORY-2 nodes; if so it becomes a CATEGORY-3 node immediately. Third, an undecided node tries to enter the ruling set with probability that is *always* inversely proportional to its *original* degree, i.e., $\Theta(1/d(v))$, where $d(v)$ is the degree of v. This is unlike in Luby's algorithm, where the marking probability is inversely proportional to its *current degree*. These ideas along with an *amortized* charging argument [8] yield our result: an algorithm using $O(n \log^2 n)$ messages and running in $O(\Delta \log n)$ rounds.

2 Time-Efficient Ruling Set Algorithms in the Congest model

The main result of this section is a 2-ruling set algorithm in the CONGEST model that runs in $O\Big(\log \Delta \cdot (\log n)^{1/2+\varepsilon} + \frac{\log n}{\log \log n}\Big)$ rounds whp, for any constant $\varepsilon > 0$, on n-node graphs with maximum degree Δ. An implication of this result is that for graphs with $\Delta = 2^{O((\log n)^{1/2-\varepsilon})}$ for any $\varepsilon > 0$, we can compute a 2-ruling set in $O\Big(\frac{\log n}{\log \log n}\Big)$ rounds in the CONGEST model. A second implication is that using this 2-ruling set algorithm as a subroutine, we can compute a 3-ruling set for *any* graph in $O\Big(\frac{\log n}{\log \log n}\Big)$ rounds whp in the CONGEST model. These are the first sublogarithmic-round CONGEST model algorithms for 2-ruling sets (for a wide range of Δ) and 3-ruling sets. Combining some of the techniques used in these algorithms with the first phase of Ghaffari's MIS algorithm [10], we show that a 5-ruling set can be computed in $O(\sqrt{\log n})$ rounds whp in the CONGEST model. The 3-ruling set and 5-ruling set results appear in the full paper [18].

2.1 The 2-ruling Set Algorithm

Our 2-ruling set algorithm (described in pseudocode below) takes as input an n-node graph with maximum degree $\Delta \leqslant 2^{\sqrt{\log n}}$, along with a parameter $\varepsilon > 0$. For $\Delta > 2^{\sqrt{\log n}}$, we simply execute Luby's MIS algorithm to solve the problem. The algorithm consists of $\lceil \log \Delta \rceil$ *scales* and in scale t, $1 \leqslant t \leqslant \lceil \log \Delta \rceil$, nodes with degrees at most $\Delta_t := \Delta/2^{t-1}$ are processed. Each scale consists of $\Theta(\log^{1/2+\varepsilon} n)$ *iterations*. In an iteration i, in scale t, each undecided node independently joins a set $M_{i,t}$ with probability $1/(\Delta_t \cdot \log^{\varepsilon} n)$ (Line 5). Neighbors of nodes in $M_{i,t}$, that are themselves not in $M_{i,t}$, are set aside and placed in a set $W_{i,t}$ (Lines 6-8). The nodes in $M_{i,t} \cup W_{i,t}$ have decided their fate and we continue to process the undecided nodes. At the end of all the iterations in a scale t, any undecided node that still has $\Delta_t/2$ or more undecided neighbors is placed in a "bad" set B_t for that scale (Line 11), thus effectively deciding the fate of all nodes with degree at least $\Delta_t/2$. We now process the set of scale-t "bad" nodes, B_t, by simply running a greedy 2-ruling set algorithm on B_t (Line 13). We also need to process the sets $M_{i,t}$ (Line 15) and for that we rely on a greedy 1-ruling

Algorithm 1: 2-RULING SET(Graph $G = (V, E)$, $\varepsilon > 0$):

```
1  I ← ∅; S ← V;
2  for scale t = 1, 2, ..., ⌈log Δ⌉ do
3  |  Let Δ_t = Δ/2^{t-1}; S_t ← S;
4  |  for iteration i = 1, 2, ..., ⌈c · log^{1/2+ε} n⌉ do
5  |  |  Each v ∈ S marks itself and joins M_{i,t} with probability 1/(Δ_t · log^ε n);
6  |  |  if v ∈ S is unmarked and a neighbor in S is marked then
7  |  |  |  v joins W_{i,t};
8  |  |  end
9  |  |  S ← S \ (M_{i,t} ∪ W_{i,t});
10 |  end
11 |  B_t ← {v ∈ S | deg_S(v) ⩾ Δ_t/2};
12 |  S ← S \ B_t;
13 |  I ← I ∪ GreedyRulingSet(G[S_t], B_t, 2);
14 end
15 I ← I ∪ (∪_t ∪_i GreedyRulingSet(G[S_t], M_{i,t}, 1));
16 return I;
```

Algorithm 2: GREEDYRULINGSET(Graph $G = (V, E)$, $R \subseteq V$, integer $\beta > 0$):

```
1  I ← ∅; U ← R; // U is the initial set of undecided nodes
2  while U ≠ ∅ do
3  |  for each node v ∈ U in parallel do
4  |  |  if (v has higher ID than all neighbors in U) then
5  |  |  |  I ← I ∪ {v};
6  |  |  |  v and nodes within distance β in G are removed from U
7  |  |  end
8  |  end
9  end
10 return I
```

set algorithm (i.e., a greedy MIS algorithm). Note that the $M_{i,t}$'s are all disconnected from each other since the $W_{i,t}$'s act as "buffers" around the $M_{i,t}$'s. Thus after all the scales are completed, we can compute an MIS on all of the $M_{i,t}$'s in parallel. Since each node in $W_{i,t}$ has a neighbor in $M_{i,t}$, this will guarantee that every node in $W_{i,t}$ has an independent set node at most 2 hops away. In the following algorithm we use $deg_S(v)$ to denote the degree of a vertex v in the $G[S]$, the graph induced by S.

The overall round complexity of this algorithm critically depends on the greedy 2-ruling set algorithm terminating quickly on each B_t (Line 13) and the greedy 1-ruling set algorithm terminating quickly on each $M_{i,t}$ (Line 15). To be concrete, we present below a specific β-ruling set algorithm that greedily picks nodes by their IDs from a given node subset R.

To show that the calls to this greedy ruling set algorithm terminate quickly, we introduce the notion of *witness paths*. If GREEDYRULINGSET(G, R, β) runs for p iterations (of the **while**-loop), then R must contain a sequence of nodes $(v_1, v_2, \ldots, v_p)$ such that v_i, $1 \leqslant i \leqslant p$, joins the independent set I in iteration i and node v_i, $1 < i \leqslant p$, must contain an undecided node with higher ID in its 1-neighborhood in G, which was removed when v_{i-1} and its β-neighborhood in G were removed in iteration $i - 1$. We call such a sequence a *witness path* for the execution of GREEDYRULINGSET. Three simple properties of witness paths are needed in our analysis:

(i) any two nodes v_i and v_j in the witness path are at least $\beta + 1$ hops away in G,

(ii) any two consecutive nodes v_i and v_{i+1} in the witness path are at most $\beta + 1$ hops away in G, and

(iii) $G[R]$ contains a simple path with $(p - 1)(\beta + 1) + 1$ nodes, starting at node v_1, passing through nodes $v_2, v_3, \ldots, v_{p-1}$ and ending at node v_p.

To show that each $M_{i,t}$ can be processed quickly by the greedy 1-ruling set algorithm we show (in Lemma 1) that whp every witness path for the execution of the greedy 1-ruling set algorithm is short. Similarly, to show that each B_t can be processed quickly by the greedy 2-ruling set algorithm we prove (in Lemma 2) that whp a "bad" set B_t cannot contain a witness path of length $\sqrt{\log n}$ or longer to the execution of the greedy 2-ruling set algorithm. At the start of our analysis we observe that the set S_t, which is the set of undecided nodes at the start of scale t, induces a subgraph with maximum degree $\Delta_t = \Delta/2^{t-1}$.

▶ **Lemma 1.** *For all scales t and iterations i,* GreedyRulingSet$(G[S_t], M_{i,t}, 1)$ *runs in* $O\Big(\frac{\log n}{\varepsilon \log\log n}\Big)$ *rounds, whp.*

Proof. Consider an arbitrary scale t and iteration i. By Property (iii) of witness paths, there is a simple path P with $(2p-1)$ nodes in $G[S_t]$, all of whose nodes have joined $M_{i,t}$. Due to independence of the marking step (Line 5) the probability that all nodes in P join $M_{i,t}$ is at most $(1/\Delta_t \cdot \log^{\varepsilon} n)^{2p-1}$. Since $\Delta(G[S_t]) \leqslant \Delta_t$, the number of simple paths with $2p-1$ nodes in $G[S_t]$ is at most $n \cdot \Delta_t^{2p-1}$. Using a union bound over all candidate simple paths with $2p-1$ nodes in $G[S_t]$, we see that the probability that there exists a simple path in $G[M_{i,t}]$ of length $2p-1$ is at most: $n \cdot \Delta_t^{2p-1} \cdot \Big(\frac{1}{\Delta_t \log^{\varepsilon} n}\Big)^{2p-1} = n \cdot \frac{1}{(\log n)^{\varepsilon(2p-1)}}$. Picking p to be the smallest integer such that $2p-1 \geqslant \frac{4 \log n}{\varepsilon \log\log n}$, we get

$$\Pr(\exists \text{ a simple path with } 2p-1 \text{ nodes that joins } M_{i,t}) \leqslant n \cdot \frac{1}{(2^{\log\log n})^{\varepsilon \frac{4\log n}{\varepsilon \log\log n}}} = n \cdot \frac{1}{n^4} = \frac{1}{n^3}.$$

We have $O(\log \Delta \cdot (\log n)^{1/2+\varepsilon})$ different $M_{i,t}$'s. Using a union bound over these $M_{i,t}$'s, we see that the probability that there exists an $M_{i,t}$ containing a simple path with $2p-1$ nodes is at most n^{-2}. Thus with probability at least $1 - 1/n^2$, all of the calls to GreedyRulingSet$(G[S_t], M_{i,t}, 1))$ (in Line 15) complete in $O\Big(\frac{\log n}{\log\log n}\Big)$ rounds. ◀

▶ **Lemma 2.** *For all scales t, the call to* GreedyRulingSet$(G[S_t], B_t, 2)$ *takes* $O(\sqrt{\log n})$ *rounds whp.*

Proof. Let P be a length-p witness path for the execution of GreedyRulingSet$(G[S_t], B_t, 2)$ (Line 13). By Property (i) of witness paths, all pairs of nodes in P are at distance at least 3 from each other. Fix a scale t. We now calculate the probability that all nodes in P belong to B_t. Consider some node $v \in P$. For v to belong to B_t, it must have not marked itself in all iterations of scale t and moreover at least $\Delta_t/2$ neighbors of v in S_t must not have marked themselves in any iteration of scale t. Since the neighborhoods of any two nodes in P are disjoint, the event that v joins B_t is independent of any other node in P joining B_t. Therefore,

$$\Pr(P \text{ is in } B_t) \leqslant \prod_{v \in P} \Pr(v \text{ and at least } \Delta_t/2 \text{ neighbors do not mark themselves in scale } t).$$

This can be bounded above by $\prod_{v \in P} \Big(1 - \frac{1}{\Delta_t (\log n)^{\varepsilon}}\Big)^{\frac{\Delta_t}{2} \cdot c(\log n)^{1/2+\varepsilon}} \leqslant \exp\big(-\frac{c}{2} \cdot (\log n)^{1/2} \cdot p\big)$. Plugging in $p = \sqrt{\log n}$ we see that this probability is bounded above by $n^{-c/2}$. By Property (ii) of witness paths and the fact that $\Delta(G[S_t]) \leqslant \Delta_t$, we know that there are at most $n \cdot (\Delta_t)^{3p}$ length-p candidate witness paths. Using a union bound over all of these, we get that the probability that there exists a witness path that joins B_t is at most $n\Delta^{3p} \cdot n^{-c/2}$. Plugging in $\Delta \leqslant 2^{\sqrt{\log n}}$ and $p = \sqrt{\log n}$ we get that this probability is at most $n \cdot n^3 \cdot n^{-c/2} = n^{-c/2+4}$.

Picking a large enough constant c guarantees that this probability is at most $1/n^2$ and taking a final union bound over all $\lceil \log \Delta \rceil$ scales gives us the result that all calls to GreedyRulingSet$(G[S_t], B_t, 2)$ take $O(\sqrt{\log n})$ rounds whp. ◀

Lemmas 1 and 2 prove upper bounds on the number of rounds it takes for the calls to GreedyRulingSet (in Lines 13 and 15). Now analyzing Algorithm 2-Ruling Set is straightforward and leads to the following theorem.

▶ **Theorem 3.** *Algorithm* 2-RulingSet *computes a 2-ruling set in the* Congest *model in* $O\left(\log \Delta \cdot (\log n)^{1/2+\varepsilon} + \frac{\log n}{\varepsilon \log \log n}\right)$ *rounds, whp.*

3 A Message-Efficient Algorithm for 2-Ruling Set

In this section, we present a randomized distributed algorithm for computing a 2-ruling set in the Congest model that takes $O(n \log^2 n)$ messages and $O(\Delta \log n)$ rounds whp, where n is the number of nodes and Δ is the maximum node degree. The algorithm does not require any global knowledge, including knowledge of n or Δ. We show in Theorem 9 that the algorithm is essentially message-optimal (up to a polylog(n) factor). This is the first message-efficient algorithm known for 2-ruling set, i.e., it takes $o(m)$ messages, where m is the number of edges in the graph. [4]

3.1 The Algorithm

Algorithm 3 is inspired by Luby's algorithm for MIS [17]; however, there are crucial differences. (Note that Luby's algorithm sends $\Theta(m)$ messages.) Given a ruling set R, we classify nodes in V into three categories:

- category-1: nodes that belong to the ruling set R;
- category-2: nodes that have a neighbor in R; and
- category-3: the rest of the nodes, i.e., nodes that have a neighbor in category-2.

At the beginning of the algorithm, each node is undecided, i.e., its category is not set and upon termination, each node knows its category.

Let us describe one iteration of the algorithm (Steps 3-19) from the perspective of an arbitrary node v. Each undecided node v marks itself with probably $1/2d(v)$. If v is marked it samples a set of $\Theta(\log(d(v))$ random neighbours and checks whether any of them belong to category-2 – we call this the *checking sampling step*. If so, then v becomes a category-3 node and is done (i.e., it will never broadcast again, but will continue to answer checking sampling queries, if any, from its neighbors). Otherwise, v performs the *broadcast step*, i.e., it communicates with all its neighbors and checks if there is a marked neighbor that is of equal or higher degree, and if so, it unmarks itself; else it enters the ruling set and becomes a category-1 node.[5] Then node v informs all its neighbors about its category-1 status causing them to become category-2 nodes (if they are not already) and they are done.

A node that does not hear from any of its neighbors knows that it is not a neighbor of any category-1 node. Note that category-2 and category-3 nodes do not initiate messages, which is important for keeping the message complexity low. Another main idea

[4] In the full paper [18] we present a simpler algorithm for 2-ruling set that, whp, takes $O(n^{1.5} \log n)$ messages and runs in $O(\log n)$ rounds.

[5] Alternately, if v finds any category-2 neighbor (that was missed by checking sampling) during broadcast step it becomes a category-3 node and is done. However, this does not give an asymptotic improvement in the message complexity analysis compared to the stated algorithm.

Algorithm 3: Algorithm `2-rulingset-msg`: code for a node v. $d(v)$ is the degree of v.

```
1  status_v = UNDECIDED;
2  while status_v = UNDECIDED do
3  |  if v receives a message from a CATEGORY-1 node then
4  |  |  Set status_v = CATEGORY-2;
5  |  end
6  |  if v is UNDECIDED then it marks itself with probability 1/(2d(v)) ;
7  |  if v is marked then
8  |  |  ( Checking Sampling Step:) Sample a set A_v of 4 log(d(v)) random neighbors
   |  |    independently and uniformly at random (with replacement) ;
9  |  |  Find the categories of all nodes in A_v by communicating with them;
10 |  |  if any node in A_v is a CATEGORY-2 node then
11 |  |  |  Set status_v = CATEGORY-3;
12 |  |  end
13 |  |  else
14 |  |  |  ( (Local) Broadcast Step:) Send the marked status and d(v) value to all neighbors;
15 |  |  |  If v hears from an equal or higher degree (marked) neighbor then v unmarks itself;
16 |  |  |  If v remains marked, set status_v = CATEGORY-1;
17 |  |  |  Announce status to all neighbors;
18 |  |  end
19 |  end
20 end
```

in reducing messages is the random sampling check of a few neighbors to see whether any of them are CATEGORY-2. Although some nodes might send $O(d(v))$ messages, we show in Section 3.2 that most nodes send (and receive) only $O(\log n)$ messages in an amortized sense. Nodes that remain undecided at the end of one iteration continue to the next iteration. It is easy to implement each iteration in a constant number of rounds.

3.2 Analysis of Algorithm 2-rulingset-msg

One *phase* of the algorithm consists of Steps 3-19, which can be implemented in a constant number of rounds. We say that a node is *decided* if it is in CATEGORY-1, CATEGORY-2, or CATEGORY-3. The first lemma, which is easy to establish, shows that if a node is marked, it has a good chance to get decided.

▶ **Lemma 4.** *A node that marks itself in any phase gets decided with probability at least* $1/2$ *in that phase. Furthermore, the probability that a node remains undecided after* $2\log n$ *marked phases is at most* $1/n^2$.

The next lemma bounds the round complexity of the algorithm and establishes its correctness. The round complexity bound is essentially a consequence of the previous lemma and the correctness of the algorithm is easy to check.

▶ **Lemma 5.** *The algorithm* `2-rulingset-msg` *runs in* $O(\Delta \log n)$ *rounds whp. In particular, with probability at least* $1 - 2/n^2$*, a node* v *becomes decided after* $O(d(v)\log n)$ *rounds. When the algorithm terminates, i.e., when all nodes are decided, the* CATEGORY-1*-nodes form a* 2*-ruling set of the graph. Moreover, each node is correctly classified according to its category.*

We now show a technical lemma that is crucially used in proving the message complexity bounds of the algorithm in Lemma 7. It gives a high probability bound on the total number of messages sent by all nodes during the Broadcast step in any particular phase (i.e., Step 14) of the algorithm in terms of a quantity that depends on the number of undecided nodes

and their neighbors. While bounding the expectation is easy, showing concentration is more involved. (We note that we really use only part (b) of the Lemma for our subsequent analysis, but showing part (a) first, helps understand the proof of part (b)).

▶ **Lemma 6.** *Let $U \subseteq V$ be a (sub-)set of undecided nodes at the beginning of a phase. Let $N(v)$ be the set of neighbors of v. Then the following statements hold:*

(a) *Let $Z(U) = U \cup (\cup_{v \in U} N(v))$. The total number of messages sent by all nodes in U during the Broadcast step in this phase (i.e., Step 14) of the algorithm is $O(|Z(U)| \log n)$ with probability at least $1 - 1/n^3$.*

(b) *Let $N'(v)$ be the set of undecided and category 3 neighbors of v and suppose $|N'(v)| \geqslant d(v)/2$ (where $d(v)$ is the degree of v), for each $v \in U$. Let $Z'(U) = U \cup (\cup_{v \in U} N'(v))$. The total number of messages sent by all nodes in U during the Broadcast step in this phase (i.e., Step 14) of the algorithm is $O(|Z'(U)| \log n)$ with probability at least $1 - 1/n^3$.*

▶ **Lemma 7.** *The algorithm `2-rulingset-msg` uses $O(n \log^2 n)$ messages whp.*

Proof. We will argue separately about two kinds of messages that any node can initiate. Consider any node v.

1. *type 1 messages*: In the checking sampling step in some phase, v samples $4 \log d(v)$ random neighbours which costs $O(\log d(v))$ messages in that phase.
2. *type 2 messages*: In the broadcast step in some phase, v sends to all its neighbors which costs $d(v)$ messages. This happens when all the sampled neighbors in set A_v (found in Step 9) are not CATEGORY-2 nodes.

Note that v initiates any message at all, i.e., both type 1 and 2 messages happen, only when v marks itself, which happens with probability $1/2d(v)$.

We first bound the type 1 messages sent overall by all nodes. By the above statement, a node does checking sampling when it marks itself which happens with probability $1/2d(v)$. By Lemma 4, with probability at least $1 - 1/n^2$, a node is marked (before it gets decided) at most $2 \log n$ times. Hence, with probability at least $1 - 1/n^2$, the number of type 1 messages sent by node v is at most $O(\log d(v) \log n)$; this implies, by union bound, that with probability at least $1 - 1/n$ every node v sends at most $O(\log d(v) \log n)$ type 1 messages. Thus, whp, the total number of type 1 messages sent is $\sum_{v \in V} O(\log d(v) \log n) = O(n \log^2 n)$.

We next bound the type 2 messages, i.e., messages sent during the broadcast step. There are two cases to consider in any phase.

Case 1. In this case we focus (only) on the broadcast messages of the set U of undecided nodes v that (each) have at least $d(v)/2$ neighbors that are in CATEGORY-3 or undecided (in that phase). We show by a charging argument that any node receives amortized $O(\log n)$ messages (whp) in this case. When a node u (in this case) broadcasts, its $d(u)$ messages are charged equally to itself and its CATEGORY-3 and undecided neighbors (which number at least $d(u)/2$).

We first show that any CATEGORY-3 or undecided node v is charged by amortized $O(\log n)$ messages in any phase. Consider the set $U(v)$ which is the set of undecided nodes (each of which satisfy Case 1 property of having at least half of its neighbors that are in CATEGORY-3 or undecided in this phase) in the closed neighborhood of v (i.e., $\{v\} \cup N(v)$). Note that v will be charged only by messages broadcast by nodes in the closed neighborhood of v (this "local charging" is needed for the subsequent argument in the next para). Hence we consider the set $Z'(U(v))$ as in in Lemma 6.(b), i.e., define $Z'(U(v)) = U(v) \cup (\cup_{w \in U(v)} N'(w))$, where $N'(w)$ is the set of all UNDECIDED or CATEGORY-3 neighbors of w. Since, by assumption of Case 1, every undecided node $u \in U(v)$ has at least $d(u)/2$ neighbors that are in CATEGORY-3 or

UNDECIDED in the current phase, applying Lemma 6 (part (b)) to the set $Z'(U(v))$ tells us that, with probability at least $1-1/n^2$, the total number of messages broadcast by undecided nodes in $U(v)$ is $O(|Z'(U(v))|\log n)$. Hence, amortizing over the total number of (undecided and CATEGORY-3) nodes in $Z'(U(v))$, we show that v (which is part of $Z'(U(v))$) is charged (amortized) $O(\log n)$ in a phase. Taking a union bound, gives a high probability result for all nodes v.

To show that the same node v is not charged too many times *across phases*, we use the fact that CATEGORY-2 nodes are never charged (and they do not broadcast). We note that if a node enters the ruling set (i.e., becomes CATEGORY-1) in some phase, then all its neighbors become CATEGORY-2 nodes and will never be charged again (in any subsequent phase). Furthermore, since a marked node enters the ruling set with probability at least $1/2$, a neighbor of v (or v itself) gets charged at most $O(\log n)$ times whp. Hence overall a node is charged at most $O(\log^2 n)$ times whp and by union bound, every node gets charged at most $O(\log^2 n)$ times whp.

Case 2. In this case, we focus on the messages broadcast by those undecided nodes v that have at most $d(v)/2-1$ neighbors that are in CATEGORY-3 or undecided, i.e., at least $d(v)/2+1$ neighbors are in CATEGORY-2. By the description of our algorithm, a node enters the broadcast step, only if checking sampling step (Step 8) fails to find a CATEGORY-2 node. The probability of this "bad" event happening is at most $\frac{1}{d(v)^4}$, which is the probability that a CATEGORY-2 neighbor (of which there are at least $d(v)/2$ many) is not among any of the $4\log(d(v))$ randomly sampled neighbors. We next bound the total number of broadcast messages generated by all undecided nodes in Case 2 during the entire course of the algorithm. By Lemma 4, for any node v, Case 2 can potentially happen only $2\log n$ times with probability at least $1-1/n^2$, since that is the number of times v can get marked. Let r.v. Y_v denote the number of Case 2 broadcast messages sent by v during the course of the algorithm. Conditional on the fact that it gets marked at most $2\log n$ times, we have $E[Y_v] = 2\log n\frac{1}{d(v)^4}d(v) = 2\log n\frac{1}{d(v)^3}$.

Let $Y=\sum_{v\in V}Y_v$. Hence, conditional on the fact that each node gets marked at most $2\log n$ times (which happens with probability $\geqslant 1-1/n$) the total expected number of Case 2 broadcast messages sent by all nodes is $E[Y]=\sum_{v\in V}E[Y_v]=\sum_{v\in V}2\log n\frac{1}{d(v)^3}=O(n\log n)$.

We next show concentration of Y (conditionally as mentioned above). We know that $\mathrm{Var}[Y_v]=4\log^2 n(\frac{1}{d(v)^2}-\frac{1}{d(v)^6})\leqslant 4\log^2 n$. Since the random variables Y_v are independent, we have $\mathrm{Var}[Y]=\sum_{v\in V}Var(Y_v)=4n\log^2 n$. Noting that $Y_v-E[Y_v]\leqslant 2n\log n$, we apply Bernstein's inequality [9] to obtain

$$\Pr(Y\geqslant E[Y]+4n\log^2 n)\leqslant \exp\left(-\frac{16n^2\log^4 n}{8n\log^2 n+(2/3)2n\log n(4n\log^2 n)}\right)\leqslant O(1/n^2).$$

Since the conditioning with respect to the fact that all nodes get marked at most $2\log n$ times happens with probability at least $1-1/n$, unconditionally, $\Pr(Y\geqslant\Theta(n\log^2 n))\leqslant O(1/n^2)+1/n$. Hence, the overall broadcast messages sent by nodes in Case 2 is bounded by $O(n\log^2 n)$ whp.

Combining type 1 and type 2 messages, the overall number of messages is bounded by $O(n\log^2 n)$ whp. ◀

Thus we obtain the following theorem. In the full paper [18], we show that this analysis of the Algorithm `2-rulingset-msg` is tight up to a polylogarithmic factor.

▶ **Theorem 8.** *The algorithm* `2-rulingset-msg` *computes a 2-ruling set using* $O(n \log^2 n)$ *messages and terminates in* $O(\Delta \log n)$ *rounds with high probability.*

4 Message Complexity Lower Bounds

We first point out that the bound of Theorem 8 is tight up to logarithmic factors. The proof is a simple indistinguishability argument and is relegated to the full paper [18].

▶ **Theorem 9.** *Any* $O(1)$*-ruling set algorithm that succeeds with probability* $1 - o(1)$ *sends* $\Omega(n)$ *messages in the worst case. This is true even if nodes have prior knowledge of the network size* n*.*

Next, we show a separation between the message complexity of computing an β-ruling set ($\beta > 1$) and an MIS (i.e., 1-ruling set) by proving an unconditional $\Omega(n^2)$ lower bound for the latter.

▶ **Theorem 10.** *Any maximal independent set algorithm that succeeds with probability* $1 - \epsilon$ *on connected networks, where* $0 \leqslant \epsilon < \frac{1}{2}$ *is a constant, has a message complexity of* $\Omega(n^2)$ *in expectation. This is true even if nodes have prior knowledge of the network size* n*.*

Proof. For the sake of a contradiction, assume that there is an algorithm $\mathcal{A}$ that, with probability $1 - o(1)$, sends at most $\mu = o(n^2)$ messages. Moreover, assume that $\mathcal{A}$ succeeds with probability $\geqslant 1 - \epsilon$, for some $\epsilon < \frac{1}{2}$. In the remainder of the proof, we condition on $\mathcal{A}$ sending at most $o(n^2)$ messages.

Consider two copies G and G' of the complete bipartite graph on $n/2$ nodes.[6] For now, we consider the anonymous case where nodes do not have access to unique IDs; we will later show how to remove this restriction. Recall that in our model (cf. Section 1.1), we assume that nodes do not have any prior knowledge of their neighbors in the graph. Instead, each node u has a list ports $1, \ldots, deg_u$, whose destination are wired in advance by an adversary.

We consider two concrete instances of our lower bound network depending on the wiring of the edges. First, let $D = (G, G')$ be the disconnected graph consisting of G and G' and their induced edge sets. It is easy to see that there are exactly 4 possible choices for an MIS on D, as any valid MIS must contain the entire left (resp. right) half of the nodes in G and G' and no other nodes. We denote the events of obtaining one of the four possible MISs by LL', LR', RL', RR', where, e.g., RL' is the event that the right half of G (i.e. nodes in R) and the left half of G' (i.e. nodes in L') are chosen. Let "on D" be the event that $\mathcal{A}$ is executed on graph D. Of course, we cannot assume that algorithm $\mathcal{A}$ does anything useful on this graph as we require $\mathcal{A}$ only to succeed on connected networks. However, we will make use of the symmetry of the components of D later on in the proof.

▶ **Observation 11.** $Pr[LL' \mid on\ D] = Pr[LR' \mid on\ D] = Pr[RL' \mid on\ D] = Pr[RR' \mid on\ D]$.

We now define the second instance of our lower bound graph. Consider any pair of edges $e = (u, v) \in G = (L, R)$ and $e' = (u', v') \in G' = (L', R')$. We define the *bridge graph* by removing e and e' from G respectively G' and, instead, adding the *bridge edges* $b = (u, u')$ and $b' = (v, v')$ by connecting the same ports that were used for e and e'; see Figure 1. We use B to denote a graph that is chosen uniformly at random from all possible bridge graphs, i.e., the edges replaced by bridge edges are chosen uniformly at random according to the above construction. Let "$G \leftrightarrow G'$" be the event that $\mathcal{A}$ sends at least 1 message over a bridge edge and, similarly, we use "$G \not\leftrightarrow G'$" to denote the event that this does not happen.

[6] To simplify our analysis, we assume that $n/2$ and $n/4$ are integers.

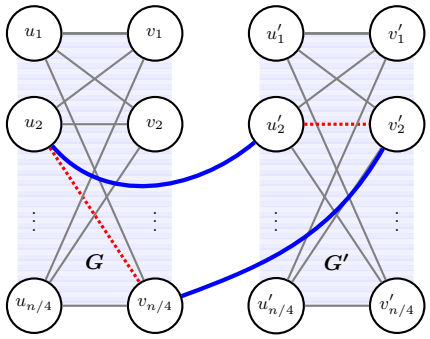


Figure 1 The lower bound graph $B(G, G')$ for Theorem 9 with bridge edges (u_2, u'_2) and $(v_{n/4}, v'_2)$. The disconnected graph D is given by replacing the bridge edges with the dashed edges.

▶ **Lemma 12.** *Consider an execution of algorithm $\mathcal{A}$ on a uniformly at random chosen bridge graph B. The probability that a message is sent across a bridge is $o(1)$, i.e., $Pr[G \not\leftrightarrow G'] = 1 - o(1)$.*

A crucial property of our construction is that, as long as no bridge edge is discovered, the algorithm behaves the same on B as it does on D. The following lemma can be shown by induction over the number of rounds.

▶ **Lemma 13.** *Let Y be any event that is a function of the communication and computation performed by algorithm $\mathcal{A}$. Then, $Pr[Y \mid G \not\leftrightarrow G'] = Pr[Y \mid \text{on } D]$.*

Now consider a run of algorithm $\mathcal{A}$ on a uniformly at random chosen bridge graph B. Let "$\mathcal{A}$ succ." denote the event that $\mathcal{A}$ correctly outputs an MIS. Observe that $\mathcal{A}$ succeeds when executed on B if and only if we arrive at an output configuration corresponding LR' or RL'. It follows that $\Pr[\mathcal{A} \text{ succ.}] = \sum_{W \in \{LR', RL'\}} \Pr[W \mid G \not\leftrightarrow G'] \cdot \Pr[G \not\leftrightarrow G'] + \Pr[\mathcal{A} \text{ succ.} \mid G \leftrightarrow G'] \cdot \Pr[G \leftrightarrow G'] \geqslant 1 - \epsilon$. Lemma 12 tells us that $\Pr[G \leftrightarrow G'] = o(1)$ and, using $\Pr[G \not\leftrightarrow G'] \leqslant 1$, allows us to rewrite the above inequality as $\sum_{W \in \{LR', RL'\}} \Pr[W \mid G \not\leftrightarrow G'] \geqslant 1 - \epsilon - o(1)$. Applying Lemma 13 to the terms in the sum, we get $\sum_{W \in \{LR', RL'\}} \Pr[W \mid \text{on } D] \geqslant 1 - \epsilon - o(1)$. By Observation 11, we know that $\Pr[LR' \mid \text{on } D] + \Pr[RL \mid \text{on } D] \leqslant \frac{1}{2}$, which we can plug into the previously obtained bound on $\sum_{W \in \{LR', RL'\}} \Pr[W \mid \text{on } D]$ to obtain $\epsilon \geqslant \frac{1}{2} - o(1)$, yielding a contradiction to ϵ being a constant less than $\frac{1}{2}$. So far, we have conditioned on $\mathcal{A}$ sending at most $o(n^2)$ messages. Since this event happens with probability $1 - o(1)$, removing the conditioning reduces the above bound on ϵ by at most $o(1)$, which still provides a contradiction.

Finally, we can remove the restriction of not having unique IDs by arguing that the algorithm can generate unique IDs with high probability, since we assume that nodes know n; see the proof of Theorem 9 in the full paper [18] for a similar argument. This completes the proof of Theorem 10. ◀

5 Conclusion

Several key open questions are motivated by the results in this paper. First, can the MIS lower bounds in the LOCAL model shown by Kuhn et al. [15] be extended to 2-ruling sets? In an orthogonal direction, can we derive time lower bounds for MIS in the CONGEST model, that are stronger than their LOCAL-model counterparts? And on the algorithms side, can we

improve ruling-set time bounds in the CONGEST model. e.g., by showing that the 2-ruling set problem can be solved in $O(\log^{\alpha} n)$ rounds, $\alpha < 1$, in CONGEST?

Second, although we have presented near-tight message bounds for 2-ruling sets, we don't have a good understanding of the message-time *tradeoffs*. In particular, a key question is whether we can design a 2-ruling set algorithm that uses $O(n \operatorname{polylog} n)$ messages, while running in $O(\operatorname{polylog} n)$ rounds? More generally, can we obtain a tradeoff that characterizes the dependence of one measure on the other or obtain lower bounds on the complexity of one measure while fixing the other measure.

References

1 Noga Alon, László Babai, and Alon Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *J. Algorithms*, 7(4):567–583, 1986.

2 Baruch Awerbuch, Oded Goldreich, David Peleg, and Ronen Vainish. A trade-off between information and communication in broadcast protocols. *J. ACM*, 37(2):238–256, 1990.

3 Leonid Barenboim, Michael Elkin, and Fabian Kuhn. Distributed $(\delta + 1)$-coloring in linear (in δ) time. *SIAM Journal on Computing*, 43(1):72–95, 2014.

4 Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The locality of distributed symmetry breaking. In *53rd Annual IEEE Symposium on Foundations of Computer Science, FOCS 2012, New Brunswick, NJ, USA, October 20-23, 2012*, pages 321–330, 2012.

5 Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The locality of distributed symmetry breaking. *J. ACM*, 63(3):20:1–20:45, June 2016.

6 József Beck. An algorithmic approach to the Lovász Local Lemma. I. *Random Struct. Algorithms*, 2(4):343–365, December 1991.

7 Tushar Bisht, Kishore Kothapalli, and Sriram Pemmaraju. Brief announcement: Super-fast t-ruling sets. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC'14, pages 379–381, New York, NY, USA, 2014. ACM.

8 T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

9 Devdatt Dubhashi and Alessandro Panconesi. *Concentration of Measure for the Analysis of Randomized Algorithms*. Cambridge University Press, New York, NY, USA, 1st edition, 2009.

10 Mohsen Ghaffari. An improved distributed algorithm for maximal independent set. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 270–277, 2016.

11 A. Goldberg, S. Plotkin, and G. Shannon. Parallel symmetry breaking in sparse graphs. *SIAM J. Discrete Math.*, 1:434–446, 1989.

12 Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. A deterministic almost-tight distributed algorithm for approximating single-source shortest paths. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 489–498, 2016.

13 Hartmut Klauck, Danupon Nanongkai, Gopal Pandurangan, and Peter Robinson. Distributed computation of large-scale graph problems. In *Proceedings of the 26th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 391–410, 2015.

14 Kishore Kothapalli and Sriram V. Pemmaraju. Super-fast 3-ruling sets. In *FSTTCS*, pages 136–147, 2012.

15 Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. Local computation: Lower and upper bounds. *J. ACM*, 63(2):17:1–17:44, March 2016.

16 Shay Kutten, Gopal Pandurangan, David Peleg, Peter Robinson, and Amitabh Trehan. On the complexity of universal leader election. *J. ACM*, 62(1):7:1–7:27, 2015. doi:10.1145/2699440.

17 M. Luby. A simple parallel algorithm for the maximal independent set. *SIAM Journal on Computing*, 15:1036–1053, 1986.

18 Shreyas Pai, Gopal Pandurangan, Sriram V. Pemmaraju, Talal Riaz, and Peter Robinson. Symmetry Breaking in the Congest Model: Time- and Message-Efficient Algorithms for Ruling Sets. *ArXiv e-prints*, May 2017. URL: https://arxiv.org/abs/1705.07861.

19 A. Panconesi and A. Srinivasan. On the complexity of distributed network decomposition. *J. Algorithms*, 20(2):356–374, 1996.

20 David Peleg. *Distributed computing: a locality-sensitive approach*. Society for Industrial and Applied Mathematics, 2000.

Hybrid Consensus: Efficient Consensus in the Permissionless Model*†

Rafael Pass[1] and Elaine Shi[2]

1 **CornellTech, New York, NY, USA**
`rafael@cs.cornell.edu`
2 **Department of Computer Science, Cornell University, Ithaca, NY, USA**
`elaine@cs.cornell.edu`

Abstract

Consensus, or state machine replication is a foundational building block of distributed systems and modern cryptography. Consensus in the classical, "permissioned" setting has been extensively studied in the 30 years of distributed systems literature. Recent developments in Bitcoin and other decentralized cryptocurrencies popularized a new form of consensus in a "permissionless" setting, where anyone can join and leave dynamically, and there is no a-priori knowledge of the number of consensus nodes. So far, however, all known permissionless consensus protocols assume network synchrony, i.e., the protocol must know an upper bound of the network's delay, and transactions confirm slower than this a-priori upper bound.

We initiate the study of the feasibilities and infeasibilities of achieving responsiveness in permissionless consensus. In a responsive protocol, the transaction confirmation time depends only on the *actual* network delay, but not on any a-priori known upper bound such as a synchronous round. Classical protocols in the partial synchronous and asynchronous models naturally achieve responsiveness, since the protocol does not even know any delay upper bound. Unfortunately, we show that in the permissionless setting, consensus is impossible in the asynchronous or partially synchronous models.

On the positive side, we construct a protocol called *Hybrid Consensus* by combining classical-style and blockchain-style consensus. Hybrid Consensus shows that responsiveness is nonetheless possible to achieve in permissionless consensus (assuming proof-of-work) when 1) the protocol knows an upper bound on the network delay; 2) we allow a non-responsive warmup period after which transaction confirmation can become responsive; 3) honesty has some stickiness, i.e., it takes a short while for an adversary to corrupt a node or put it to sleep; and 4) less than 1/3 of the nodes are corrupt. We show that *all these conditions are in fact necessary* – if only one of them is violated, responsiveness would have been impossible. Our work makes a step forward in our understanding of the permissionless model and its differences and relations to classical consensus.

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases Distributed Consensus, Permissionless, Responsiveness

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.39

1 Introduction

State machine replication, also commonly referred to as atomic broadcast, has always been a central distributed systems abstraction. In a state machine replication protocol, a set

* A full version of the paper is available at `https://eprint.iacr.org/2016/917`.
† This research is funded in part by NSF through grant number CNS-1561209.


© Rafael Pass and Elaine Shi;
licensed under Creative Commons License CC-BY
31st International Symposium on Distributed Computing (DISC 2017).
Editor: Andréa W. Richa; Article No. 39; pp. 39:1–39:16

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

of servers seek to agree on an ever-growing, *linearly-ordered log*, such that two important properties are satisfied:

1. *consistency*, i.e., all servers must have the same view of the log; and
2. *liveness*, i.e., whenever a client submits a transaction, the transaction is incorporated quickly into the log.

For more than a decade, companies such as Google and Facebook rely on state machine replication protocols (in the crash fault model) to replicate a significant fraction of their database and computing infrastructure. For simplicity, henceforth we use the term *consensus* to refer to state machine replication protocols throughout this paper.

Traditionally, consensus protocols were studied in a permissioned setting where the set of participating nodes is known a-priori. The enormous success of decentralized cryptocurrencies such as Bitcoin and Ethereum have brought to our attention a new model of distributed consensus, that is, the *permissionless* model. Informally speaking, a permissionless model has the following notable differences from the classical permissioned model:

1. anyone can join the protocol and the network provides no authentication;
2. nodes come and go;
3. there may not be a priori knowledge how many nodes will actually show up; and
4. the number of nodes participating can vary over time.

Although recent (and concurrent) works have begun to explore the theoretical (in)feasibility of reaching consensus in the permissionless model [13, 17, 19, 21], our understanding of the permissionless model is nonetheless relatively little in comparison with our rich body of knowledge for permissioned consensus. In this paper, we make an endeavor at furthering our understanding of the permissionless model.

All known permissionless consensus protocols work in the synchronous model. So far, it is somewhat well-understood that proofs-of-work can be employed to thwart Sybil attacks [5, 13,14,18,19] and thus circumvent earlier known theoretical infeasibilities for the permissionless model [7] – but all known consensus protocols in the permissionless model rely on *network synchrony* [5, 13, 14, 18, 19]. Remarkedly, the famous Nakamoto blockchain [13, 18, 19] (that underlies Bitcoin) also *works only in the synchronous model* where the protocol must know an a-priori upper bound of the network delay (henceforth denoted Δ) [19]. Otherwise, if a delay upper bound Δ is unknown, Pass et al. [19] shows that Nakamoto blockchain's security can be broken. Pass et al. [19] also show that the expected block interval of Nakamoto's blockchain must be set to be, roughly speaking, a constant factor larger than Δ for consistency.

Relying a synchronous model, however, can be undesirable in practice. Since the synchrony parameter Δ must be set conservatively to leave a sufficient safety margin, in practice the actual network's delay is typically much better than this pessimistic upper bound. Unfortunately, any protocol that must wait for at least one synchronous round (or one block-interval) for transaction confirmation cannot benefit from the network's actual performance. We naturally desire a protocol that can confirm transactions as fast as the network makes progress, a notion that we shall henceforth refer to as *responsiveness*. The notion of responsiveness was first defined by Attiya et al. [6]: in a responsive consensus protocol, we require that the transaction confirmation time is a function of the actual network's delay δ, but not of the a priori upper bound Δ that is provided to the protocol as input [6].

We thus ask the following natural question:

Can we achieve responsiveness in permissionless consensus? In other words, *is slow confirmation inherent in permissionless consensus?*

1.1 Our Results and Contributions

Impossibility of permissionless consensus in the partially synchronous model. Traditionally, permissioned state machine replication protocols achieve responsiveness by adopting a partially synchronous or asynchronous model[1], where the protocol does not know any a-priori upper bound of the network delay Δ.

Thus, to answer our earlier question, a first attempt is to design a permissionless consensus protocol for the partially synchronous or asynchronous setting. Unfortunately, this task turns out to be impossible – we show that no consensus protocol unaware of a delay upper bound Δ can simultaneously guarantee consistency and liveness in the permissionless setting – yet another reason why the permissionless setting is fundamentally different from permissioned!

Is responsiveness hopeless for permissionless? At first sight, this pessimistic observation seems to have closed the question: perhaps waiting for the synchronization delay is inevitable in permissionless consensus. Even more discouragingly, it is easy to see that when the number of nodes n is unknown a priori, this lower bound generalizes to one-shot consensus even when the protocol knows Δ but must be responsive – by one-shot consensus, we mean that we need to confirm exactly one transaction proposed to honest nodes upfront, and responsiveness requires that the confirmation time be independent of Δ.

Hybrid consensus. One might now be attempted to think that responsiveness is impossible in a permissionless setting. Perhaps somewhat surprisingly, we construct a protocol called Hybrid Consensus through which we show that responsiveness is actually possible in permissionless state machine replication (assuming proof-of-work), provided that

1. less than $\frac{1}{3} - \epsilon$ fraction of nodes are corrupt where ϵ is an arbitrarily small constant;
2. the protocol is aware of the upper bound Δ, and moreover, we allow a warmup period that depends on Δ – transactions start to confirm as fast as the actual network's performance after this warmup period but not before; and
3. honest nodes stick around and remain honest for a while even when an adversary may try to adaptively corrupt or crash the node – in other words, corruption is not an instant operation but requires some time to take effect. We shall formally define this requirement as the τ-mildly adaptive corruption model where τ is the stickiness parameter.

Lower bounds. *Are the above constraints necessary for achieving responsiveness in the permissionless model?* Earlier we have argued that requirement (2) is necessary since responsiveness is impossible for one-shot consensus even when the protocol knows Δ. Later we will prove several simple but hopefully insightful lower bounds, and show that indeed, all the remaining conditions are necessary as well!

1. We prove that $\frac{1}{3}$ corruption is the optimal resilience parameter for responsiveness – even assuming proof-of-work, static corruption, and even when the number of players is known in advance and all players spawn upfront.
2. If honest nodes do not stick around at least for a while (i.e., if the stickiness parameter $\tau = 0$), responsiveness would also be impossible when the number of players can vary by a factor of 2 (or more) in adjacent rounds.

In summary, our paper gives a somewhat comprehensive answer towards understanding the feasibilities and infeasibilities of achieving *responsiveness* in permissionless consensus.

[1] For the purpose of this paper, the technical difference between partial synchrony and asynchrony is non-essential.

Our results contribute to the further understanding of the permissionless model, how it differs from the classical permissioned model, and how techniques from permissioned consensus can lend to permissionless consensus.

2 Preliminaries

2.1 Permissionless Execution Model

We consider a standard, *round-based* Interactive Turing Machine (ITM) execution model. The execution proceeds in atomic time steps called *rounds* – henceforth in our paper the terms *time* and *round* are used interchangably.

Corruption model. We consider the following corruption model.

- *Spawning.* The adversary is allowed to *spawn* new nodes in any round during the execution. Newly spawned nodes can either be honest or corrupt.
- *Corruption.* To corrupt a node during the protocol, the adversary must first issue a "target corrupt" instruction to an honest node i – let t denote this round. This "target corrupt" instruction will take effect only τ rounds later, i.e., the node i actually becomes corrupt in round $t + \tau$. When a node actually becomes corrupt, its internal states are exposed to the adversary and the adversary fully controls the node's actions. Henceforth, the parameter τ is referred to as the agility parameter. If $\tau = 0$, we model fully adaptive corruptions; if $\tau = \infty$, we model static corruption; anywhere in between, corruption is said to be *mildly adaptive*.
- *Killing.* The adversary can issue a "kill" instruction to kill a corrupt node, and the instruction takes effect immediately. A killed node is no longer considered live. The adversary is not allowed to corrupt honest nodes directly without corrupting them first.

Henceforth, in every round, all nodes that have been spawned but have not been killed are considered live. We say that the adversary is subject to a ρ corruption budget iff in every round, the number of honest nodes that have not received "target corrupt" is more than $1 - \rho$ fraction of the total number of live nodes in that round.

Communication model. The adversary is responsible for delivering all messages sent by nodes (honest or corrupt) to *all* other nodes. The adversary cannot modify the contents of messages broadcast by honest nodes, *but it may arbitrarily delay or reorder the delivery of a message* subject to the following δ-bounded delay constraint: if an honest node sends a message at time t, then in any round $r \geq t + \delta$, any honest node that is live in round r will have received the message, including nodes that may possibly have been sleeping but just woke up in round r, as well as nodes which may have just been spawned at the beginning of round r. We assume that the identity of the sender is not known to the recipient.

Henceforth in this paper, we assume that the protocol is provided with an input Δ which is an a-priori upper bound of the actual network delay δ.

Number of nodes. For simplicity we make a mild assumption about how the number of players may vary. We assume that

1. the protocol is given an estimate n^* of the number of players; and
2. the number of players in every round is between $[\frac{n^*}{2}, n^*]$.

We stress that our partial synchrony impossibility and fully adaptive impossibility results hold as long as the protocol's estimate of the number of players can be off by a factor of 2 (or more)[2].

2.2 State Machine Replication

In this paper, a *permissionless consensus* protocol is a protocol that realizes the "state machine replication" abstraction in a permissionless environment. In a state machine replication protocol, in every round, every honest node receives one or more transactions as input, and outputs a log. Henceforth let LOG_i^r denote node i's output log in round r. A state machine replication protocol is said to satisfy *consistency*, iff except with negligible probability,

1. for any node i honest in round r and any node j honest in round t, either $\mathsf{LOG}_i^r \prec \mathsf{LOG}_j^t$ or $\mathsf{LOG}_j^t \prec \mathsf{LOG}_i^r$, where $\prec$ denotes "is a prefix of" (by convention, we assume that $x \prec x$ for any x); and
2. any honest node's log should not shrink over time.

A state machine replication protocol is said to satisfy $(T_{\text{warm}}, T_{\text{confirm}})$-*liveness*, iff except with negligible probability, if an honest node receives some transaction tx as input or has tx in its output log in round $r \geq T_{\text{warm}}$, then in round $r + T_{\text{confirm}}$ or later, tx appears in all honest nodes' output logs.

Responsiveness. A state machine replication protocol that satisfies $(T_{\text{warm}}, T_{\text{confirm}})$-*liveness* is said to be responsive, iff the function T_{confirm} depends only on the actual maximum network delay δ but not on the a-priori upper bound Δ. Note that we allow the warmup period T_{warmup} to be non-responsive, i.e., dependent on the a-priori upper bound Δ.

3 Limits on Responsiveness in Permissionless Consensus

We present our lower bounds for responsiveness. Due to space constraints, we present proof intuitions in this section and defer full, formal proofs to our online full version [20].

Impossibility of partial synchrony for permissionless. One obvious approach towards achieving responsiveness is to rely on a partially synchronous or asynchronous model – indeed, classical, permissioned consensus protocols in the partially synchronous or asynchronous models naturally provide responsiveness, since the protocol does not even know a network delay upper bound. Unfortunately, as it turns out, the same approach would fail in the permissionless setting.

We prove the following theorem where a partially synchronous protocol is defined to be one whose protocol instructions do not depend on the a-priori delay upper bound Δ.

▶ **Theorem 1** (Impossibility of partial synchrony). *No partially synchronous, permissionless consensus protocol can achieve both consistency and liveness in an execution environment where the protocol is provided with an a-priori estimate of the number of players that can be off by a factor of 2 (or more) – even when no node is corrupt.*

[2] On the other hand, for upper bounds: although the earlier blockchain analysis work by Pass et al. [19] and Fruitchain [21] assume that the number of players in every round stays fixed and is known to the protocol – we observe that it is straightforward to extend these earlier works to the case when the number of players may vary by any known constant factor, and that the protocol is given an estimate that can be off by a known constant factor (see online full version [20] for additional details).

Proof sketch. We explain the proof intuition but defer the formal proof to the online full version [20]. Imagine an execution where honest nodes P_0 and P_1 have a slow network link. P_0 cannot distinguish whether its network link to P_1 is slow, or if P_1 simply did not show up. Had it been the latter case, P_0 must output a decision quickly (and for a sufficiently large choice of Δ, the decision will be output before the end of Δ rounds). Thus P_0 must output a decision quickly in the current execution too (i.e., former case). By symmetry, P_1 must output a decision quickly too. Now, if P_0 and P_1 received different, high-entropy transactions as inputs, they would have resulted in disagreement except with negligible probability. ◀

Resilience lower bound. We next ask the question, suppose that we do allow the protocol to know Δ, and moreover we allow a non-responsive warmup period – in this case, can we achieve responsive permissionless consensus, and if so, what fraction of corruption can we tolerate? We prove the following lower bound.

▶ **Theorem 2** (Resilience lower bound)**.** *No responsive, permissionless consensus protocol can tolerate $\frac{1}{3}$ or more corruptions, even assuming the existence of a proof-of-work oracle, static corruptions, and even when assuming that all nodes are spawned upfront (i.e., no late spawning), and moreover, the protocol is provided with the exact number of players.*

Proof sketch. Our proof is inspired by the well-known $\frac{1}{3}$-corruption lower bound by Dwork et al. [11]. Dwork et al.'s proof considers an execution with three nodes, honest nodes P_0, P_1 and a corrupt node Q. The corrupt node Q simulates two nodes Q_0 and Q_1 in its head where Q_b plays with P_b for $b \in \{0, 1\}$. Messages between P_0 and P_1 are delayed for a sufficiently long time. Dwork et al. argues that in such an execution, P_0's view is the same as an alternate execution where P_1 is the corrupt node, and thus P_0 should output a decision responsively. By symmetry, so will P_1. Now, if P_0 and P_1 received different, high-entropy transactions as inputs, then they would disagree.

The main challenge in our proof is that we must essentially prove the same statement, but assuming a proof-of-work oracle. Since a corrupt node can query the proof-of-work oracle only once in each time step, the adversary cannot simulate two parallel executions without causing any slowdown. Fortunately, there is a way to temporally interleave the two simulated executions, such that the second execution waits for the first one to finish before starting – and the lapse in time till the second execution starts can be charged to the network delay. We defer the full, formal proof to the online full version [20]. ◀

Impossibility of responsiveness with a fully adaptive adversary. We show that responsiveness is impossible if the adversary is fully adaptive, i.e., $\tau = 0$, and moreover, if the number of players in every round can differ by a factor of 2. The intuition is the following: when the agility parameter $\tau = 0$, i.e., if "honesty" has no stickiness, then the adversary can corrupt and kill honest nodes instantly. In other words, the adversary can kill an old batch of honest nodes and spawn a new batch instantly, such that the nodes in two adjacent rounds are completely disjoint – notice that the adversary can do this without even charging to the corruption budget ρ! Thus every round behaves like the start of the execution where the number of online nodes is unpredictable by a factor of $2\times$. This means that even a non-responsive warmup period will not help. We formalize this intuition in the following theorem. It is interesting to note why this lower bound no longer holds when honesty does have some stickiness. In this case, swapping out an old batch of nodes and swapping in a new batch is no longer for free – since when an honest node receives a "target corrupt" instruction, it is charged to the corruption budget ρ.

▶ **Theorem 3** (Impossibility of responsiveness with a fully adaptive adversary)**.** *No protocol (even in the proof-of-work model), given an estimate n^*, can achieve responsive permissionless consensus in a fully adaptive environment (i.e., $\tau = 0$) where the number of nodes in each round is allowed to vary between $[\frac{n^*}{2}, n^*]$ – moreover this holds for any corruption budget ρ.*

We defer the formal proof of the above theorem to our online full version [20].

4 Hybrid Consensus

4.1 Blockchain Preliminaries

An abstract blockchain protocol. One way to realize state machine replication is through a blockchain protocol. In a blockchain protocol, in every round, every honest node receives one or more transactions as input. In every round, every honest node outputs a chain consisting of linearly ordered blocks, where each block is a sequence of logical records (e.g., transactions). As defined by Garay et al. [13] and Pass et al. [19], a blockchain protocol is expected to satisfy three properties except with negligible probability:

1. *consistency*, i.e., all honest nodes' output chains agree with each other except for the trailing λ blocks where λ is a security parameter;
2. *Q-chain quality*, i.e., in any λ consecutive blocks in an honest node's output chain, more than Q fraction must be contributed by honest nodes that have not received "target corrupt"; and
3. *(G, G')-chain growth*, i.e., over any duration of length t where $Gt \geq \lambda$, honest nodes' chains grow by at least Gt and at most $G't$.

It is not hard to see that given a blockchain protocol snailchain, we can construct a non-responsive permissionless consensus protocol, where we simply run the blockchain protocol snailchain, and have each node's output log be its chain minus the trailing $\Theta(\lambda)$ blocks. In particular, the resulting permissionless consensus protocol's liveness can be inferred from snailchain's (positive) chain quality and chain growth lower bound[3].

Nakamoto's blockchain. Earlier, Garay et al. [13] and Pass et al. [19] analyze Nakamoto's blockchain protocol [18] for the case when the number of players n is fixed – we observe that it is straightforward to extend these earlier results to deal with the case when n varies by a known constant factor (see our online full version [20] for additional details). In other words, the following statement is immediately implied by these earlier works [13, 19]: assuming the existence of an "idealized" proof-of-work oracle, there exists a blockchain protocol, as long as

- **(i)** the protocol is provided with an estimate of the number of players that is off by a known constant factor;
- **(ii)** the number of players in every round varies by a known constant; and
- **(iii)** the protocol is aware of a upper bound Δ of the network's delay.

[3] Although a blockchain abstraction may resemble the state machine replication definition in Section 2.2, the blockchain abstraction additionally allows us to express a rough notion of time through chain growth, and express "fairness" through chain quality.

4.2 Our Ideas in a Nutshell

To clarify our contributions, the high-level idea of combining classical- and blockchain-style consensus has been proposed before [9] or in concurrent work [15]. However, these earlier works do not achieve responsiveness or are flawed (see Section 5 for detailed discussions). Our hybrid consensus scheme is different in nature from these other works [9, 15] despite the superficial resemblance in the high-level idea. We now explain intuitively how to derive our hybrid consensus scheme step by step.

At a very high level, the idea is to run an underlying blockchain protocol denoted snailchain – for the time being, imagine that we are running Nakamoto's blockchain as the snailchain. We rely on the blockchain to re-elect committees over time where each committee consists of recently online miners. Each committee will now execute a classical, partially synchronous consensus instance (e.g., PBFT) henceforth referred to as "daily consensus" to confirm transactions. To periodically elect committees, we can do the following: whenever the blockchain advances by λ number of blocks, we re-elect a committee in the following way:

1. first, chop off $\Theta(\lambda)$ number of trailing, unstable blocks; and
2. in the remaining chain, the most recent λ blocks' miners' are elected as the new committee.

Now, we make the following useful observations:

1. Due to the *consistency* property of the blockchain, as long as we removed the trailing $\Theta(\lambda)$ blocks, except with negligible probability, all nodes will agree on the committee – for this reason, removing the trailing $\Theta(\lambda)$ blocks is important, and previous works that neglected this [15] could lead to inconsistency.
2. If the underlying blockchain satisfies $\frac{2}{3}$-*chain quality*, then in every window of λ consecutive blocks in an honest node's chain, at least $\frac{2}{3}$ fraction of blocks that are contributed by honest nodes (that have not received "target corrupt").
3. Finally, due to *chain growth*, it does not take too long to re-elect each new committee.

To make our scheme fully work, however, various non-trivial challenges must be addressed, including

1. how to achieve (near) optimal resilience;
2. how to smoothly switch between multiple daily consensus instances and compose their output logs; and
3. how to deal with a posterior corruption attack.

As we discuss more in the Section 5, some earlier works [9, 15] neglect a subset to all of these issues, making their claims somewhat incomplete or flawed. We now discuss how to address these non-trivial technicalities one by one.

Achieving optimal resilience. For near optimal resilience, our hope is that we can achieve $\frac{2}{3}$-chain quality as long as more than $\frac{2}{3}$ of the nodes are honest (and have not received "target corrupt") – however, perhaps somewhat surprisingly, this is fact *false* for Nakamoto's blockchain! Due to a well-known selfish mining attack [12], Nakamoto's blockchain in fact does not achieve "perfect" chain quality. Specifically, for Nakamoto's blockchain to achieve $\frac{2}{3}$ chain quality, we must in fact assume that more than $\frac{3}{4}$ of the nodes are honest!

To aid understanding, we briefly explain the selfish mining attack. When a corrupt node mines a block, it withholds the block without releasing it to honest nodes. Later, when an honest node mines a equal-length fork, the adversary now immediately releases his private block to race against the honest node's. If the adversary additionally controls network delivery, he can perform a rushing attack such that his private fork is guaranteed to arrive

first at other nodes, and thus everyone will now mine off the adversary's private fork. Such an attack effectively "erases" a portion of the honest node's mining power, a direct effect of which is degraded chain quality as mentioned above.

As argued above, if we use Nakamoto's blockchain as the underlying snailchain, we can tolerate only $\frac{1}{4}$ corruption. However, our goal is to tolerate $\frac{1}{3} - \epsilon$ corruption and thus achieve near optimal resilience. To this end, we rely on the recent Fruitchains [21] as a drop-in replacement for Nakamoto's blockchain, i.e., we will instantiate snailchain with Fruitchains [21]. Interestingly, Fruitchains realizes exactly the same abstraction as Nakamoto's blockchain but achieves near optimal chain quality – more specifically, it achieves this by piggybacking two independent mining processes on top of each other, one for mining fruits, and one for mining blocks. In Fruitchains, blocks contain fruits and fruits in turn contain transactions. One can show that the underlying blockchain's liveness guarantees that honest nodes' work in mining fruits cannot be erased by the adversary. Thus, if we regard the fruits as the new blocks, Fruitchain achieves near optimal chain quality.

Switching between daily consensus instances. Another technicality is how to smoothly transition between multiple daily consensus instances without causing "glitches" in responsive transaction confirmation. To this end, our idea is the following: whenever the blockchain advances by another λ number of blocks, we initiate a stopping procedure for the present daily consensus instance while simultaneously starting the next daily consensus instance. Since the stopping procedure may take some time to complete, during a short window, two or more daily consensus instances may be executing concurrently. Although the new daily consensus instance may start accumulating a log, we defer including this log in the output until the previous daily consensus instance has fully terminated and its log fully output. It is not hard to see that as long as this stopping procedure is responsive, all transactions will confirm responsively without any glitches during the switch. In our online full version [20], we show that subtle composition issues arise when composing multiple daily consensus instances.

On-chain stamping and posterior corruption defense. Although it takes a while for the adversary to adaptively corrupt nodes, it is possible for the adversary to eventually corrupt entire past committees. We would like to retain consistency even when entire past committees can be corrupt – henceforth we refer to this problem as *posterior corruption*. The challenge with posterior corruption is that the corrupt past committee can sign equivocating transactions; and whenever a new node joins the protocol, it cannot distinguish which signature is real and which signature was generated a-posteriori by a corrupt past committee – this can lead to inconsistency. To defend against such a posterior corruption attack, we introduce an on-chain stamping mechanism. When a committee completes its term of appointment (where each term is said to be a day), it will propose hash of the daily log, signed by more than $\frac{1}{3}$ of the committee, to be stamped on the blockchain. Now, although the entirety of a past committee can become corrupt sometime after its term of appointment and can sign arbitrary messages of the past – at this point, these signatures will be too late to deceive anyone, since the hash of the committee's daily log will already be stamped on the blockchain, and thus the corrupt committee of the past can no longer equivocate about their past daily log unless they can find hash collisions.

4.3 Detailed Protocol

We present our protocol below but defer formal definition of building blocks and proofs to the online full version [20]. For modular composition, we make use of a global signing

functionality $\mathcal{G}_{\text{sign}}^{\Sigma}$ (parametrized by a signature scheme Σ) shared by all protocol instance. $\mathcal{G}_{\text{sign}}$ provides the following interfaces:

1. upon receiving `keygen` from a node i: $\mathcal{G}_{\text{sign}}$ calls Σ's key generation to generate a new $(\mathsf{pk}, \mathsf{sk})$ pair and returns pk;
2. upon receiving `mykeys` from a node i, output the set of all public keys that have been generated for i;
3. upon receiving $\texttt{sign}(\mathsf{pk}, \mathsf{msg})$ from a node i, if pk was recorded as a public key generated for i, call $\Sigma.\mathsf{Sign}(\mathsf{sk}, (sid, \mathsf{msg}))$ where sid is the current session identifier, and sk is the secret key corresponding to pk, and return the signature;
4. when i becomes corrupt: return all of node i's secret keys to $\mathcal{A}$.

4.3.1 Offchain BFT

We call each committee's term of service a *day*. For modular protocol composition, we define an intermediate abstraction called a daily offchain consensus protocol, denoted DailyBFT. In DailyBFT, committee members run an offchain BFT instance to decide a daily log, whereas non-members count signatures from committee members. A DailyBFT$[R]$ parametrized by the session identifier R (representing the day) has the following syntax.

Inputs and outputs. In each time step, the environment $\mathcal{Z}$ can provide the following types of inputs multiple times:

1. $\texttt{start}(\mathsf{comm})$ where $\mathsf{comm} = \{\mathsf{pk}_i\}_{i \in [m]}$;
2. TXs; and
3. `stop`.

Honest nodes output the following to $\mathcal{Z}$:

- In each time step t, honest nodes output to the environment $\mathcal{Z}$ $\texttt{notdone}(\mathsf{log}^t)$, until in one final step t^*, it outputs $\texttt{done}(\mathsf{log}^{t^*}, \mathsf{recs})$, where recs is either $\emptyset$ or a set of signed tuples vouching for the hash of the final daily log. After outputting $\texttt{done}(\mathsf{log}^{t^*}, \mathsf{recs})$, honest nodes stop outputting in future time steps.

Construction. In Figure 1, we construct DailyBFT from a "strongly secure BFT" protocol denoted BFT. A strongly secure BFT protocol is a strengthening of the classical, property-based definition of a state machine replication protocol – this strengthening is necessary to defend against a selective opening attack as we discuss in more detail in our formal proofs in the online full version. We give an overview of the protocol below.

- *BFT virtual nodes and selective opening of committee.* When DailyBFT receives a $\texttt{start}(\mathsf{comm})$ command, if comm contains one or more of its own public keys, then the node is elected as a committee member. In this case, the node will fork a BFT virtual node for each public key in comm that belongs to itself. Here the committee is selectively opened by the environment through the $\texttt{start}(\mathsf{comm})$ command, later our proof will need to leverage the strong security of BFT.
- *Member and non-member basic operations.* Committee members populate their daily logs relying on the BFT protocol, whereas committee non-members count signatures from committee members to populate their logs.
- *Termination.* Nodes implement a termination procedure as follows: whenever an honest committee member receives a `stop` instruction, it inputs a special, signed `stop` transaction to each of its BFT virtual node. As soon as the inner BFT instance outputs a log containing `stop` transactions signed by at least $\lceil |\mathsf{comm}|/3 \rceil$ distinct committee public keys, the log is finalized and output. All transactions after the first $\lceil |\mathsf{comm}|/3 \rceil$ `stop` transactions (with distinct committee public keys) are ignored.

- *Signed daily log hashes.* When committee members output `done`, they also output a signed digest of the final daily log – later, our HybridConsensus protocol will stamp this digest onto the snailchain.

4.3.2 Hybrid Consensus

We now describe our hybrid consensus protocol built on top of a blockchain protocol denoted snailchain. The description of our hybrid consensus relies on snailchain being any protocol that realizes an abstract blockchain (see Section 4.1). For conceptual simplicity, we recommend that the reader first think of snailchain as being Nakamoto's original blockchain protocol [13, 18, 19]. However, later we will actually plug in the Fruitchains [21] protocol as a drop-in replacement of Nakamoto to instantiate snailchain. Since Fruitchains [21] has near-optimal chain quality, the resulting hybrid consensus protocol, instantiated with Fruitchains, will have almost-optimal resilience.

Hybrid consensus consumes multiple instances of DailyBFT where rotating committees agree on daily logs. Hybrid consensus primarily does the following:

- It manages the spawning and termination of DailyBFT instances effectively using snailchain as a global clock that offers weak synchronization among honest nodes;
- Recall that each DailyBFT instance does not ensure security for nodes that spawn too late, since committee members can become corrupt far out in the future at which point they can sign arbitrary tuples. Therefore, hybrid consensus introduces an on-chain stamping mechanism to extend security guarantees to even nodes that spawn late.

Figure 2 is an algorithmic description of the HybridConsensus protocol. Each node maintains a history of all past transcripts denoted history – we assume this for simplicity of formalism, and it can be optimized away in practice. Nodes that newly spawn obtain the historical transcripts instantly (in practice this can be instantiated by having honest nodes offer a history retrieval service). When a new node spawns, it populates its LOG as follows:

- *Matching on-chain valid tuples.* A newly spawned node first identifies all on-chain valid tuples of the form (R, h), where R is the day number and h is the hash of the daily log. Then, the node will search history and identify an appropriate daily log log_R that is consistent with h. The node populates LOG with these daily logs. This on-chain matching process effectively provides a safe mechanism for a newly spawned node to catch up and populate old entries of its output LOG.
- *Through daily offchain consensus.* Once this catch-up process is complete, the node will henceforth rely on DailyBFT instances to further populate remaining entries of its output LOG. In each DailyBFT instance, a node can act as a committee member or non-member. To do this, a node monitors its chain from the snailchain instance. As soon as the chain length exceeds $\mathsf{csize} \cdot R + \lambda$, the R-th day starts, at which point the node inputs `stop` to the previous DailyBFT$[R-1]$ instance (if one exists), and inputs `start`(MinersOf(chain[lower(R) : upper(R)])) to the DailyBFT$[R]$ instance. There is typically a period of overlap during which both DailyBFT$[R-1]$ and DailyBFT$[R]$ instances are running and outputting daily logs simultaneously. When nodes assimilate their daily logs into the final output LOG, they make sure that LOG is always contiguous leaving no gaps in between. Due to the timely termination property of DailyBFT, the old DailyBFT$[R-1]$ will terminate fairly soon at which point the new DailyBFT$[R]$ instance fully takes over.

Concrete instantiation. The recent work Fruitchains [21] showed the following informal theorem: under a corruption budget of $\rho < \frac{1}{2}$ (where ρ is a constant), for any $\tau \geq 0$, and any arbitrarily small positive constant η, there is a blockchain protocol (called Fruitchains) in the

Subprotocol DailyBFT$[R]$

On input `init`: $\ell := 0$, $\mathsf{log} := \emptyset$, henceforth let mykeys be an alias for $\mathcal{G}_{\text{sign}}.\texttt{mykeys}$

On input `stop` : for each $\mathsf{BFT}^{\mathsf{pk}}$ virtual node forked: input $\mathsf{TXs} := \{\{\texttt{stop}\}_{\mathsf{pk}^{-1}}\}$ to $\mathsf{BFT}^{\mathsf{pk}}$

On input `start`(comm): if $\mathsf{mykeys} \cap \mathsf{comm} \neq \emptyset$: $\mathsf{isMember} = \mathsf{true}$, else $\mathsf{isMember} = \mathsf{false}$

Case $\mathsf{isMember} = \mathsf{true}$

For each $\mathsf{pk} \in \mathsf{mykeys} \cap \mathsf{comm}$: fork a BFT virtual node, and $\mathsf{BFT}.\texttt{start}(\mathsf{pk}, \mathsf{comm})$.
// henceforth this BFT virtual node is denoted $\mathsf{BFT}^{\mathsf{pk}}$

On input TXs: input TXs to each BFT virtual node forked

Let BFT_0 denote the first such BFT virtual node forked

Let $\mathsf{complete}(\mathsf{log}) = \mathsf{true}$ iff log contains `stop` correctly signed by $\mathsf{th} := \lceil |\mathsf{comm}|/3 \rceil$ distinct pks in comm

Every time step t if `start` has been received and `done` has not been output:
Receive output log^* from BFT_0
If $\mathsf{complete}(\mathsf{log}^*)$ then $\mathsf{log}^* :=$ shortest prefix of log^* such that $\mathsf{complete}(\mathsf{log}^*)$
For each $\mathsf{tx} \in \mathsf{log}^* - \mathsf{log}$ that is not a `stop` transaction:
Let $\ell := \ell + 1$, for each $\mathsf{pk} \in \mathsf{mykeys} \cap \mathsf{comm}$: gossip $\{R, \ell, \mathsf{tx}\}_{\mathsf{pk}^{-1}}$
$\mathsf{log} := \mathsf{log}^*$
If $\mathsf{complete}(\mathsf{log})$: call `Finalize`; else output $\texttt{notdone}(\mathsf{log})$

`Finalize`:
$\mathsf{recs} := \emptyset$, remove all `stop` transactions from log
For each $\mathsf{pk} \in \mathsf{mykeys} \cap \mathsf{comm}$: let $x := \{R, \mathsf{hash}(\mathsf{log})\}_{\mathsf{pk}^{-1}}$, $\mathsf{recs} := \mathsf{recs} \cup \{x\}$, gossip x
Output $\texttt{done}(\mathsf{log}, \mathsf{recs})$, and stop outputting in future time steps.

Case $\mathsf{isMember} = \mathsf{false}$

On receive $\{R, \ell\}_{\mathsf{pk}^{-1}}$ or $\{R, \ell, \mathsf{tx}\}_{\mathsf{pk}^{-1}}$: add message to history and check the following:

On collect (r, ℓ, tx) and signatures s.t. $r = R$ and $\mathsf{th} := \lceil |\mathsf{comm}|/3 \rceil$ distinct pks in comm signed the tuple correctly:
If $\mathsf{log}[\ell]$ has not been set, let $\mathsf{log}[\ell] := \mathsf{tx}$

On collect (r, ℓ) and signatures s.t. $r = R$ and $\mathsf{th} := \lceil |\mathsf{comm}|/3 \rceil$ distinct pks in comm signed the tuple correctly:
Wait till $\mathsf{log}[: \ell]$ all populated
Output $\texttt{done}(\mathsf{log}, \emptyset)$, and stop outputting in future time steps.

Each time step until `done` is output:
let log' := longest contiguous prefix of log, output $\texttt{notdone}(\mathsf{log}')$

Figure 1 Daily offchain consensus protocol. All signing operations are achieved by calling $\mathcal{G}_{\text{sign}}.\texttt{sign}$ which signs the message tagged with the session identifier.

Protocol $\mathsf{HybridConsensus}^{\lambda}$, parametrized by λ

On `init`: let $R := 0$, $\mathsf{LOG}_s := \emptyset$, $\mathsf{LOG} := \emptyset$, $\mathsf{csize} := \lambda$, let the "warmup window" $W_0 := 2\lambda$.

Mempools:

- Fork an instance of mempool denoted snailpool that stores pending records for snailchain.
- Fork another instance of mempool that stores pending transactions, denoted txpool.
- On input TXs: txpool.`propose`(TXs).

Snailchain: Fork an instance of snailchain, in each time step:

- let chain denote the current local chain
- let $\mathsf{pk} := \mathcal{G}_{\text{sign}}$.`keygen`
- let recs := snailpool.`query`(ExtractRecs(chain$[: -\lambda]$)), input (recs, pk) to snailchain

Preprocess:

Wait till $|\mathsf{chain}| \geq W_0$

$L :=$ Find in history maximal, ordered list of (R, log_R) tuples such that R is incrementing with no gaps, and $(R, \mathsf{hash}(\mathsf{log}_R))$ is on-chain valid w.r.t. chain

$\mathsf{LOG} := \mathsf{log}_1 || \mathsf{log}_2 || \ldots || \mathsf{log}_{|L|}$

Daily Offchain Consensus:

Let $R := |L|$

Loop:

Wait till $|\mathsf{chain}| \geq \mathsf{upper}(R+1) + \lambda$, let $R := R + 1$

Let $\mathsf{comm}_R :=$ MinersOf(chain[lower(R) : upper(R)]) where MinersOf parses each block as $(\mathsf{recs}_i, \mathsf{pk}_i)$ and returns a list containing all pk_is

If an instance $\mathsf{DailyBFT}[R-1]$ was started, $\mathsf{DailyBFT}[R-1]$.`stop`

$\mathsf{DailyBFT}[R]$.`start`(comm_R)

Each time step: let TXs := txpool.`query`(LOG), input TXs to $\mathsf{DailyBFT}[R]$

Output: In each time step: let R denote the current day. Let $\mathsf{isdone}(r) = \mathsf{true}$ if $\mathsf{DailyBFT}[r]$ has output `done` in this or earlier time steps.

- If $\mathsf{DailyBFT}[R-1]$ outputs `done`(log_{-1}, recs) in this time step: $\mathsf{LOG}_s := \mathsf{LOG}_s || \mathsf{log}_{-1}$, snailpool.`propose`(recs)
- Let log_{-1} and log be the output logs of $\mathsf{DailyBFT}[R-1]$ and $\mathsf{DailyBFT}[R]$ in this time step respectively (or $\emptyset$ if nothing is output)
- If $\mathsf{isdone}(R-1)$: $\mathsf{LOG} := \mathsf{LOG}_s || \mathsf{log}$; else $\mathsf{LOG} := \mathsf{LOG}_s || \mathsf{log}_{-1}$. Output LOG

On-chain valid: A tuple (R, h) is on-chain valid w.r.t chain iff the following holds

- For at least $\frac{1}{3}$ fraction of distinct $\mathsf{pk} \in$ MinersOf(chain[lower(R), upper(R)]): $\{R, h\}_{\mathsf{pk}^{-1}}$ is the first occurrence in chain$[: -\lambda]$ where pk signed some tuple of the form $(R, \_)$.

Figure 2 Main HybridConsensus protocol. A newly spawned, honest node starts running this protocol. We assume history is the set of all historical transcripts sent and received. We assume that message routing to subprotocol instances is implicit: whenever any subprot[sid] instance is forked, history[subprot[sid]] and protocol messages pertaining to subprot[sid] are automatically routed to the subprot[sid] instance.

proof-of-work model that satisfies consistency, $(1-\eta)(1-\rho)$-chain quality, and $(\frac{1}{c_0\Delta}, \frac{1}{c_1\Delta})$ for appropriate constants c_0 and c_1. As mentioned, although the original Fruitchains work assumes a pre-determined and fixed n, it is straightforward to extend their result to the case when n can vary by a known constant factor, and moreover the protocol knows only an estimate of n that can be off by a known constant factor (see our full version for details [20]).

We will instantiate hybrid consensus with Fruitchains as the underlying snailchain, since Fruitchains has almost ideal chain quality. If we do so, and assuming that the agility parameter τ is sufficiently large, we have the following theorem whose proof is deferred to the online full version [20].

▶ **Theorem 4** (Hybrid consensus over Fruitchains)**.** *Assume the existence of a proof-of-work oracle and one-way functions. Further, assume that $\tau > C\lambda\Delta$ for some appropriate constant C, and assume that the adversary is subject to $\rho < \frac{1}{3}$ corruption budget where ρ is a constant. Then, there exists a permissionless consensus protocol that achieves consistency and $(T_{\text{warm}}, T_{\text{confirm}})$-liveness for $T_{\text{warm}} = O(\lambda\Delta)$ and $T_{\text{confirm}} = O(\lambda\delta)$.*

5 Related Work

Although the idea of combining permissionless consensus and permissioned consensus has been discussed in the community (e.g., the recent work by Decker et al. [9] and the concurrent and independent work ByzCoin [15]), these other works are of a different nature. The concurrent work ByzCoin [15] (Usenix Security'16) does not remove trailing unstabilized blocks from the blockchain for committee election; consequently the nodes may not agree on the committee (and thus consistency can be broken). Although their paper claims to tolerate $\frac{1}{3}$ corruption, the claim is incorrect – due to a well-known selfish mining attack, the adversary can control up to a half of the blocks under $\frac{1}{3}$ corruption, and thus the adversary will control a half of the committee in Byzcoin. Indeed, the authors of Byzcoin themselves acknowledged flaws of their protocol in subsequent blog posts [1, 2], and acknowledged that they need to rely on some of our ideas to fix their incorrect claim. Even with the fixes in their blog posts, it remains unclear what properties their protocol guarantees since they do not offer formal proofs of security – for example, their protocol overlooks the issue of posterior corruption which, as we show, requires non-trivial techniques to handle. The prior work by Decker et al. [9] does not aim to achieve responsiveness which is our primary goal. Specifically, Decker et al. [9] relies on classical consensus such as PBFT to vote on each block as it is mined – thus they have to wait for at least one "block interval". Moreover, Decker et al.'s blockchain variant [9] suffers from the same type of selfish mining attack [12] that Nakamoto's blockchain is prone to. Thus, they cannot tolerate $\frac{1}{3}$ corruption due to degraded chain quality.

Earlier works on permissioned consensus have also considered group reconfiguration. For example, Vertical Paxos [16] and BFT-SMART [8] allow nodes to be reconfigured in a dynamic fashion. These works consider group reconfiguration for a related but somewhat different purpose. A line of research starting from Dolev et al. [10] investigated Byzantine agreement protocols capable of early-stopping when conditions are more benign than the worst-case faulty pattern: e.g., the actual number of faulty nodes turns out to be smaller than the worst-case resilience bound. However, these works are of a different nature: First, these earlier works focus on stopping in a fewer number of synchronous rounds, and it is not part of their goal to achieve *responsiveness*. Second, although some known lower bounds [10] show that the number of actual rounds must be proportional to the actual number of faulty processors – these lower bounds work only for deterministic protocols, and thus they are not applicable in our setting.

The concurrent work SPETRE [22] also aims to achieve responsiveness in permissionless consensus by relaxing the consistency requirement. The subsequent work Solidus [4] aims to achieve responsive permissionless consensus and additionally obtain incentive compatibility guarantees – their paper does not precisely articulate under what model their protocol retains security, but all the lower bounds we prove should apply to their setting, so even if their protocol could be proven secure, it would require the same type of restrictions on the model that we impose. Their incentive compatibility guarantees are heuristic and without a formal proof [3] – we also point out for regarding incentive compatibility, by combining the reward distribution mechanism proposed in Fruitchains [21] with Hybrid Consensus, it is straightforward how to distribute rewards in a manner that *provably* achieves ϵ-Nash equilibrium against any $\frac{1}{3} - \epsilon'$ coalition. This means that no attacker wielding less than $\frac{1}{3}$ hashpower can gain more than ϵ fraction more rewards than its fair share even if he is allowed to arbitrarily deviate from the honest protocol. We leave it as an open question how to *provably* achieve *strict* Nash equilibrium.

Acknowledgements. We would like to thank the reviewers for their insightful comments, especially Christian Cachin whose feedback was critical in helping us improve the paper.

References

1 Byzcoin: Securely scaling blockchains. `http://hackingdistributed.com/2016/08/04/byzcoin/`.

2 Untangling mining incentives in bitcoin and byzcoin. `http://bford.github.io/2016/10/25/mining/`.

3 Personal communication with Kartik Nayak and Ling Ren.

4 Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Alexander Spiegelman. Solidus: An incentive-compatible cryptocurrency based on permissionless byzantine consensus. *CoRR*, abs/1612.02916, 2016.

5 Marcin Andrychowicz and Stefan Dziembowski. Pow-based distributed cryptography with no trusted setup. In *CRYPTO*, pages 379–399, 2015.

6 Hagit Attiya, Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Bounds on the time to reach agreement in the presence of timing uncertainty. *J. ACM*, 41(1):122–152, 1994.

7 Boaz Barak, Ran Canetti, Yehuda Lindell, Rafael Pass, and Tal Rabin. Secure computation without authentication. In *CRYPTO*, pages 361–377, 2005.

8 Alysson Neves Bessani, João Sousa, and Eduardo Adílio Pelinson Alchieri. State machine replication for the masses with BFT-SMART. In *DSN*, pages 355–362, 2014.

9 Christian Decker, Jochen Seidel, and Roger Wattenhofer. Bitcoin meets strong consistency. In *ICDCN*, 2016.

10 Danny Dolev, Ruediger Reischuk, and H. Raymond Strong. Early stopping in byzantine agreement. *J. ACM*, 37(4):720–741, October 1990.

11 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 1988.

12 Ittay Eyal and Emin Gun Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *FC*, 2014.

13 Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Eurocrypt*, 2015.

14 Jonathan Katz, Andrew Miller, and Elaine Shi. Pseudonymous secure computation from time-lock puzzles. *IACR Cryptology ePrint Archive*, 2014:857, 2014.

15 Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *Usenix Security*, 2016.

16 Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In *PODC*, pages 312–313, 2009.

17 Silvio Micali. Algorand: The efficient and democratic ledger. https://arxiv.org/abs/1607.01341, 2016.

18 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

19 Rafael Pass, Lior Seeman, and Abhi Shelat. Analysis of the blockchain protocol in asynchronous networks. In *Eurocrypt*, 2017.

20 Rafael Pass and Elaine Shi. Hybrid consensus: Efficient consensus in the permissionless model. Technical report version, `https://eprint.iacr.org/2016/917`, 2016.

21 Rafael Pass and Elaine Shi. Fruitchains: A fair blockchain. In *PODC*, 2017.

22 Yonatan Sompolinsky, Yoad Lewenberg, and Aviv Zohar. SPECTRE: A fast and scalable cryptocurrency protocol. *IACR Cryptology ePrint Archive*, 2016:1159, 2016.

Dynamic Reconfiguration: Abstraction and Optimal Asynchronous Solution*

Alexander Spiegelman[†1], Idit Keidar[2] and Dahlia Malkhi[3]

1 **Viterbi Dept. of Electrical Engineering, Technion, Haifa, Israel**
2 **Viterbi Dept. of Electrical Engineering, Technion, Haifa, Israel**
3 **VMware Research, Palo Alto, USA**

Abstract

Providing clean and efficient foundations and tools for reconfiguration is a crucial enabler for distributed system management today. This work takes a step towards developing such foundations. It considers classic fault-tolerant atomic objects emulated on top of a static set of fault-prone servers, and turns them into dynamic ones. The specification of a dynamic object extends the corresponding static (non-dynamic) one with an API for changing the underlying set of fault-prone servers. Thus, in a dynamic model, an object can start in some configuration and continue in a different one. Its liveness is preserved through the reconfigurations it undergoes, tolerating a versatile set of faults as it shifts from one configuration to another.

In this paper we present a general abstraction for asynchronous reconfiguration, and exemplify its usefulness for building two dynamic objects: a read/write register and a max-register. We first define a dynamic model with a clean failure condition that allows an administrator to reconfigure the system and switch off a server once the reconfiguration operation removing it completes. We then define the Reconfiguration abstraction and show how it can be used to build dynamic registers and max-registers. Finally, we give an optimal asynchronous algorithm implementing the Reconfiguration abstraction, which in turn leads to the first asynchronous (consensus-free) dynamic register emulation with optimal complexity. More concretely, faced with n requests for configuration changes, the number of configurations that the dynamic register is implemented over is n; and the complexity of each client operation is $O(n)$.

1998 ACM Subject Classification F.1.2 Modes of Computation

Keywords and phrases Reconfiguration, Dynamic Objects, Optimal Algorithm

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.40

1 Introduction

The goal of this paper is to take a static fault-tolerant object like an atomic read/write register and turn it into a dynamic fault-tolerant one. A static object exposes an API (e.g., read/write) to its clients, and is emulated on top of a set of fault-prone servers (sometimes called base objects) via protocols like ABD [5]. We refer to the underlying set of fault-prone servers as a *configuration*. To convert a static object into a dynamic one, we first extend the object's API to support *reconfiguration*. Such an API is essential for administrators, who should be able to remove old or faulty servers and add new ones without shutting down the service. One of the challenges in formalizing dynamic models is to define a precise fault

* A full version of the paper is available at `https://alexanderspiegelman.github.io/alexanderspiegelman.github.io/DynamicTasks.pdf`.

† Alexander Spiegelman is grateful to the Azrieli Foundation for the award of an Azrieli Fellowship.

© Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi;
licensed under Creative Commons License CC-BY

condition, so that an administrator who requests to remove a server s via a reconfiguration operation will know when she can switch s off without risking losing the object's state (e.g., the last written value to a read/write register).

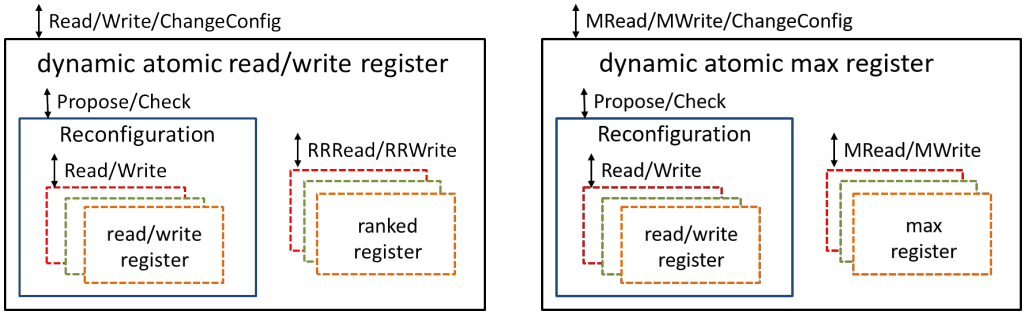
To this end , we first define a clean dynamic failure model, in which an administrator can immediately switch a server s off once a reconfiguration operation that removes s completes. Then, we provide an abstraction for consensus-less reconfiguration in this model. To demonstrate the power of our *Reconfiguration* abstraction we use it to implement two dynamic atomic objects. First, we focus on the basic building block of a read/write register; thus, other (static) objects that can be emulated from read/write registers (e.g., atomic snapshots) can be made dynamic by replacing the underlying registers with dynamic ones. Second, we emulate a max-register [4], which on the one hand can be implemented asynchronously [5, 12] (on top of fault-prone servers), and on the other hand cannot be emulated (for an unbounded number of clients) on top of a bounded set of read/write registers[1] [12, 4]. Thus, a standalone implementation of dynamic max-registers is required.

Complexity. We present an optimal-complexity implementation of our Reconfiguration abstraction in asynchronous environments, which in turn leads to the first optimal implementation of a dynamic read/write register in this model. More concretely, faced with n administrator reconfiguration requests, the number of configurations that the dynamic object is implemented over is n; and the number of rounds (when the algorithm accesses underlying servers) per client operation is $O(n)$. A comparison with previous solutions appears in Section 2.

Dynamic fault model. In Section 3 we provide a succinct failure condition capturing a versatile set of faults under which the dynamic object's liveness is guaranteed. We define the dynamic fault model as an interplay between the object's implementation and its environment: New configurations are *introduced* by clients, (which are part of the object's environment). The object implementation then *activates* the requested configuration, at which point old configurations are *expired*. Between the time when a configuration is introduced and until it is expired, the environment can crash at most a minority of its servers. For example, when reconfiguring a register from configuration $\{A, B, C\}$ into $\{D, E, F\}$, initially a majority of $\{A, B, C\}$ must be available to allow read/write operations to complete. Then, when reconfiguration is triggered, $\{D, E, F\}$ is introduced, and subsequently, majorities of both configurations must be available, to allow state-transfer to occur. Finally, when the reconfiguration operation completes, leading to $\{D, E, F\}$'s activation, $\{A, B, C\}$ is expired, and every server in it may be immediately shutdown.

Reconfiguration abstraction. Since a configuration is a finite set of servers, we can use ABD [5] to emulate in each configuration a set of (static) atomic read/write registers (as well as max-registers), which are available as long as the configuration is not expired. The Reconfiguration abstraction, in contrast, is not tied to a specific configuration, but rather abstracts away the coordination among clients that wish to change the underlying set of servers (configuration) emulating the dynamic object. Its specification, which is formally defined in Section 4, exposes two API methods, *Propose* and *Check*. Clients use Propose to request changes to the configuration, and Check to learn of changes proposed by other

[1] A max-register for k clients requires at least k read/write registers [12].



(a) Dynamic atomic read/write register on top of the Reconfiguration abstraction.

(b) Dynamic atomic max-register on top of the Reconfiguration abstraction.

Figure 1 The Reconfiguration abstraction usage. Solid (dashed) blocks depict dynamic (resp. static) objects.

clients. Both return a configuration and a set of *speculations*. The returned configuration reflects all previous proposals and possibly some ongoing ones. The less obvious return value of Reconfiguration is the speculation set. This set is required since there is no guarantee that all clients see the same sequence of configurations (indeed, Reconfiguration is weaker than consensus). Therefore, a dynamic object implementation that uses Reconfiguration needs to read from every configuration that Check returned to any *other* client, and transfer the most up-to-date value read in any of these to the new configuration returned from Check. To this end, Reconfiguration returns a speculation set that includes all configurations previously returned to all clients (and possibly additional proposed ones).

In Section 5, we implement (1) a dynamic atomic read/write register on top of the Reconfiguration abstraction and static atomic ranked registers [11] (one in every configuration), and (2) a dynamic atomic max-register on top of Reconfiguration and static atomic max-registers. See Figure 1 for illustrations. In Section 6 we give an optimal consensus-less algorithm for Reconfiguration, which together with the read/write register emulation of Section 5 yields an optimal dynamic read/write register algorithm.

In summary, this paper makes three contributions: it defines a failure condition that allows an administrator to shutdown removed servers; it introduces the Reconfiguration abstraction, which captures the essence of reconfiguration; and it presents an asynchronous optimal-complexity solution for dynamic atomic registers. Section 7 concludes the paper, and formal correctness proofs of all algorithms are given in the full paper [25].

2 Related Work

Model. The problem of object reconfiguration has gained growing attention in recent years [15, 20, 3, 21, 18, 14, 24, 13, 23, 17, 22, 6, 7]. However, dynamic failure models do not always make it clear when exactly an administrator can shutdown a removed server. Early works supporting dynamic objects [20, 15, 10] simply assume that a configuration is available as long as some client may try to access it. SmartMerge [18] uses a shared non-reconfigurable auxiliary object (lattice agreement) that is forever available to all clients, meaning that a majority of the servers emulating this auxiliary object can never be switched off. DynaStore [3] was the only previous work to define dynamic failure conditions based on a reconfiguration API, but these conditions are complicated, and restrict reconfiguration attempts as well as failures. Moreover, DynaStore does not separate clients from servers as

we do here. Following [13, 18], we formulate the problem in shared memory, which makes it easier to reason about and clearer.

Other works [6, 7] assume a broadcast mechanism for announcing joins instead of an API for adding and removing processes, and bound the rate of changes of the underlying set of servers; the latter is necessary if one wants to ensure liveness for all operations (as [6] does) – no asynchronous reconfigurable service can ensure liveness unless the reconfiguration rate is limited in some way [23]. Like many earlier works [3, 13, 18], we do not explicitly bound the reconfiguration rate, and hence ensure liveness only if the number of reconfigurations is finite.

Abstractions. All previous works have considered reconfiguration in some specific context – state machine replication [19, 8, 9], read/write register emulation [3, 18, 13, 15], or atomic snapshot [22]. To the best of our knowledge, this work is the first to specify general dynamic objects as extensions of their static counterparts and to provide a general abstraction for dynamic reconfiguration. We note that although [13] define a reconfigure primitive intended to capture the core reconfiguration problem, that primitive is not sufficiently strong for implementing an atomic register, (in particular, since it does not require real-time order), and indeed, they do not implement their atomic register on top of it.

Dynamic register complexity. In a recent non-refereed tutorial [24], we give a generic formulation that allows us to compare the complexity of different algorithms [15, 20, 18, 13, 3], as follows: Given that n is the number of proposed configuration changes and m is the total number of operations (read/write/reconfig) invoked on the atomic register, DynaStore [3] goes through $\mathrm{O}(min(mn, 2^n))$ configurations, and requires a constant number of operations in every configuration, so $\mathrm{O}(min(mn, 2^n))$ is also DynaStore's operation complexity. Parsimonious SpSn [13] reduces the number of traversed configurations to $O(n)$, but since they invoke a linear number of operations in every configuration, their total operation complexity is $O(n^2)$.

Now notice that it is always possible to stagger reconfiguration proposals in a way that forces the system to go through $\Omega(n)$ configurations. The asymptotically optimal $O(n)$ operation complexity is straightforward to achieve in consensus-based solutions [15, 20, 10]. This complexity was also achieved by SmartMerge [18], but this was done using an auxiliary object that was assumed to be live indefinitely, i.e., was not reconfigurable in itself. Our algorithm is the first consensus-free and fully reconfigurable dynamic register algorithm with optimal complexity.

3 Dynamic Model

We consider a fault-prone shared memory model [16]: The system consists of an infinite set Π of *clients* (sometimes called processes), any number of which may fail by crashing, and an infinite set Φ of *servers* (sometimes called base objects) supporting arbitrary atomic low-level objects. Clients access servers via-low level operations (e.g., read/write), which may take arbitrarily long to arrive and complete, hence the system is asynchronous.

We address in the paper two atomic objects: a classical fault tolerant read/write register and a max-register [4]. Both registers provide clients with two API methods: Read and Write in case of read/write register, and MRead and MWrite in case of max-register. In a well-formed execution, a client invokes API methods one at a time, though calls by different clients may be interleaved in real time. For a well-formed execution, there exists a serialization of all client operations that preserves the operations' real time order, such that (1) in case of

read/write register a Read returns the value written in the latest Write preceding it, or $\perp$ if there is no preceding Write; and (2) in case of max-register an MRead returns the highest value written by an MWrite that precedes it, or $\perp$ if there is no preceding MWrite. (In case of max-registers, the values domain is ordered.)

Configurations. The universe of servers is infinite, but at any moment in time, a client chooses to interact with a subset of it. In our model, a *configuration* is a set of included and excluded servers, where configuration *membership* is the set of included and not excluded servers in the configuration. Formally:

$Changes$	$\triangleq$	$\{+s \mid s \in \Phi\} \cup \{-s \mid s \in \Phi\}$
$Configuration$	$\triangleq$	subset of $Changes$
$C.membership$	$\triangleq$	$\{s \mid +s \in C \wedge -s \notin C\}$

For example $C = \{+s_1, +s_2, -s_2, +s_3\}$ is a configuration representing the inclusion of servers s_1, s_2, and s_3, and the exclusion of s_2, and $C.membership$ is $\{s_1, s_3\}$. Tracking excluded servers in addition to the configuration's membership is important in order to reconcile configurations suggested by different clients. The configuration size is the number of changes it includes– in this example, $|C| = 4$.

Dynamic fault model. A dynamic fault model is an interplay between the adversary's power and the following events, which are invoked as part of client operations:

introduce(C): indicates that C is going into use; and
activate(C): indicates that the state transfer to C is complete.

By convention we say that the initial configuration C_{init} is introduced and activated at time 0.

The above events govern the life-cycle of configurations. A configuration C becomes *activated* once an activate(C) event occurs. Note that not all introduced configurations are necessarily activated at some point. A configuration C becomes *expired* once activate(D) occurs s.t. C does not contain D. Intuitively, D reflects events (inclusions or exclusions) that are not reflected in C, and hence C has become "outdated". Our algorithm will enforce a containment order among activated configurations, and will thus ensure that the latest activated one is not expired.

The following two conditions constrain the power of the adversary:

▶ **Definition 1** (liveness conditions).
Availability: The adversary can crash at most a minority of C.membership between the time when introduce(C) occurs and until C is expired.
Weak Oracle: When a client interacts with an expired configuration C, it either receives responses to calls from a majority of $C.membership$, or returns an exception notification $\langle error, D\rangle$ for some activated D, where $C \not\supseteq D$.

Note that such an oracle (sometimes called directory service) is inherently required in order to allow slow clients to find non-expired configurations in an asynchronous system where old configurations may become unavailable [2, 22]. Our oracle definition is weak– in particular, the activated configuration it returns may itself be expired, and different clients may get different responses; it can be trivially implemented using a broadcast mechanism as assumed in some previous works [6, 7], and trivially holds if configurations must remain available as long as some client may access them, as in other previous works [15, 20, 10].

Static versus dynamic objects. A *static object* is one in which clients interact with a fixed configuration. In order to disambiguate a static object, scoped within a configuration C, from a dynamic one, we will label the methods of a static object with a "C.". For every configuration C, as long as a majority of $C.membership$ is alive, clients can use ABD [5] to emulate (static) atomic registers on top of the servers in $C.membership$. We denote:

$C.x \leftarrow value$	A Write($value$) operation to register x in configuration C
$C.x$	A Read of x
$C.collect(array)$	A bulk Read of all the registers in $array$

Since a complete array can be collected from servers using ABD in the same number of rounds as reading a single variable, we count a collect as a single operation for complexity purposes. Note that each register in the array is atomic in itself, but the collect is not atomic.

The methods of a dynamic object are not scoped with any configuration; it can start in some configuration and continue in a different one. A dynamic object's API includes a ChangeConfig operation that allows clients to change the set of servers implementing the object. The implementation of ChangeConfig is object-specific, because it needs to transfer the state of the object across configurations, e.g., the last written value in case of an atomic register.

Clients pass to ChangeConfig a parameter $Proposal \subset Changes$ containing a proposed set of configuration changes. ChangeConfig returns a configuration C s.t. (1) C is activated, (2) $C \supseteq Proposal$, and (3) every configuration introduced or activated by ChangeConfig consists of C_{init} plus a subset of changes proposed by clients before the operation returns.

The liveness guarantee of a dynamic object is that, assuming the number of ChangeConfig proposals is finite, every correct client's operation eventually completes. Note that if the number of ChangeConfig proposals is infinite, it is impossible to ensure liveness for all operations [23].

Usage example. Consider an administrator (a privileged client) who wants to switch server s off and invokes ChangeConfig($\{-s\}$). By liveness, ChangeConfig completes, and by properties (1) and (2), it returns an activated configuration $C \supseteq \{-s\}$. The activation of C expires all configurations that do not contain C, and in particular, those that do not include $-s$. Hence, s is not part of the membership of any unexpired configuration, and by the availability condition, the administrator can safely switch s off immediately once ChangeConfig($\{-s\}$) returns.

4 Reconfiguration Abstraction

We introduce a generic reconfiguration abstraction, which can be used for implementing dynamic objects as we illustrate in the next section. A Reconfiguration abstraction has two operations:

Propose(C, P) for a configuration C and a proposed set of changes P; and
Check(C) for a configuration C.

Propose is used to reconfigure the system, whereas Check is used in order to learn about other clients' reconfiguration attempts. Propose and Check invoke the introduce and activate events. Both Check and Propose return a pair of values $\langle D, S\rangle$, where D is a configuration and S is a *speculation set* containing configurations; when $\langle D, S\rangle$ is returned we say that D is *nominated* by the operation that returns it. Intuitively, a nominated configuration is one that has been introduced and is a candidate for activation. By convention, we say that C_{init}

is nominated at time 0. We assume that the first argument passed to both operations is a nominated configuration.

The first propert of Reconfiguration is validity, which (i) requires Propose(C, P) to include P in the returned nominated configuration; and (ii) does not allow configurations to include spurious changes not proposed by any client. Formally:

D_1 (Validity) (i) If Propose(C, P) returns $\langle D, S\rangle$ for some S, then $D \supseteq P$, and (ii) for every configuration D that is introduced or nominated by an operation *op*, for every $e \in D \setminus C_{init}$, there is a Propose$(C', P')$ for some C' that is invoked before *op* returns s.t. $e \in P'$.

The second property ensures that nominated configuration sizes monotonically increase over time, which is essential for real-time order of operations invoked on objects that use this abstraction:

D_2 (Real-time Order) A configuration D nominated by operation *op* is larger than or equal to every configuration nominated by an operation that strictly precedes *op*.

Since Reconfiguration is weaker than consensus, clients do not agree on a sequence of nominated configurations. Hence, in case some client c_1 proceeds to a configuration C', we want to ensure that if another client c_2 "skips" C', c_2 has C' in its speculation set, and can thus transfer any state that c_1 may have written there to the newer configuration c_2 nominates. This is captured by property S_1(ii) below. Property S_1(i) stipulates that these configurations are also introduced, ensuring a live majority in these configurations in order to allow state transfer.

S_1 (Speculation) If Check(C) or Propose(C, P) returns $\langle D, S\rangle$, then every $C' \in S$ is (i) introduced and (ii) S includes all nominated configurations C' s.t. $|C| \leq |C'| \leq |D|$. As a practical matter, if any C' between C and D has been activated, any C'' s.t. $|C''| < |C'|$ may be omitted.

In addition, we have to define when configurations are activated. Note that an activation of a new configuration leads to expiration of old ones, and thus to possible loss of information stored in them. Therefore, a configuration D is not immediately activated when a Propose returns $\langle D, S\rangle$ for some S. Instead, a configuration C is activated if Check(C) does not report any newer configuration:

A_1 (Activation) If Check(C) returns $\langle C, S\rangle$ for some S, then C is activated.

The liveness property of Reconfiguration is the same as in other dynamic objects [3, 18, 13, 22], namely, if the number of Propose operations is finite, then every operation by a correct client completes.

5 Building Dynamic Objects Using Reconfiguration

We first present a dynamic atomic read/write register emulation using Reconfiguration, and then explain the modifications needed for supporting a dynamic atomic max-register [4]. A formal proof is provided in the full paper [25].

5.1 Dynamic atomic read/write register

Besides the Reconfiguration abstraction, our dynamic register implementation uses a (static) *ranked register* [11] emulation in every configuration, as illustrated in Figure 1a. A ranked register stores a tuple, called *version*, that consists of a value v and a monotonically increasing

Algorithm 1 Dynamic atomic read/write register using Reconfiguration.

Client local variables:

1: configuration C_{curr}, initially C_{init}
2: $TS = \mathbb{N} \times \Pi$ with selectors *num* and *id*
3: $version \in \mathbb{V} \times TS$ with selectors v and ts, initially $\langle v_0, \langle 0,$client's id$\rangle\rangle$
4: $pickTS \in \{true, false\}$, initially *true*.

Code for client $c_i \in \Pi$:

5: **Read()**
6: *transferState*$(Check(C_{curr}), \perp)$
7: *checkConfig*()
8: return $version.v$

9: **Write(v)**
10: *transferState*$(Check(C_{curr}), v)$
11: *checkConfig*()
12: $pickTS \leftarrow true$
13: return ok

14: **ChangeConfig(P)**
15: *transferState*$(Propose(C_{curr}, P), \perp)$
16: *checkConfig*()
17: return C_{curr}

18: **On** $\langle error, D\rangle$ **do**
19: $C_{curr} \leftarrow D$
20: restart operation

21: **procedure** *checkConfig*()
22: $\langle D, S\rangle \leftarrow Check(C_{curr})$
23: **while** $D! = C_{curr}$ **do**
24: *transferState*$(\langle D, S\rangle, \perp)$
25: $\langle D, S\rangle \leftarrow Check(C_{curr})$

26: **procedure** *transferState*$(\langle D, S\rangle, value)$
27: **for each** $C \in S$ **do**
28: $tmp \leftarrow C.RRRead()$
29: **if** $tmp.ts > version.ts$ **then**
30: $version \leftarrow tmp$
31: **if** $value \neq \perp \ \vee \ pickTS = true$ **then**
32: $version \leftarrow \langle value, \langle version.ts.num + 1, i\rangle\rangle$
33: $pickTS \leftarrow false$
34: *D.RRWrite(version)*
35: $C_{curr} \leftarrow D$

timestamp ts, and supports *RRRead*() and *RRWrite*($version$) operations. The sequential specification of a *ranked register* is following: An *RRRead*() returns the version with the highest ts written by an *RRWrite* that precedes it, or $\perp$ if there is no preceding *RRWrite*. Like all static objects in our model, if the configuration where the ranked register is emulated expires, the oracle returns an error.

The basic framework for implementing the Read, Write, and ChangeConfig operations is a loop: (i) Check, (ii) read (using RRread) the highest version from all speculated configurations returned by Check, (iii) write (with RRWrite) the highest version to the configuration nominated by Check, (iv) repeat. The loop terminates when Check does not nominate a new configuration. The specific action of each of the three operations is as follows. A Read simply returns the value of the highest version at the end of the loop. A Write increments the timestamp and writes it with a new value at the beginning of the loop. ChangeConfig proposes a configuration change via Propose instead of Check in the first iteration.

The pseudocode appears in Algorithm 1. The *transferState* method reads the register's version from the entire speculation set S and writes the latest version to the new configuration D. The *checkConfig* method repeatedly calls *transferState* until the configuration returned by *Check* stops changing. During the loop execution, an operation on an expired configuration may incur an exception, with a notification of the form $\langle error, D\rangle$ (see line 18). In this case, the loop is aborted and the operation starts over at configuration D. In case write is restarted after it has chosen a new timestamp, it skips the timestamp selection step.

Algorithm 2 Dynamic atomic max-register using Reconfiguration.

```
   Client local variables:
1:     configuration C_curr, initially C_init
2:     value ∈ 𝕍, initially v_0

   Code for client c_i ∈ Π:

3: MRead()
4:     transferState(Check(C_curr), ⊥)
5:     checkConfig()
6:     return value

7: MWrite(v)
8:     transferState(Check(C_curr), v)
9:     checkConfig()
10:    return ok

11: ChangeConfig(P)
12:    transferState(Propose(C_curr, P), ⊥)
13:    checkConfig()
14:    return C_curr

15: On ⟨error, D⟩ do
16:    C_curr ← D
17:    restart operation

18: procedure checkConfig()
19:    ⟨D, S⟩ ← Check(C_curr)
20:    while D! = C_curr do
21:        transferState(⟨D, S⟩, ⊥)
22:        ⟨D, S⟩ ← Check(C_curr)

23: procedure transferState(⟨D, S⟩, v)
24:    if v ≠ ⊥ then
25:        value ← v
26:    for each C ∈ S do
27:        tmp ← C.MRead()
28:        if tmp > value then
29:            value ← tmp
30:    D.MWrite(value)
31:    C_curr ← D
```

We say that a configuration C becomes *stable* when some version is written to C in step (iii). We refer to the first version written to C as the *opening* version of C. Consider a completed operation (Read, Write, or ChangeConfig) *op* and let C be the last configuration in which *op* writes some version v, we say that *op* commits v in C when it completes. The correctness of the register emulation, proven in the full paper [25], is based on the following key invariant:

▶ **Invariant 2.** For every stable configuration C, the opening version of C is higher than or equal to the highest version committed in any configuration C' s.t. $|C'| < |C|$.

In other words, a larger stable configuration always holds a newer (or equal) version of the register's value than that committed in a smaller activated one.

Complexity. We measure complexity in terms of the number of accesses to low level objects, namely static atomic registers. Note that Read/Write/collect operations on static registers are emulated in a constant number of rounds using ABD. The complexity of the dynamic register's operations is determined by (1) the complexity of the operations inside the Checks invoked during the loop (plus possibly one Propose); and (2) the sum of the sizes of all speculation sets returned by Propose/Check operations in this loop (where the register's implementation performs Reads).

In a run with n ChangeConfig proposals, clearly, the best complexity we can hope for is $O(n)$. In the next section we present our algorithm for Reconfiguration, which achieves the asymptotically optimal $O(n)$ complexity.

5.2 Dynamic atomic max-register

The emulation of a max-register on top of Reconfiguration is similar to the read/write register emulation. It differs in how we keep and transfer the state, i.e., the register's value. First, instead of a (static) ranked register in each configuration, we use a (static) max-register. Second, instead of timestamps, we use the actual written values, that is, a writer writes its value in step (iii) only if it is higher than all the values read in step (ii) (Otherwise, it transfers the highest value it read). The pseudocode appears in Algorithm 2.

6 The Reconfiguration Abstraction Implementation

In this section we present an optimal and modular Reconfiguration implementation. In Section 6.1 we introduce the *Common Set (CoS)* building block, which is used by the Reconfiguration abstraction in every configuration. In Section 6.2 we show how CoS is used for non-optimal Reconfiguration and give the main correctness argument. In Section 6.3 we optimize the algorithm and give the main complexity and correctness claims. Formal proofs can be found in the full paper [25].

6.1 CoS building block

The *Common Set (CoS)* building clock is a static shared object, emulated in every configuration C over a set of (static) registers. Its API consists of a single operation, denoted C.CoS(P), where P is a set of arbitrary values. C.CoS returns an output set of sets satisfying the following:

▶ **Definition 3** (*Common Set* in configuration C).
(CoS_1) Each set in the output is the union of some of the inputs and strictly contains C;
(CoS_2) if a client's input strictly contains C, then its output is not empty;
(CoS_3) there is a common non-empty set in all non-empty outputs; and
(CoS_4) every C.CoS invocation that strictly follows a C.CoS call that returns a non-empty output returns a non-empty output.

For example, consider three concurrent clients that input to $C.CoS$ the sets P1, P2, and P3, all of which contain C. A possible outcome is for their outputs to be $\{P1\}, \{P1, P1 \cup P2\}$, and $\{P1, P2, P3\}$, respectively. The intuitive explanation behind using CoS is that it builds a *common sequence* of configurations inductively: The first configuration in the sequence is C_{init}, the next is the common configuration returned by $C_{init}.CoS$ (property CoS_3), and so on. Although this sequence is not known to the clients themselves, every client observes this sequence starting with some activated configuration. Every configuration in this sequence contains the previous one.

CoS can be implemented directly using consensus or atomic snapshot, as illustrated in [24]. In Algorithm 3, (without the PreCompute function, which is an optimization and will be discussed later), we give an implementation based on DynaStore's weak snapshot [3]. In the pseudocode, we denote by $\bigcup S$ the union of all sets in a set of sets S. If the proposal P strictly contains C, p_i has something new to propose and it writes P into its cell in the "weak" snapshot array *Warr* (lines 9-10). (Note that *Warr* is a static array emulated in the configuration where CoS is implemented). Either way, it collects *Warr* (line 11). In case the collect is not empty, p_i collects *Warr* again and returns the set of collected proposals (lines 12-15). The second collect ensures that the intersection of non-empty outputs includes the first written input, implying CoS_3; the remaining properties are satisfied by a single collect.

Algorithm 3 Efficient CoS; algorithm of client p_i in configuration C; optimization code shaded.

```
1:  Local variables:                                          ▷ flags accessible outside CoS
2:      firstTime set by reconfig and read by CoS
3:      drop set by CoS and read by reconfig

4:  Shared variables (emulated in configuration C):
5:      Boolean startingPoint, initially false                ▷ Is C a starting point for some client
6:      Mapping from client to registers Warr and Sarr , initially {}.

7:  procedure CoS(P)
8:      P ← PreCompute(P)                                     ▷ optimization
9:      if P ⊃ C then
            ▷ Something new to propose
10:         C.Warr[i] ← P
11:     ret ← C.collect(Warr)
12:     if ret = {} then
13:         return ret
14:     else
15:         return C.collect(Warr)

16: procedure PreCompute(P)
17:     if firstTime then
18:         C.startingPoint ← true
19:     C.Sarr[i] ← P
20:     drop ← false
21:     if ¬C.startingPoint then
22:         return P
        ▷ repeat collect until P stops changing.
23:     drop ← true
24:     tmp ← ⋃C.collect(Sarr)
25:     while tmp ≠ P do
26:         P ← tmp
27:         tmp ← ⋃C.collect(Sarr)
28:     return P
```

6.2 Simple Reconfiguration

Given CoS, we can solve Reconfiguration in a generic way as shown in Algorithm 4 (ignore the shaded areas for now). Both Check and Propose use the auxiliary procedure *reconfig*. Propose(C, P) first sets a local variable *proposal* to the union of C and P, whereas Check(C) initiates *proposal* to be C. Both then execute the loop in line 40. Each iteration selects the smallest configuration in *ToTrack*; we say that the iteration *tracks* this configuration. The loop tracks all configurations returned by CoS, smallest to largest, starting with C. In each tracked configuration C', the client introduces C', invokes $C'.CoS(proposal)$ and adds to *proposal* the union of the configurations returned from $C'.CoS$. This repeats for every configuration C' returned from CoS until there are no more configurations to track. Recall that by the liveness condition, if some configuration C' is expired and no longer supports $C'.CoS$, then the client gets in return to $C'.CoS$ an exception with some newer activated configuration C_a. In this case, *reconfig* starts over from C_a. At the end, Propose and Check return *proposal* and the set of all tracked configurations.

The common sequence starts with C_{init}, and is inductively defined as follows: If $C_k.CoS$ has a non-empty output, then C_{k+1} is the smallest common configuration returned by all non-empty $C_k.CoS$s. By CoS_3, all non-empty return values have at least one configuration in common, and if there is more than one such configuration, then we pick the smallest, breaking ties using lexicographic order. By CoS_1, each configuration in the common sequence strictly contains the previous one.

Correctness. The validity property (D_1) immediately follows from CoS property CoS_1 and the observation that *proposal* is set to include P at beginning of *reconfig* and never decreases.

To provide intuition for the remaining properties, we discuss the case in which all operations start in C_{init} and no exceptions occur; the proof for the general case appears in the full paper [25]. Observe that since *proposal* always contains $\bigcup$ *ToTrack* and configurations

Algorithm 4 Generic Reconfiguration algorithm; optimization code shaded.

```
29: Propose(C, P)
30:     return reconfig(C, P)

31: Check(C)
32:     ret ← reconfig(C, {})
33:     if ret = ⟨C, ∗⟩ then activate(C)
34:     return ret

35: procedure reconfig(C, P)
36:     proposal ← P ∪ C
37:     ToTrack ← {C}
38:     speculation ← {}
39:     firstTime ← true
40:     while ToTrack ≠ {} do
41:         C′ ← argmin_{C″∈ToTrack} |C″|                ▷ smallest configuration
42:         introduce(C′)
43:         speculation ← speculation ∪ {C′}
44:         ret ← C′.CoS(proposal)
45:         if ret = ⟨"error", C_a⟩ then                 ▷ C′ is expired - restart from C_a
46:             return reconfig(C_a, proposal)
47:         ToTrack ← (ToTrack ∪ ret) \ {C′}
48:         firstTime ← false
49:         if drop = true then                          ▷ drop old configurations in ToTrack
50:             ToTrack ← ret
51:         proposal ← proposal ∪ ⋃ ToTrack
52:     C_curr ← proposal
53:     return ⟨proposal, speculation⟩
```

are traversed from smallest to largest, we get from property CoS_2 that $C.CoS$ returns an empty set only if C includes *ToTrack*, i.e., C is the last traversed configuration. The key correctness argument is that all nominated configurations belong to the common sequence, and are thus related by containment:

▶ **Lemma 4.** *For every reconfig that returns $\langle D, S\rangle$, D belongs to the common sequence.*

Proof - sketch for the special case (starting in C_{init}, no exceptions). Assume by way of contradiction that D_j is returned by reconfig operation rec_j but does not belong to the common sequence. Note that C_{init} is in the common sequence and is tracked by rec_j. Let $\tilde{C}_j$ be the last configuration tracked by rec_j that belongs to the common sequence. By assumption, $\tilde{C}_j \neq D_j$, and thus, rec_j gets a non-empty output from $\tilde{C}_j.CoS$ (it gets an output since we assume that there are no exceptions). But, this output includes some configuration in the common sequence, so rec_j tracks a configuration in the common sequence after $\tilde{C}_j$. A contradiction.

Liveness follows since (i) every call to CoS returns, either successfully or with an exception; and (ii) tracked configurations are monotonically increasing, and, provided that the number of reconfigurations is finite, they are bounded.

6.3 Optimal Reconfiguration

The key to the efficiency of our new algorithm is in its thrifty CoS implementation, and the signals it conveys to the reconfiguration algorithm, which minimize the number of tracked configurations. To this end, the efficient solution for CoS shares (local) state variables *firstTime* and *drop* with the Reconfiguration implementation.

To explain the intuition behind our algorithm, let us first consider a scenario in which all clients invoke register operations (Read, Write, or ChangeConfig) in the same starting configuration C_0 (e.g., C_0 may be C_{init}), and no exceptions occur. If n of the clients invoke Propose, then there are n sets $P_1, \ldots, P_n$ proposed by *reconfig*(C, P_i) operations. The unoptimized (weak snapshot-based) CoS may return up to 2^n different subsets in CoS responses (assuming many clients invoke Read/Write operations), inducing high complexity.

Our algorithm reduces this complexity by running a pre-computation phase in *PreCompute*, which imposes a containment order on all configurations passed to, and hence returned from, CoS. This is done by running a variant of (strong) atomic snapshot [1] on all client proposals in configuration C_0. Specifically, each process writes its own proposal P (line 19) to the "strong" array *Sarr*, and then (lines 24-27) repeatedly collects the union of all *Sarr* cells into P, until P stops changing. Like an atomic snapshot, this ensures that all results of PreCompute are related by containment. Note, however, that unlike an atomic snapshot, the complexity of this pre-computation is linear in the number of *different* proposals written, rather than in the number of participating processes; if collect encounters a newly written value that does not change the union of written values, PreCompute returns. In case all operations start in C_0, there are no new proposals in other configurations, and so the containment order is preserved throughout the computation. This ensures that the number of different configurations tracked by all clients is at most n.

Next, we account for the case that clients invoke (or restart due to exceptions) their operations in different starting configurations. We have to identify configurations where some client starts, and run PreCompute in them too. To this end, we have clients signal (by raising the startingPoint flag) if a configuration is their starting point. Every client that later runs $C.CoS$ sees this flag true, and executes the pre-computation. If a client p_i sees the flag false in $C.CoS$, p_i does not run the pre-computation. Nevertheless, since p_i checks the flag after writing its value to *Sarr*, p_i's proposal is already in the array before new clients that start in this configuration perform their collects, and so p_i's proposal is contained in theirs. Thus, at this new starting point, all clients obtain proposals that are related by containment among themselves.

The tricky part is that old proposals that were included in *ToTrack* before the new starting point are not necessarily ordered relative to ensuing proposals, as in the following scenario:

- Clients p_1 and p_2 start in C_0 and propose $C_0 \cup \{+a\}$ and $C_0 \cup \{+b\}$, respectively; p_1 gets $\{C_1\}$, where $C_1 = C_0 \cup \{+a\}$, from $C_0.CoS$ and p_2 gets $\{C_1, C_2\}$, where $C_2 = C_0 \cup \{+a, +b\}$.
- Client p_1 tracks C_1, gets an empty set from $C_1.CoS$, and activates it. Client p_3 starts in C_1, (which is now activated), proposes $C_3 = C_1 \cup \{+c\}$ in $C_1.CoS$, and gets $\{C_3\}$.
- Later, p_2 tracks C_1, and gets C_3 in $C_1.CoS$'s output. At this point p_2's *ToTrack* contains C_2 and C_3, neither of which contains the other.

To achieve linear complexity, we have clients *drop* all configurations previously returned from CoS at all the starting points they encounter. One subtle point is ensuring safety in the presence of such drops, and our proof of the general case of Lemma 4 in the full paper [25] addresses this issue.

Intuitively, since the purpose of tracking all configurations is to ensure that clients traverse the common sequence, once we know C is in the common sequence, there is no need to continue to track any configuration older than C. So, the drop is safe.

A second subtle point is preserving linear complexity despite executing PreCompute in multiple starting points. But since (i) the worst-case complexity of a single pre-computation

is linear in the number of different proposals written to it, (ii) each CoS begins with a proposal that reflects all those seen in previous CoSs, and (iii) there are n new proposals overall, the combined complexity of *all* pre-computations is $O(n)$.

Finally, we provide intuition for the complexity of the high-level dynamic atomic register given in Section 5. The full proof, which wraps this intuition into a technical induction, appears in the full paper [25]. Recall that the register emulation performs a loop in which it repeatedly calls Check(C), where C is the configuration returned from the previous Check/Propose, until some Check(C') returns $\langle C', S\rangle$ for some C' and S. The loop performs a constant number of operations in every configuration returned in a speculated set S from Check. Therefore, we want the Checks in this loop to return the optimal number of configurations, and have optimal complexity themselves.

Since all the configurations introduced (and returned in speculation sets) by our algorithm are related by containment, we immediately conclude that the number of configurations returned in speculated sets S of all Checks together is bounded by n. Now we show that the complexity of all Checks combined is $O(n)$. First observe that all Checks combined invoke at most n CoSs. Second, each CoS writes at most three times to shared registers (lines 10, 18, and 19), reads once (in line 21), and performs each of the collects in lines 11, 15, and 24 at most once. Now observe that CoS performs the collect in line 27 only if the previous collect (in line 24 or 27) contained a proposal $P_1 \not\subseteq P$, which means that none of the CoSs collected P_1 before. Since there are at most n proposals, all CoSs together perform the collect in line 27 at most n times. All in all, we get that the complexity of all Checks is $O(n)$.

7 Conclusions

We defined a dynamic model with a clean failure condition that allows an administrator to reconfigure an object and switch a removed server off once the reconfiguration operation completes. In this model, we have captured a succinct abstraction for consensus-less reconfiguration, which dynamic objects like atomic read/write register and max-register may use. We demonstrated the power of our abstraction by providing an optimal implementation of a dynamic register, which has better complexity than previous solutions in the same model.

Acknowledgements. We thank Christian Cachin for helpful comments on our OPODIS 2015 tutorial, which served as the basis for parts of this paper. We thank Eli Gafni for interesting discussions.

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 Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, Jean-Philippe Martin, and Alexander Shraer. Reconfiguring replicated atomic storage: A tutorial. *Bulletin of the EATCS*, 2010.

3 Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic atomic storage without consensus. *J. ACM*, 58(2):7:1–7:32, April 2011.

4 James Aspnes, Hagit Attiya, and Keren Censor. Max registers, counters, and monotone circuits. In *PODC 2009*, pages 36–45. ACM, 2009.

5 Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)*, 42(1):124–142, 1995.

6 Hagit Attiya, Hyun Chul Chung, Faith Ellen, Saptaparni Kumar, and Jennifer L Welch. Simulating a shared register in an asynchronous system that never stops changing. In *International Symposium on Distributed Computing*, pages 75–91. Springer, 2015.

7 Roberto Baldoni, Silvia Bonomi, and Michel Raynal. Implementing a regular register in an eventually synchronous distributed system prone to continuous churn. *Parallel and Distributed Systems, IEEE Transactions on*, 2012.

8 Ken Birman, Dahlia Malkhi, and Robbert Van Renesse. Virtually synchronous methodology for dynamic service replication. *Appears as Appendix A in [4]*, 2010.

9 Vita Bortnikov, Gregory Chockler, Dmitri Perelman, Alexey Roytman, Shlomit Shachor, and Ilya Shnayderman. Frappé: Fast replication platform for elastic services. *LADIS*, 2011.

10 Gregory Chockler, Seth Gilbert, Vincent Gramoli, Peter M Musial, and Alex A Shvartsman. Reconfigurable distributed storage for dynamic networks. *Journal of Parallel and Distributed Computing*, 69(1):100–116, 2009.

11 Gregory Chockler and Dahlia Malkhi. Active disk paxos with infinitely many processes. *Distributed Computing*, 18(1):73–84, 2005.

12 Gregory Chockler and Alexander Spiegelman. Space complexity of fault-tolerant register emulations. In *Proceedings of PODC'17*. ACM, 2017.

13 Eli Gafni and Dahlia Malkhi. Elastic configuration maintenance via a parsimonious speculating snapshot solution. In *DISC 2015*, pages 140–153. Springer, 2015.

14 Seth Gilbert, Nancy Lynch, and Alex Shvartsman. RAMBO II: Rapidly reconfigurable atomic memory for dynamic networks. In *DSN 2013*. IEEE Computer Society, 2003.

15 Seth Gilbert, Nancy A Lynch, and Alexander A Shvartsman. RAMBO: A robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing*, 23(4), 2010.

16 Prasad Jayanti, Tushar Deepak Chandra, and Sam Toueg. Fault-tolerant wait-free shared objects. *J. ACM*, 45(3):451–500, May 1998.

17 Leander Jehl and Hein Meling. The case for reconfiguration without consensus. In *Proceedings of the 2016 ACM symposium on Principles of distributed computing*. ACM, 2016.

18 Leander Jehl, Roman Vitenberg, and Hein Meling. SmartMerge: A new approach to reconfiguration for atomic storage. In *Proceedings of DISC 2015*. Springer, 2015.

19 Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Reconfiguring a state machine. *SIGACT News*, 41(1):63–73, March 2010.

20 Nancy Lynch and Alex A Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Distributed Computing*, pages 173–190. Springer, 2002.

21 Alexander Shraer, Jean-Philippe Martin, Dahlia Malkhi, and Idit Keidar. Data-centric reconfiguration with network-attached disks. In *LADIS*. ACM, 2010.

22 Alexander Spiegelman and Idit Keidar. Dynamic atomic snapshots. In *Proceedings of the 2016 ACM symposium on Principles of distributed computing*. ACM, 2016.

23 Alexander Spiegelman and Idit Keidar. On liveness of dynamic storage. In *Proceedings of SIROCCO 2017*, 2017.

24 Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi. Dynamic reconfiguration: A tutorial. In *International Conference on Principles of Distributed Systems*, 2016.

25 Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi. Dynamic reconfiguration: Abstraction and optimal asynchronous solution, 2017. URL: `https://alexanderspiegelman.github.io/alexanderspiegelman.github.io/DynamicTasks.pdf`.

Brief Announcement: Practical Synchronous Byzantine Consensus*

Ittai Abraham[1], Srinivas Devadas[2], Kartik Nayak[3], and Ling Ren[4]

1 **VMware Research, Herzliya, Israel**
iabraham@vmware.com
2 **MIT, Cambridge, MA, USA**
devadas@mit.edu
3 **University Of Maryland, College Park, MD, USA**
kartik@cs.umd.edu
4 **MIT, Cambridge, MA, USA**
renling@mit.edu

Abstract

This paper presents new protocols for Byzantine state machine replication and Byzantine agreement in the synchronous and authenticated setting. The PBFT state machine replication protocol tolerates f Byzantine faults in an asynchronous setting using $n = 3f + 1$ replicas. We improve the Byzantine fault tolerance to $n = 2f + 1$ by utilizing the synchrony assumption. Our protocol also solves synchronous authenticated Byzantine agreement in fewer expected rounds than the best existing solution (Katz and Koo, 2006).

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases consensus, agreement, Byzantine fault tolerance, replication, synchrony

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.41

1 Introduction

Byzantine consensus is a fundamental problem in distributed computing and cryptography. Broadly speaking, Byzantine consensus considers the problem of reaching agreement among a group of n parties, among which up to f can have Byzantine faults and deviate from the protocol arbitrarily. There exist a few variant formulations for the Byzantine consensus problem. Two theoretical formulations are Byzantine broadcast and Byzantine agreement [4]. In Byzantine broadcast (BB), there is a designated *sender* who tries to broadcast a value; In Byzantine agreement (BA), every party holds an input value. To rule out trivial solutions, both problems have additional validity requirements. BA and BB have been studied under various combinations of timing (synchrony, asynchrony or partial synchrony) and cryptographic assumptions (whether or not to assume digital signatures). It is now well understood that these assumptions drastically affect the bounds on fault tolerance. In particular, BA requires $f < n/3$ under partial synchrony or asynchrony even with digital signatures, but can be solved with $f < n/2$ under synchrony with digital signatures.

A more practice-oriented problem formulation is Byzantine fault tolerant (BFT) state machine replication (SMR) [1]. In this formulation, the goal is to design a replicated service that provides the same interface as a single server, despite some replicas experiencing

* The full version of this paper is available at https://arxiv.org/abs/1704.02397. We use the word "consensus" as a collective term for all variants; other papers have different conventions.


© Ittai Abraham, Srinivas Devadas, Kartik Nayak, and Ling Ren;
licensed under Creative Commons License CC-BY

Byzantine faults. In particular, honest replicas agree on a sequence of values and their *order*, while the validity of the values is left outside the protocol. PBFT is an asynchronous SMR protocol that tolerates $f < n/3$ Byzantine faults [1]. As the first BFT protocol designed for practical efficiency, PBFT has since inspired numerous follow-up works.

Perhaps somewhat surprisingly, we do not yet have a practical solution for Byzantine consensus in the seemingly easier synchronous and authenticated (i.e., with digital signatures) setting. To the best of our knowledge, the most efficient BA protocol with the optimal $f < n/2$ fault tolerance in this setting is due to Katz and Koo [2], which requires in expectation 24 rounds of communication (not counting the random leader election subroutine). The only SMR protocol we know of in this setting is XFT [5]. Relying on an active group of $f + 1$ honest replicas to make progress, XFT is designed to optimize efficiency for small n and f (e.g., $f = 1$). Its performance degrades as n and f increase, especially when $f = \lfloor \frac{n-1}{2} \rfloor$. In that case, among the $\binom{n}{f+1}$ $f + 1$-sized groups in total, only one is all-honest. The simplest variant of XFT, as presented in [5], requires an exponential number of view changes to find that group. The best XFT variant we can think of still requires $\Theta(n^2)$ view changes.

This paper presents efficient Byzantine consensus protocols for the synchronous and authenticated setting tolerating $f < n/2$ faults. Our main focus is BFT SMR, for which our protocol requires amortized 4 rounds per slot independent of n and f. (We say each value in the sequence fills one *slot*.) Meanwhile, our protocol can also solve multi-valued BA and BB for $f < n/2$ in expected 10 rounds assuming a random leader oracle. (The higher round complexity is due to the fact that BA/BB considers a single slot and cannot be amortized.)

1.1 Overview of the Our Protocols

Interestingly, our core protocol draws inspiration from the Paxos protocol [3], which is neither synchronous nor Byzantine fault tolerant. Since our main focus is SMR, we will describe the core protocol with "replicas" instead of "parties". The core of our protocol resembles the synod algorithm in Paxos, but is adapted to the synchronous and Byzantine setting. In a nutshell, it runs in iterations with a unique leader in each iteration (how to elect leaders is left to higher level protocols). Each new leader picks up the states left by previous leaders and drives agreement in its iteration. A Byzantine leader can prevent progress but cannot violate safety. As soon as an honest leader emerges, then all honest replicas reach agreement and terminate at the end of that iteration.

While synchrony is supposed to make the problem easier, it turns out to be non-trivial to adapt the synod algorithm to the synchronous and Byzantine setting while achieving the optimal $f < n/2$ fault tolerance. The major challenge is to ensure *quorum* intersection [3] at one *honest* replica. The core idea of Paxos is to form a quorum of size $f + 1$ before a commit. With $n = 2f + 1$, two quorums always intersect at one replica, which is honest in Paxos. In order to tolerate f Byzantine faults, PBFT uses quorums of size $2f + 1$ out of $n = 3f + 1$, so that two quorums intersect at $f + 1$ replicas, among which one is guaranteed to be honest. At first glance, our goal of one honest intersection seems implausible with the $n = 2f + 1$ constraint. Following PBFT, we need two quorums to intersect at $f + 1$ replicas which seems to require quorums of size $1.5f + 1$. On the other hand, a quorum size larger than $f + 1$ (the number of honest replicas) seems to require participation from Byzantine replicas and thus loses liveness. Our solution is to utilize the synchrony assumption to form a *post-commit quorum* of size $2f + 1$. A post-commit quorum does not affect liveness and intersects with any *pre-commit quorum* (of size $f + 1$) at $f + 1$ replicas. This satisfies the requirement of one honest replica in intersection.

To implement the above quorum intersection idea, each iteration in our core protocol

consists of 4 rounds. The first three rounds are conceptually similar to Paxos: (1) the leader learns the states of the system, (2) the leader proposes a safe value, and (3) every replica sends a commit request to every other replica. If a replica receives $f + 1$ commit requests for the same value, it commits on that value and *notifies* all other replicas about the commit using a 4th round. Upon receiving a notification, other replicas *accept* the committed value and will vouch for that value to future leaders. To tolerate Byzantine faults, we need to add equivocation checks and other proofs of honest behaviors at various steps. We can then apply the core synod protocol to SMR as well as BA/BB.

For SMR, a simple strategy is to rotate the leader role among the replicas after each iteration. Because each honest leader is able to fill at least one slot, the protocol spends amortized 2 iterations (8 rounds) per slot with $f < n/2$ faults. We then improve the protocol to allow a stable leader and only replace the leader if it is not making progress. The improved protocol fills one slot in every iteration (4 rounds). While our view change protocol resembles that of PBFT at a high level, the increased fault threshold $f < n/2$ again creates new challenges. In particular, two views in PBFT cannot make progress concurrently: $f + 1$ honest replicas need to enter the new view to make progress there, leaving not enough replicas for a quorum in the old view. In contrast, with a quorum size of $f + 1$ and $n = 2f + 1$ in our protocol, if a single honest replica is left behind in the old view, the f Byzantine replicas can exploit it to form a quorum and violate safety. Thus, our view change protocol needs to ensure that two honest replicas are never in different views. In the end, our protocol achieves the result in Theorem 1.

▶ **Theorem 1.** *There exists a synchronous leader-based SMR protocol with optimal Byzantine fault tolerance $n = 2f + 1$. If a leader is non-faulty, each decision takes 4 rounds. A view change (replacing a leader) takes 4 rounds.*

To solve BB, we let the designated sender be the leader for the first iteration. After the first iteration, we rotate the leader role among all n parties. It is straightforward to see that this solution achieves both agreement and validity. If the designated sender is honest, every honest party agrees on its value and terminates. Otherwise, the first honest leader that appears down the line will ensure agreement and termination for all honest parties. Assuming we have a random leader oracle, there is a $(f + 1)/(2f + 1) > 1/2$ probability that each leader after the first iteration is honest, so the protocol terminates in expected 2 iterations after the first iteration. To solve BA, we can use the classical transformation from Lamport et al. [4]. These give rise to the results in Theorem 2.

▶ **Theorem 2.** *Assuming a random leader election oracle, there exist synchronous BA and BB protocols for $f < n/2$ that terminate in expected 10 rounds.*

We remark that the $f < n/2$ Byzantine fault tolerance in our protocols is optimal for synchronous authenticated BA and SMR, but not for BB. Our quorum-based approach cannot solve BB in the dishonest majority case ($f \geq n/2$).

References

1 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *OSDI*, pages 173–186. USENIX Association, 1999.

2 Jonathan Katz and Chiu-Yuen Koo. On expected constant-round protocols for byzantine agreement. *J. Comput. Syst. Sci.*, 75(2):91–112, 2009.

3 Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

4 Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.
5 Shengyun Liu, Christian Cachin, Vivien Quéma, and Marko Vukolic. XFT: practical fault tolerance beyond crashes. In *OSDI*, pages 485–500. USENIX Association, 2016.

Brief Announcement: The Synergy of Finite State Machines

Yehuda Afek[1], Yuval Emek[2], and Noa Kolikant[3]

1 **Tel Aviv University, Tel Aviv, Israel**
afek@cs.tau.ac.il
2 **Technion - Israel Institute of Technology, Haifa, Israel**
yemek@technion.ac.il
3 **Tel Aviv University, Tel Aviv, Israel**
noakolikant@mail.tau.ac.il

Abstract

What can be computed by a network of n randomized finite state machines communicating under the *stone age* model [4] (a generalization of the *beeping* model's communication scheme)? The inherent linear upper bound on the total space of the network implies that its global computational power is not larger than that of a randomized linear space Turing machine, but is this tight? The reported reseach answers this question affirmatively for bounded degree networks by introducing a stone age algorithm (operating under the most restrictive form of the model) that given a designated *I/O node*, constructs a *tour* in the network that enables the simulation of the Turing machine's tape. To construct the tour, it is first shown how to 2-*hop color* the network concurrently with building a spanning tree, with high probability.

1998 ACM Subject Classification C.2.1 Distributed networks

Keywords and phrases finite state machines, stone-age model, beeping communication scheme, distributed network computability

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.42

1 Introduction

Synergy, the whole is greater than the sum of its parts, is many times true, however in traditional distributed computing, each node is usually assumed to be as powerful as a Turing machine, hence its local computational power is equivalent to the global computational power of the whole network. Here, we address the computational power of a network of *randomized finite state machines* with a very weak communication scheme (similar to the communication scheme of the *beeping* model), and show that even under these harsh conditions, synergy can be achieved: *the whole is at least as powerful as the sum of its parts.*

Recently, there is a growing interest in the study of networks of sub-silicon devices, including biological networks [1, 6, 8] and networks of man-made nano-devices [7, 3, 2], through the lens of theoretical distributed computing. These are typically large networks of primitive devices that nevertheless perform complicated tasks, thus raising the following question: How do limitations on the local computation and communication capabilities of the individual nodes affect the global computational power of the whole network?

The reported research addresses this question using the *stone age (SA)* model of Emek and Wattenhofer [4] that captures a network of randomized *finite state machines (FSMs)* with very weak communication capabilities (refer to [4] for a formal definition). It has been shown in [4, Sec. 5 (full version)] that an n-node SA network with a path topology can


© Yehuda Afek, Yuval Emek, and Noa Kolikant;
licensed under Creative Commons License CC-BY

simulate a randomized $O(n)$-space Turing machine, referred to hereafter as an RSPACE(n) machine. Little is known though about the global computational power of SA networks with more general topologies and/or more restrictive communication schemes. In th reported research, we shed some light into this unexplored research domain.

2 Sequential Stone Age Machines

We wish to use a SA network to simulate an RSPACE(n) machine $\mathcal{M}$, but before we can describe this simulation, we have to explain how the $O(n)$-bit input I of $\mathcal{M}$, that is normally stored in $\mathcal{M}$'s tape at the beginning of the execution, is provided to the implementing SA algorithm. Clearly, no node in the network can hold more than a constant number of bits of I and unlike [4], where a path topology is assumed, here the network topology is arbitrary and does not (initially) induce any sequential order on the nodes, so storing I in the network nodes before the execution commences does not seem to work. Instead, we introduce the key notion of a *sequential stone age machine (SSAM)*, where I is fed to the SA algorithm bit-by-bit in a sequential fashion.

Formally, given a network $G = (V, E)$, a SSAM is a SA algorithm operating in G that allows an external user to

(1) pick any node $v \in V$ and send to it a designated `I/O_prepare` message;

(2) wait until v sends a designated `I/O_ready` message;

(3) feed v with a sequence of input bits by means of sending a sequence of designated input messages (and receiving a corresponding sequence of acknowledgments from v);

(4) wait until the computational process terminates; and

(5) get the desired output back from v by means of receiving from it a sequence of designated output messages.

We refer to node v picked by the user in (1) as an *I/O node*. To exploit the combined computational power of all the nodes the computational process described in (4) typically involves the whole network. The SSAM is said to be a (T^p, T^{io})-*SSAM* if it is guaranteed that the external user waits at most T^p time between sending the `I/O_prepare` message and receiving the `I/O_ready` message and at most T^{io} time between feeding the input bits and getting back the output bits.

3 Our Results

We prove that any problem that can be solved with high probability (abbreviated *whp* hereafter) by an RSPACE(n) machine in time T can be solved whp on any bounded degree network of n randomized finite state machines, with a designated I/O node, using a $b = 1$ stone age communication scheme, in $O(D + T)$ time, where D denotes the diameter of the network. In the $b = 1$ stone age communication model a node cannot tell whether a received message with text M was sent to it by one neighbor or by more than one. In other words, it takes $O(D)$ time to initialize the SSAM so that it is ready to accept its input, whereas the actual simulation of the RSPACE(n) machine takes $O(T)$ time. Specifically, our main algorithmic contribution is a SA algorithm that given an n-node bounded degree graph $G = (V, E)$ and a designated *root* node $r \in V$, constructs a 2-*hop coloring* of G and a node sequence $\langle S(i)\rangle_{i=0}^{2n-1}$, referred to as a *tour*, that satisfies: (i) every node appears in S exactly twice; (ii) $S(0) = S(2n-1) = r$; and (iii) the state of node $S(i)$ encodes enough information to route a message to $S(i+1 \mod 2n)$ and to $S(i-1 \mod 2n)$ that reaches its destination

in $O(1)$ time for every $0 \leq i \leq 2n-1$; our algorithm terminates with a correct 2-hop coloring and a correct tour in time $O(D)$ whp.

In the SSAM context, the tour S is constructed during phase (2) (while the external user waits for the `I/O_ready` message) with the I/O node serving as the root. This tour can then be employed to simulate a randomized Turing machine $\mathcal{M}$ with a $(2n)$-cell tape in phase (4).

4 Main Technical Challenges

A 2-hop coloring is a useful construction in anonymous networks (see, e.g., [5]) that enables local point-to-point communication under broadcast communication schemes. As discussed in [4, Sec. 4.3], it is fairly easy to design a 2-hop coloring SA algorithm in bounded degree graphs with *bounding parameter* $b = 2$ (refer to [4] for the definition of a bounding parameter). However, here the bounding parameter is set to $b = 1$, thus turning the 2-hop coloring construction into a challenging task because the nodes can no longer verify (deterministically) that their neighborhood does not admit color conflicts. The setting considered in the reported research is even harder since the graph may contain self-loops (unlike the simple graphs considered in [4]).

Our algorithm resolves this issue by coloring the nodes concurrently with growing a tree $\widetilde{T}$ of depth $O(D)$, rooted at the designated root r. The nodes use a randomized test that looks for color conflicts and if a conflict is detected, the tree $\widetilde{T}$ is carefully used to reset the coloring and tree construction processes. It is interesting to point out that without a designated root, it is impossible to obtain even a 1-hop coloring in our setting.

Another source of difficulty that we had to overcome when designing our algorithm stems from the requirement that the algorithm terminates correctly whp. While whp guarantees are common in traditional distributed graph algorithms, they are more challenging to obtain with SA algorithms: the individual nodes do not (and cannot) have any notion of n; nevertheless, the algorithm should err with probability that decreases (polynomially) with n.

References

1 Y. Afek, N. Alon, O. Barad, E. Hornstein, N. Barkai, and Z. Bar-Joseph. A biological solution to a fundamental distributed computing problem. *Science*, 331(6014):183–185, 2011.

2 S. Cannon, J.J. Daymude, D. Randall, and A.W. Richa. A markov chain algorithm for compression in self-organizing particle systems. In *PODC*, pages 279–288, 2016.

3 Z. Derakhshandeh, R. Gmyr, T. Strothmann, R.A. Bazzi, A.W. Richa, and C. Scheideler. Leader election and shape formation with self-organizing programmable matter. In *DNA*, pages 117–132, 2015.

4 Y. Emek and R. Wattenhofer. Stone age distributed computing. In *PODC*, pages 137–146, 2013. The full version can be obtained from `https://ie.technion.ac.il/~yemek/Publications/stone-age.pdf`.

5 Yuval Emek, Christoph Pfister, Jochen Seidel, and Roger Wattenhofer. Anonymous networks: randomization = 2-hop coloring. In *ACM Symposium on Principles of Distributed Computing, PODC*, pages 96–105, 2014.

6 O. Feinerman and A. Korman. *Theoretical distributed computing meets biology: a review*, pages 1–18. Springer Berlin Heidelberg, 2013.

7 O. Michail, I. Chatzigiannakis, and P.G. Spirakis. *New models for population protocols*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2011.

8 S. Navlakha and Z. Bar-Joseph. Distributed information processing in biological and computational systems. *Commun. ACM*, 58(1):94–102, 2014.

Brief Announcement: Compact Self-stabilizing Leader Election in Arbitrary Graphs*

Lélia Blin[1] and Sébastien Tixeuil[2]

1 **Université d'Evry-Val-d'Essonne, CNRS, LIP6 UMR 7606, France**
`lelia.blin@lip6.fr`
2 **Sorbonne Universités, UPMC Univ Paris 06, CNRS, LIP6 UMR 7606, France**
`sebastien.tixeuil@lip6.fr`

Abstract

We present the first self-stabilizing algorithm for leader election in arbitrary topologies whose space complexity is $O(\max\{\log\Delta, \log\log n\})$ bits per node, where n is the network size and Δ its degree. This complexity is sub-logarithmic in n when $\Delta = n^{o(1)}$.

1998 ACM Subject Classification B.8 Performance and Reliability, C.2.1 Distributed Networks, C.2.4 Distributed Systems

Keywords and phrases Leader Election, Self-stabilization, Memory Complexity, Arbitrary Graphs

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.43

1 Context and Motivarion

This paper tackles the problem of designing memory efficient self-stabilizing algorithms for the leader election problem. *Self-stabilization* [5] is a general paradigm to provide recovery capabilities to networks. Intuitively, a protocol is self-stabilizing if it can recover from any transient failure, without external intervention. *Leader election* is one of the fundamental building blocks of distributed computing, as it enables a single node in the network to be distinguished, and thus to perform specific actions. Leader election is especially important in the context of self-stabilization as many protocols for various problems assume that a single leader exists in the network, even after faults occur. *Memory efficiency* relates to the amount of information to be sent to neighboring nodes for enabling stabilization. A small space-complexity induces a smaller amount of information transmission, which *(i)* reduces the overhead of self-stabilization when there are no faults, or after stabilization, and *ii)* facilitates mixing self-stabilization and replication [9].

A foundational result regarding space-complexity in the context of self-stabilizing silent algorithms is due to Dolev et al. [6], stating that in n-node networks, $\Omega(\log n)$ bits of memory per node are required for solving tasks such as leader election. So, only *talkative* algorithms may have $o(\log n)$-bit space-complexity for self-stabilizing leader election. So far, $o(\log n)$-bits solutions only exist for ring shaped networks, and the best protocol to date is due to Blin *et al.* [3], which present a deterministic solution for arbitrary shaped n-rings with $O(\log\log n)$ bits per node.

In general networks, self-stabilizing leader election is tightly connected to self-stabilizing tree-construction. On the one hand, the existence of a leader permits time- and memory-efficient self-stabilizing tree-construction [5]. On the other hand, growing and merging trees

* This work was partially supported by ANR ESTATE.

© Lélia Blin and Sébastien Tixeuil;
licensed under Creative Commons License CC-BY

is the main technique for designing self-stabilizing leader election algorithms in networks, as the leader is often the root of an inward tree [5]. This high space-complexity is due to the implementation of two main techniques, used by all algorithms, and recalled below.

The first technique is the use of a *pointer-to-neighbor* variable, that is meant to designate unambiguously one particular neighbor of every node. For the purpose of tree-construction, pointer-to-neighbor variables are typically used to store the parent node in the constructed tree. Specifically, the parent of every node is designated unambiguously by its identifier, requiring $\Omega(\log n)$ bits for each pointer variable. In principle, it would be possible to reduce the memory to $O(\log \Delta)$ bits per pointer variable in networks with maximum degree Δ, by using node-coloring at distance 2 instead of identifiers to identify neighbors. However, this, in turn, would require the availability of a self-stabilizing distance-2 node-coloring algorithm that uses $o(\log n)$ bits per node. Unfortunately, self-stabilizing distance-2 coloring algorithms [10, 8, 8] use variables of $O(\log n)$ bits per node. To date, no self-stabilizing algorithm implements pointer-to-neighbor variables with space-complexity $o(\log n)$ bits in arbitrary networks.

The second technique for tree-construction or leader election is the use of a *distance* variable that is meant to store the distance of every node to the elected node in the network. Such distance variable is used in self-stabilizing spanning tree-construction for breaking cycles resulting from arbitrary initial state (see [5]). Clearly, storing distances in n-node networks may require $\Omega(\log n)$ bits per node. There are a few self-stabilizing tree-construction algorithms that are not using explicit distance variables (see, e.g., [11, 7, 4]), but their space-complexity is $O(\log n)$ bits per node. Using the general principle of distance variables with space-complexity below $\Theta(\log n)$ bits was attempted by Awerbuch et al. [1], and Blin et al. [2, 3]. These papers distribute pieces of information about the distances to the leader among the nodes according to different mechanisms, enabling to store $o(\log n)$ bits per node, however, these sophisticated mechanisms have only been demonstrated in rings. To date, no self-stabilizing algorithms implement distance variables with space-complexity $o(\log n)$ bits in arbitrary networks.

2 Compact Leader Election

In this "Brief Announcement", we present a self-stabilizing leader election algorithm with space-complexity $O(\max\{\log \Delta, \log\log n\})$ bits in n-node networks with maximum degree Δ. This algorithm is the first self-stabilizing leader election algorithm for arbitrary networks with space-complexity $o(\log n)$ (whenever $\Delta = n^{o(1)}$). It is designed for the standard state model (a.k.a. shared memory model) for self-stabilizing algorithms in networks.

The design of our algorithm requires overcoming several bottlenecks, including the difficulties of manipulating pointer-to-neighbor and distance variables using $o(\log n)$ bits in arbitrary networks. Overcoming these bottlenecks was achieved thanks to the development of sub-routine algorithms, each deserving independent special interest described hereafter.

First, we generalize to arbitrary networks the results proposed [2, 3] for rings, and aiming at publishing the identifiers in a bit-wise manner. This generalization allows us to manipulate the identifiers with just $O(\log\log n)$ bits of memory per node.

Second, we propose the first *silent* self-stabilizing algorithm for distance-2 coloring that breaks the space-complexity of $\Omega(\log n)$ bits per node. More precisely this new algorithm achieves a space-complexity of $O(\max\{\log \Delta, \log\log n\})$ bits per node. As opposed to previous distance-2 coloring algorithms, we do not use identifiers for encoding pointer-to-neighbor variables, but we use a compact representation of the identifiers to break symmetries. This algorithm allows us to design a compact encoding of spanning trees.

Third, we design a new technique to detect the presence of cycles in the initial configuration resulting from a transient failure. This approach does not use distances, but it is based on the uniqueness of each identifier in the network. Notably, this technique can be implemented by a *silent* self-stabilizing algorithm, with space-complexity $O(\max\{\log \Delta, \log\log n\})$ bits per node.

Last but not least, we design a new technique to avoid the creation of cycles during the execution of the leader election algorithm. Again, this technique does not uses distances but maintains a spanning forest, which eventually reduces to a single spanning tree rooted at the leader at the completion of the leader election algorithm. Implementing this technique results in a self-stabilizing algorithm with space complexity $O(\max\{\log \Delta, \log\log n\})$ bits per node.

Due to space constraints, the details of our approach are presented in the companion technical report available as arXiv:1702.07605 (`https://arxiv.org/abs/1702.07605`).

References

1. B. Awerbuch and R. Ostrovsky. Memory-efficient and self-stabilizing network reset. In *PODC*, pages 254–263. ACM, 1994.
2. L. Blin and S. Tixeuil. Compact deterministic self-stabilizing leader election: The exponential advantage of being talkative. In *Proceedings of the 27th International Conference on Distributed Computing (DISC 2013)*, Lecture Notes in Computer Science (LNCS), pages 76–90. Springer Berlin / Heidelberg, 2013.
3. L. Blin and S. Tixeuil. Compact deterministic self-stabilizing leader election on a ring: The exponential advantage of being talkative. *Distributed Computing*, page to appear, 2017.
4. S. Delaët, B. Ducourthial, and S. Tixeuil. Self-stabilization with r-operators revisited. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 3(10):498–514, 2006. `doi:10.2514/1.19848`.
5. S. Dolev. *Self-stabilization*. MIT Press, March 2000.
6. S. Dolev, M. G. Gouda, and M. Schneider. Memory requirements for silent stabilization. *Acta Inf.*, 36(6):447–462, 1999.
7. B. Ducourthial and S. Tixeuil. Self-stabilization with path algebra. *Theoretical Computer Science (TCS)*, 293(1):219–236, February 2003. `doi:10.1016/S0304-3975(02)00238-4`.
8. M. Gradinariu and C. Johnen. Self-stabilizing neighborhood unique naming under unfair scheduler. In *Euro-Par 2001: Parallel Processing, 7th International Euro-Par Conference Manchester, UK August 28-31, 2001, Proceedings*, pages 458–465, 2001. `doi:10.1007/3-540-44681-8_67`.
9. T. Herman and S. V. Pemmaraju. Error-detecting codes and fault-containing self-stabilization. *Inf. Process. Lett.*, 73(1-2):41–46, 2000.
10. T. Herman and S. Tixeuil. A distributed tdma slot assignment algorithm for wireless sensor networks. In *Proceedings of the First Workshop on Algorithmic Aspects of Wireless Sensor Networks (AlgoSensors'2004)*, number 3121 in Lecture Notes in Computer Science, pages 45–58, Turku, Finland, July 2004. Springer-Verlag.
11. J.Beauquier, M. Gradinariu, C. Johnen, and J. O. Durand-Lose. Token-based self-stabilizing uniform algorithms. *J. Parallel Distrib. Comput.*, 62(5):899–921, 2002.

Brief Announcement: A Note on Hardness of Diameter Approximation*

Karl Bringmann[1] and Sebastian Krinninger[2]

1 **Max Planck Institute for Informatics, Saarland Informatics Campus, Germany**
kbringma@mpi-inf.mpg.de
2 **Faculty of Computer Science, University of Vienna, Austria**
sebastian.krinninger@univie.ac.at

Abstract

We revisit the hardness of approximating the diameter of a network. In the CONGEST model, $\tilde{\Omega}(n)$ rounds are necessary to compute the diameter [Frischknecht et al. SODA'12]. Abboud et al. [DISC 2016] extended this result to sparse graphs and, at a more fine-grained level, showed that, for any integer $1 \leq \ell \leq \text{polylog}(n)$, distinguishing between networks of diameter $4\ell+2$ and $6\ell+1$ requires $\tilde{\Omega}(n)$ rounds. We slightly tighten this result by showing that even distinguishing between diameter $2\ell+1$ and $3\ell+1$ requires $\tilde{\Omega}(n)$ rounds. The reduction of Abboud et al. is inspired by recent conditional lower bounds in the RAM model, where the orthogonal vectors problem plays a pivotal role. In our new lower bound, we make the connection to orthogonal vectors explicit, leading to a conceptually more streamlined exposition. This is suited for teaching both the lower bound in the CONGEST model and the conditional lower bound in the RAM model.

1998 ACM Subject Classification G.2.2 Graph algorithms

Keywords and phrases diameter, fine-grained reductions, conditional lower bounds

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.44

1 Introduction

In distributed computing, the diameter of a network is arguably the single most important quantity one wishes to compute. In the CONGEST model, where in each round every vertex can send to each of its neighbors a message of size $O(\log n)$, it is known that $\tilde{\Omega}(n)$ rounds are necessary to compute the diameter [3] even in sparse graphs [1], where n is the number of vertices. With this negative result in mind, it is natural that the focus has shifted towards *approximating* the diameter. In this note, we revisit hardness of computing a diameter approximation in the CONGEST model from a *fine-grained* perspective.

The current fastest approximation algorithm [4], which is inspired by a corresponding RAM model algorithm [5], takes $O(\sqrt{n \log n} + D)$ rounds and computes a $\frac{3}{2}$-approximation of the diameter, i.e., an estimate $\hat{D}$ such that $\lfloor \frac{2}{3} D \rfloor \leq \hat{D} \leq D$, where D is the true diameter. In terms of lower bounds, Abboud, Censor-Hillel, and Khoury [1] showed that $\tilde{\Omega}(n)$ rounds are necessary to compute a $(\frac{3}{2} - \epsilon)$-approximation of the diameter for any constant $0 < \epsilon < \frac{1}{2}$. At a more fine-grained level, they show that, for any integer $1 \leq \ell \leq \text{polylog}(n)$, at least $\tilde{\Omega}(n)$ rounds are necessary to decide whether the network has diameter $4\ell+2$ or $6\ell+1$, thus ruling out any "relaxed" notions of $(\frac{3}{2} - \varepsilon)$-approximation that additionally allow small

* The work of S. Krinninger was partially done while at Max Planck Institute for Informatics. Full version of this paper available at https://arxiv.org/abs/1705.02127.

© Karl Bringmann and Sebastian Krinninger;
licensed under Creative Commons License CC-BY

additive error. We tighten this result by showing that, for any integer $\ell \geq 1$, at least $\tilde{\Omega}(n)$ rounds are necessary to distinguish between diameter $2\ell + 1$ and $3\ell + 1$.

The reduction of Abboud et al. [1] is inspired by recent work on conditional lower bounds in the RAM model, where the *orthogonal vectors problem* plays a pivotal role. In particular, the Orthogonal Vectors Hypothesis (OVH) is a weaker "polynomial-time analogue" of the Strong Exponential Time Hypothesis (SETH); it is well-known that SETH implies OVH. In our new lower bound, we make the connection to orthogonal vectors explicit: we consider a communication complexity version of orthogonal vectors that we show to be hard *unconditionally* by a reduction from set disjointness and then devise a reduction from orthogonal vectors to diameter approximation. The latter reduction also has implications in the RAM model. We show that under OVH, for any integer $1 \leq \ell \leq n^{o(1)}$, there is no algorithm that distinguishes between graphs of diameter 2ℓ and 3ℓ with running time $O(m^{2-\delta})$ for some constant $\delta > 0$, where m is the number of edges of the graph. This tightens the result of Cairo, Grossi, and Rizzi [2], who provide the same lower bound under the stronger hardness assumption SETH. To summarize, our approach is more streamlined than in previous works [3, 2, 1], allowing for a more unified view of CONGEST model and RAM model lower bounds.

2 Reduction via Orthogonal Vectors

Set disjointness is a problem in communication complexity between two players, called Alice and Bob, in which Alice is given an n-dimensional bit vector x and Bob is given an n-dimensional bit vector y and the goal for Alice and Bob is to find out whether there is some index k at which both vectors contain a 1, i.e., such that $x[k] = y[k] = 1$. The relevant measure in communication complexity is the number of bits exchanged by Alice and Bob in any protocol that Alice and Bob follow to determine the solution. A classic result states that any such protocol requires Alice and Bob to exchange $\Omega(n)$ bits to solve set disjointness.

In the orthogonal vectors problem, Alice is given a set of bit vectors $L = \{l_1, \ldots, l_n\}$ and Bob is given a set of bit vectors $R = \{r_1, \ldots, r_n\}$, and the goal for them is to find out if there is a pair of orthogonal vectors $l_i \in L$ and $r_j \in R$ (i.e., such that, for every $1 \leq k \leq d$, $l_i[k] = 0$ or $r_j[k] = 0$). We give a reduction from set disjointness to orthogonal vectors.

▶ **Theorem 1.** *Any b-bit protocol for the orthogonal vectors problem in which Alice and Bob each hold n vectors of dimension $d = 2\lceil \log n \rceil + 3$, gives a b-bit protocol for the set disjointness problem where Alice and Bob each hold an n-dimensional bit vector.*

▶ **Corollary 2.** *Any protocol solving the orthogonal vectors problem with n vectors of dimension $d = 2\lceil \log n \rceil + 3$, requires Alice and Bob to exchange $\Omega(n)$ bits.*

We now establish hardness of distinguishing between networks of diameter $2\ell + q$ and $3\ell + q$, where $\ell \geq 1$ and in the CONGEST model $q \geq 1$, whereas in the RAM model $q \geq 0$. To unify the cases of odd and even ℓ, we introduce an additional parameter $p \in \{0, 1\}$ and change the task to distinguishing between networks of diameter $4\ell' - 2p + q$ and $6\ell' - 3p + q$ for integers $\ell' \geq 1$, $q \geq 0$, and $p \in \{0, 1\}$. This covers the original question: if ℓ is even, then set $\ell' := \ell/2$ and $p := 0$ and if ℓ is odd, then set $\ell' := \lceil \ell/2 \rceil$ and $p := 1$.

Given an orthogonal vectors instance $\langle L := \{l_1, \ldots, l_n\}, R := \{r_1, \ldots, r_n\} \rangle$ of d-dimensional vectors and parameters $\ell \geq 1$, $q \geq 0$, and $p \in \{0, 1\}$, we define an unweighted undirected graph $G := G_{L,R,\ell,p,q}$ as follows. The graph G contains the following *exterior* vertices: $u_1^L, \ldots, u_n^L$, $u_1^R, \ldots, u_n^R$, $v_1^L, \ldots, v_n^L$, $v_1^R, \ldots, v_n^R$, $w_1^L, \ldots, w_d^L$, $w_1^R, \ldots, w_d^R$, x^L, x^R, y^L, and y^R. These exterior vertices are connected by paths as depicted in Figure 1, where

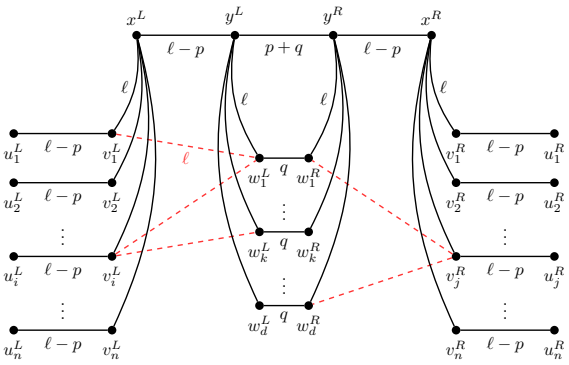


Figure 1 Visualization of the graph $G := G_{L,R,\ell,p,q}$ used in our reduction from orthogonal vectors to diameter distinction. The red, dashed edges encode the orthogonal vectors instance.

each path introduces a separate set of interior vertices. In particular, the instance $\langle L, R\rangle$ is encoded as follows: for every $1 \leq i \leq n$ and every $1 \leq k \leq d$, if $l_i[k] = 1$, then add a path from v_i^L to w_k^L of length ℓ, and if $r_i[k] = 1$, then add a path from v_i^R to w_k^R of length ℓ.

▶ **Theorem 3.** *Let $\langle L, R\rangle$ be an orthogonal vectors instance of two sets of d-dimensional vectors of size n each and let $\ell \geq 1$, $p \in \{0, 1\}$, and $q \geq 0$ be integer parameters. Then the unweighted, undirected graph $G := G_{L,R,\ell,p,q}$ has $O(nd\ell + dq)$ vertices and edges and its diameter D has the following property: if $\langle L, R\rangle$ contains an orthogonal pair, then $D = 6\ell - 3p + q$, and if $\langle L, R\rangle$ contains no orthogonal pair, then $D = 4\ell - 2p + q$.*

For the CONGEST model, observe that G has a small cut of size $d + 1$ between its left hand side and its right hand side. A standard simulation argument, where communication between Alice and Bob is limited to messages sent along the small cut, yields our main result.

▶ **Corollary 4.** *In the CONGEST model, any algorithm distinguishing between graphs of diameter $2\ell + q$ and $3\ell + q$ when $\ell \geq 1$ and $q \geq 1$ requires $\Omega(n/((\ell + q)\log^3 n))$ rounds.*

In the RAM model, the Orthogonal Vectors Hypothesis (OVH) states that there is no algorithm that decides whether a given orthogonal vectors instance contains an orthogonal pair in time $O(n^{2-\delta}\,\mathrm{poly}(d))$ for some constant $\delta > 0$. Under this hardness assumption, our reduction has the following straightforward implication.

▶ **Corollary 5.** *In the RAM model, under OVH, there is no algorithm distinguishing between graphs of diameter $2\ell + q$ and graphs of diameter $3\ell + q$ when $\ell \geq 1$ and $q \geq 0$ in time $O(m^{2-\delta}/(\ell + q)^{2-\delta})$ for any constant $\delta > 0$.*

References

1 Amir Abboud, Keren Censor-Hillel, and Seri Khoury. Near-linear lower bounds for distributed distance computations, even in sparse networks. In *DISC*, pages 29–42, 2016.

2 Massimo Cairo, Roberto Grossi, and Romeo Rizzi. New bounds for approximating extremal distances in undirected graphs. In *SODA*, pages 363–376, 2016.

3 Silvio Frischknecht, Stephan Holzer, and Roger Wattenhofer. Networks cannot compute their diameter in sublinear time. In *SODA*, pages 1150–1162, 2012.

4 Stephan Holzer, David Peleg, Liam Roditty, and Roger Wattenhofer. Brief announcement: Distributed 3/2-approximation of the diameter. In *DISC*, pages 562–564, 2014.

5 Liam Roditty and Virginia Vassilevska Williams. Fast approximation algorithms for the diameter and radius of sparse graphs. In *STOC*, pages 515–524, 2013.

Brief Announcement: Black-Box Concurrent Data Structures for NUMA Architectures

Irina Calciu[1], Siddhartha Sen[2], Mahesh Balakrishnan[3], and Marcos K. Aguilera[4]

1 **VMware Research, Palo Alto, CA, USA**
2 **Microsoft Research, New York, NY, USA**
3 **Yale University, New Haven, CT, USA**
4 **VMware Research, Palo Alto, CA, USA**

Abstract

Recent work introduced a method to automatically produce concurrent data structures for NUMA architectures. We present a summary of that work.

1998 ACM Subject Classification D.1.3 Concurrent Programming

Keywords and phrases concurrent data structures, log, NUMA architecture, replication

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.45

1 Introduction

Modern data centers increasingly employ non-uniform memory access (NUMA) machines with multiple NUMA *nodes*. Each node consists of some (hardware) threads and local memory. A thread can access memory in any node, but accessing local memory is faster than remote memory at another node. To obtain the best performance, concurrent data structures must take this fact into consideration: they must be NUMA-*aware*. Unfortunately, designing concurrent data structures is difficult, and NUMA-awareness makes it harder because the algorithm must try to reduce the number of remote memory accesses.

In this work, we show how to obtain NUMA-aware concurrent data structures *automatically*. We propose an algorithm called Node Replication or NR, which can transform any sequential data structure into a NUMA-aware concurrent data structure. In a nutshell, NR relies on three techniques: replication, an efficient log data structure, and flat combining.

The data structures produced by NR are linearizable [3], thus providing a strong consistency guarantee: an operation appears to take effect instantaneously at some point in time between the operation's invocation and response.

This brief announcement summarizes work published recently [1].

2 The Node Replication algorithm

The NR algorithm takes a sequential data structure and replicates it across NUMA nodes to promote locality of accesses. We use flat combining [2] within each node to ensure safe access to each replica (§2.2). Across nodes, we use a shared log data structure to provide consistency across replicas (§2.1). The log is implemented as a circular buffer allocated from the memory of one of the nodes, providing efficient memory management. Only the combiner within each node accesses the log, ensuring the amount of sharing and contention between nodes is kept to a minimum.


© Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera;
licensed under Creative Commons License CC-BY

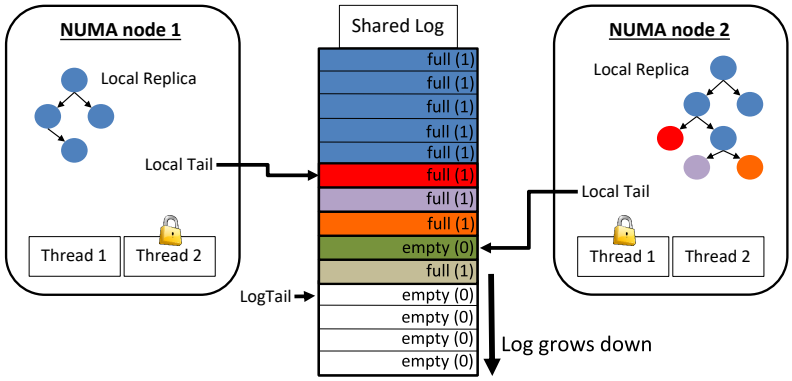


Figure 1 NR algorithm, shared log and per-node replicas. *logTail* indicates the first unreserved entry in the log. Each *localTail* indicates the next operation in the log to be executed on each local replica. Threads on the same node share a replica and coordinate access to the replica using a combiner lock. Threads 2 and 1 are the combiners for nodes 1 and 2, respectively. Node 1's replica executed 5 operations from the log. Node 2's replica executed 3 more operations and found a reserved entry that is not yet filled. A combiner must wait for all empty entries preceding its batch in the log, so Thread 1 cannot proceed until the entry is filled. Readers can return when they find an empty entry without waiting (§2.3).

2.1 Sharing across NUMA nodes: the log data structure

A shared log data structure is used to encode the state of the concurrent data structure. This log contains a representation of all the update operations performed on the data structure, giving a total order of these operations. Read-only operations, which do not modify the data structure, are not included in the log.

There are a few important variables that act as indices on the log. First, *logTail* is a global index in the log that points to the next empty entry in the log. Second, each NUMA node has an index into the log, *localTail*, which indicates how far each replica has been updated from the log. When the *localTail*s differ, replicas will be in different states.

A single elected thread on each node, the *combiner*, can access the log on behalf of all the concurrent threads executing at the same time on that node. This thread collects all the other operations and writes them to the log in a batch using a Compare-And-Swap (CAS) instruction to first reserve space in the log, and then using normal stores to write the operations. This batching strategy decreases the amount of contention on the log. Next, the combiner reads the old operations from the log, starting with the entry at *localTail*, and updates the local replica with the operations logged before its own batch. The combiner may find empty entries in the log, identified by a bit in the entry. If so, it must wait until the operation becomes available in the entry. Figure 1 shows two combiners on different NUMA nodes reading from the log to update their local replicas.

The log is implemented as a circular buffer, for efficient memory management. The bit indicating empty entries alternates as the log wraps around. Another index in the log, *logMin*, indicates which entries are safe to write to – the ones that have been applied to all replicas. This index is updated lazily and in a lock-free manner, by the thread that writes to the last available safe entry in the log. This thread checks the values of all *localTail* indices on all NUMA nodes and updates *logMin* to the smallest one. The algorithm could block if a node is slow to update its replica, but in practice this is not a problem if at least one thread on each node accesses the data structure regularly.

2.2 Sharing within a NUMA node: combining

On each NUMA node, safe access to the replica on that node is provided by flat combining [2]. Using this method, threads announce their operations in per-thread slots[1] and then try to acquire a combiner lock. The thread who succeeds becomes the *combiner*: it collects operations from all other threads, writes them to the shared log, and executes them sequentially on the local replica as described above. Only the combiner on each NUMA node accesses the shared log (Figure 1).

2.3 Read-only operations

Flat combining treats update and read-only operations in the same way: the combiner executes all operations sequentially. In contrast, NR optimizes read-only operations, by extending the local replicas with a readers-writer lock that enables the threads performing read-only operations to proceed in parallel. Moreover, read-only operations do not need to be inserted in the log, because they do not need to be executed at all replicas. However, before returning a value read from the local replica, a read-only operation needs to ensure that the replica is fresh so that it does not return a stale value. We use a new index in the shared log, *completedTail*, to indicate all completed operations in the log. The read-only operation needs to read this index as it starts executing and ensure that the replica is updated at least until this index. A thread performing a read-only operation could either wait for a co-located combiner to update the replica, or acquire a writer lock and update the replica itself if no co-located combiner exists.

3 Conclusion

This brief announcement summarized NR, a method to automatically transform sequential data structures into concurrent data structures optimized for NUMA architectures. NR replicates the sequential data structure on each NUMA node, using a shared log to synchronize the replicas across nodes and flat combining to synchronize access to each replica. We implemented and evaluated NR (not shown in this paper) [1]. We found that NR outperforms prior black-box techniques for concurrent data structures and, under high contention, can even perform better than specialized data structures.

References

1 Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. Black-box concurrent data structures for NUMA architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 207–221. ACM, 2017. doi:10.1145/3037697.3037721.

2 Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 355–364, June 2010. doi:10.1145/1810479.1810540.

3 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990. doi:10.1145/78969.78972.

[1] We refer to the locations used by the combiner as *slots*, and to the locations in the shared log as *entries*.

Brief Announcement: Crash-Tolerant Consensus in Directed Graph Revisited

Ashish Choudhury[*1], Gayathri Garimella[†2], Arpita Patra[‡3], Divya Ravi[4], and Pratik Sarkar[5]

1 **International Institute of Information Technology Bangalore, India**
ashish.choudhury@iiitb.ac.in
2 **International Institute of Information Technology Bangalore, India**
AnnapurnaGayathri.Garimella@iiitb.org
3 **Computer Science and Automation, Indian Institute of Science, Bangalore, India**
arpita@csa.iisc.ernet.in
4 **Computer Science and Automation, Indian Institute of Science, Bangalore, India**
divya.ravi@csa.iisc.ernet.in
5 **Computer Science and Automation, Indian Institute of Science, Bangalore, India**
pratik.sarkar@csa.iisc.ernet.in

Abstract

We revisit the problem of distributed consensus in directed graphs tolerating crash failures; we improve the round and communication complexity of the existing protocols. Moreover, we prove that our protocol requires the optimal number of communication rounds, required by any protocol belonging to a specific class of crash-tolerant consensus protocols in directed graphs.

1998 ACM Subject Classification B.8.1 Reliability, Testing and Fault-Tolerance, E.1 Distributed data structures

Keywords and phrases Directed graph, Consensus, Crash failure, Round complexity

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.46

1 Introduction

A crash-tolerant reliable consensus protocol [3] allows a set of n mutually distrusting parties, each with some private input, to agree on a common output. This is ensured even in the presence of a *computationally unbounded* centralized adversary, who may crash any f out of the n parties and try to prevent the remaining parties from achieving consensus. While most of the prior work in the literature (see [2, 1, 4] and their references) have considered the undirected graph model, where parties are assumed to be a part of a *complete undirected graph*, in [5], necessary and sufficient condition for crash-tolerant consensus is presented for a more generic *directed* graph model. We revisit the round complexity of crash-tolerant consensus protocols in the latter model.

* The author would like to acknowledge the financial support from Infosys foundation.
† The author would like to acknowledge the financial support from Infosys foundation.
‡ The author would like to acknowledge the financial support by INSPIRE Faculty Fellowship (DST/INSPIRE/04/2014/015727) from Department of Science & Technology, India.

© Ashish Choudhury, Gayathri Garimella, Arpita Patra, Divya Ravi, and Pratik Sarkar;
licensed under Creative Commons License CC-BY

Informally, in directed graphs the necessary condition for crash-tolerant consensus demands that even if an arbitrary set of f nodes crashes, there should still exist a special node in the graph, called *source*, which should have a directed path to every other node in the remaining graph. The authors in [5] proved the sufficiency of this condition by presenting two consensus protocols, a consensus protocol for the binary domain and a multi-valued consensus protocol for an arbitrary domain $\{0, \ldots, K\}$. These protocols belong to a special class of protocols, based on "flooding". In more detail, the protocols consist of several "phases" of d rounds of "send-receive-update", where d is called the *crash-tolerant diameter* of a directed graph. Informally, d is the maximum distance of any node from a potential source in the graph. Thus any given potential source can propagate its value to all remaining nodes in a single phase within the d rounds of flooding. In a round every node (including the source) broadcasts its value to its neighbours. At the end of the round, each node "updates" its value, by locally applying an update function to the received values. In the subsequent round, nodes broadcast their updated value. The two types of update function applied are a min function for a *min phase* and a max function for a *max phase*. The min (resp. max) function requires nodes to update their value by taking the minimum (resp. maximum) of all the received values (including its own value).

The binary consensus protocol of [5] requires $2f + 2$ alternate min-max phases, each with d rounds. The round complexity of the protocol is $(2f + 2) \cdot d$ rounds and the communication complexity is $\mathcal{O}(nfd)$ bits. In [5] the authors claimed that their binary consensus protocol cannot be extended trivially to the multi-valued case. They present a multi-valued consensus protocol, which requires $(2f + 2) \cdot d \cdot K$ rounds of interaction and communication complexity is $\mathcal{O}(nfdK \log K)$ bits. Clearly the protocol has exponential round and communication complexity, as $K = 2^{\log K}$ (K is the domain size).

Our Results. In this work, we improve the round and communication complexity of the min-max based consensus protocols of [5]. We consider the binary consensus protocol of [5] and observe that if instead of d, we allow $d+1$ rounds of communication in each of the phases, then it is possible to achieve consensus with just $f+2$ alternate min-max phases, thus making the round complexity $(f + 2)(d + 1)$. We then show an optimization of our protocol, where we allow *only* d rounds in the *first* and the *last* phase, thus reducing the round complexity to $(f + 2)(d + 1) - 2$. Interestingly, we show that our protocol works even for the multi-valued case, with *no* modifications what so ever. Thus, unlike [5], the round complexity of our multi-valued consensus protocol is *independent* of K. The communication complexity of our protocol is $\mathcal{O}(nfd \log K)$ bits and for significantly large values of K our protocol improves upon the round and communication complexity of the multi-valued consensus protocol of [5]. Moreover, we improve the number of rounds for the binary consensus, for every $f, d \geq 2$.

We also address the problem of lower bound on the minimum number of rounds required by any crash-tolerant consensus protocol in a directed graph, based on min-max strategy and derive three interesting lower bounds. We first consider the case, where only $f + 1$ min-max phases are allowed in the protocol and with *no restriction* on the number of communication rounds in each phase. We show that it is impossible to achieve crash-tolerant consensus within $f + 1$ phases. Next we consider min-max based consensus protocols with *at least* d rounds in each phase. For such protocols, we show that it is impossible to achieve consensus in general with $(f + 2)(d + 1) - 3$ rounds in total. This implies that our min-max based protocol with $(f + 2)(d + 1) - 2$ rounds is *round optimal*. Finally we consider min-max based consensus protocols with *exactly* d rounds of communication in each phase. Note that the consensus protocols of [5] belong to this class. For several values of f and d, we show that the

minimum number of phases required to achieve consensus in this case is $2f+2$, thus showing that the binary consensus protocol of [5] has the *optimal* number of communication rounds. The lower bounds establish that our protocol is the best in terms of the round complexity if one is interested to design consensus protocols based on min-max strategy.

High Level Description of Our Protocol. Our starting point is the binary consensus protocol of [5] with $2f+2$ phases, each with d rounds. The correctness of their protocol is based on the guaranteed occurrence of two *consecutive* crash-free phases, among the $2f+2$ alternate min-max phases, within which consensus is shown to be achieved. We observe that if instead of d rounds, we allow $d+1$ rounds in each phase then consensus can be achieved if we either have two consecutive crash-free phases or a crashed phase followed by a crash-free phase, provided *only one* node crashes during the crashed phase. The base of our observation is the following: if during the crashed phase the single node to be crashed is a non-source node, then it is equivalent to having two consecutive crash-free phases (with source node(s) being unaltered) and so consensus will be achieved within these two phases. On the other hand, if during the crashed phase the single node to be crashed is a source node, then at least one of new source nodes will be at a distance of *one* from the crashed source (this observation lies at the heart of our protocol). So if at all the crashed source node sends its value to one of the new source node before crashing, there will be still d rounds left for this new source node in the crashed phase to further propagate the crashed source node's value in the remaining graph. So in essence, we still get the effect of two consecutive crash-free phases. We further show that with $f+2$ alternate min-max phases, there always exist either two crash-free phases or a crashed phase with a single crash, followed by a crash-free phase.

We find that the above ideas are applicable even for the multi-valued case. For simplicity, we consider the case when there are two crash-free phases and without loss of generality, let these be a min phase followed by a max phase. Let λ^{min} be the least value among the source nodes at the beginning of crash-free min phase. If the non-source nodes have their value greater than or equal to λ^{min} at the beginning of this phase, then clearly consensus will be achieved at the end of this min phase itself; this is because each node will update their value to λ^{min} at the end of the min phase. On the other hand, if some *non-source* node has a value smaller than λ^{min} at the beginning of the crash-free min phase, then consensus will not be achieved in this phase. However, at the end of this min phase, the modified values of all the nodes (both source as well as non-source) is upper bounded by λ^{min}; moreover all the *source nodes* will have λ^{min} as their modified value. Hence in the next crash-free phase which is a max phase, the value λ^{min} of the source nodes will be the maximum value in the graph and hence consensus will be achieved at the end of the crash-free max phase. The above argument also works for the case when there is a crashed phase followed by a crash-free phase, where it is guaranteed that exactly one node crashes during the crashed phase. The complete formal details of the protocols and the lower bounds will be available in the full version of the article.

References

1 H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulation and Advanced Topics.* Wiley series on Parallel and Distributed Computing, 2004.

2 N. A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

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

4 L. Tseng. Recent Results on Fault-Tolerant Consensus in Message-Passing Networks. In *SIROCCO*, volume 9988 of *Lecture Notes in Computer Science*, pages 92–108, 2016.
5 L. Tseng and N. H. Vaidya. Crash-Tolerant Consensus in Directed Graphs. *CoRR*, abs/1412.8532, 2014. Full version appeared in PODC 2015.

Brief Announcement: On the Parallel Undecided-State Dynamics with Two Colors*

Andrea Clementi[1], Luciano Gualà[1], Francesco Pasquale[1], and Giacomo Scornavacca[2]

1 **Università Tor Vergata di Roma, Rome, Italy**
clementi@mat.uniroma2.it
2 **Università Tor Vergata di Roma, Rome, Italy**
guala@mat.uniroma2.it
3 **Università Tor Vergata di Roma, Rome, Italy**
pasquale@mat.uniroma2.it
4 **Università degli Studi dell'Aquila, L'Aquila, Italy**
giacomo.scornavacca@graduate.univaq.it

Abstract

The Undecided-State Dynamics is a well-known protocol that achieves Consensus in distributed systems formed by a set of n anonymous nodes interacting via a communication network. We consider this dynamics in the parallel PULL communication model on the complete graph for the binary case, i.e., when every node can either support one of two possible colors or stay in the undecided state. Previous work in this setting only considers initial color configurations with no undecided nodes and a large bias (i.e., $\Theta(n)$) towards the majority color. A interesting open question here is whether this dynamics reaches consensus quickly, i.e. within a polylogarithmic number of rounds. In this paper we present an unconditional analysis of the Undecided-State Dynamics which answers to the above question in the affirmative. Our analysis shows that, starting from any initial configuration, the Undecided-State Dynamics reaches a monochromatic configuration within $O(\log^2 n)$ rounds, with high probability (w.h.p.). Moreover, we prove that if the initial configuration has bias $\Omega(\sqrt{n \log n})$, then the dynamics converges toward the initial majority color within $O(\log n)$ round, w.h.p. At the heart of our approach there is a new analysis of the symmetry-breaking phase that the process must perform in order to escape from (almost-)unbiased configurations. Previous symmetry-breaking analysis of consensus dynamics essentially concern sequential communication models (such as Population Protocols) and/or symmetric updated rules (such as majority rules).

1998 ACM Subject Classification F.2 Theory of Computation – Analysis of Algorithms and Problem Complexity

Keywords and phrases Distributed Consensus, Dynamics, Gossip model, Markov chains

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.47

1 Introduction

Strong research interest has been recently focussed on the study of simple, local mechanisms for *Consensus* problems in distributed systems [3, 2, 11, 12, 16, 17]. In a basic setting of the consensus problem, the system consists of a set of n anonymous nodes that run elementary operations and interact by exchanging messages. Every node initially supports a *color* chosen

* The full version of the paper can be found at https://arxiv.org/abs/1707.05135.


© Andrea Clementi, Luciano Gualà, Francesco Pasquale, and Giacomo Scornavacca;
licensed under Creative Commons License CC-BY
31st International Symposium on Distributed Computing (DISC 2017).
Editor: Andréa W. Richa; Article No. 47; pp. 47:1–47:4

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

from an alphabet Σ and a *Consensus Protocol* is a local procedure that, starting from any color configuration, let the system converge to a monochromatic configuration. The consensus is *valid* if the *winning* color is one among those initially supported by at least one node. Moreover, once the system reaches a consensus configuration it will stay there forever.

We study the consensus problem in the $\mathcal{PULL}$ *model* [8, 10, 14] in which, at every round, each active node contacts one neighbor uniformly at random to pull information. A well-studied consensus protocol is the *Undecided-State Dynamics* (for short, the *U-Dynamics*) in which the state of a node can be either a color or the *undecided state*. When a node is activated, it pulls the state of a random neighbors and updates its state according to the following updating rule: If a colored node pulls a different color from its current one, then it becomes undecided, while in all other cases it keeps its color; moreover, if the node is in the undecided state then it will take the state of the pulled neighbor. The U-Dynamics has been studied in both *sequential* and *parallel* models. As for the sequential model, [3] provides an unconditional analysis showing (among other results) that the U-Dynamics solves the *binary* consensus problem (i.e. when $|\Sigma| = 2$) in the complete graph within $\mathcal{O}(n \log n)$ activations (and, thus in $\mathcal{O}(\log n)$ "parallel" time), *w.h.p.*[1] As for the parallel $\mathcal{PULL}$ model, even though it is easy to verify that the U-Dynamics achieves consensus in the complete graph (w.h.p.), the convergence time of this dynamics is still an interesting open issue, even in the binary case. We remark that the stochastic process yielded by the parallel dynamics significantly departs from the process yielded by the sequential one. A crucial difference lies in the random number of nodes that may change color at every round: In the sequential model, this is at most one, while in the parallel one *all* nodes may change state in one shot and indeed, for most phases of the process, the expected number of changes is linear in n. It thus turns out that the probabilistic arguments used in the analysis of [3] appear not useful in the parallel setting. In [5], the author analyze the U-Dynamics in the parallel $\mathcal{PULL}$ model on the complete graph when the alphabet Σ has size k, where $k = o(n^{1/3})$. The analysis in [5] considers this dynamics as a protocol for *Plurality Consensus* [2, 3, 15], a variant of Consensus, where the goal is to reach consensus on the color that was initially supported by the *plurality* of the nodes: Their analysis requires that the initial configuration must have a relatively-large *bias* $s = c_1 - c_2$ between the size c_1 of the (unique) initial plurality and the size c_2 of the second-largest color. More in details, in [5] it is assumed that $c_1 \geq \alpha c_2$, for some absolute constant $\alpha > 1$ and, thus, this condition for the binary case would result into requiring a very-large initial bias, i.e., $s = \Theta(n)$. This analysis clearly does not show that the U-Dynamics efficiently solves the binary consensus problem, mainly because it does not manage *balanced* initial configurations.

Our results. We prove that, starting from any color configuration[2] on the complete graph, the U-Dynamics reaches a monochromatic configuration (thus consensus) within $\mathcal{O}(\log^2 n)$ rounds, w.h.p. This bound is almost tight since, for some (in fact, a large number of) initial configurations, the process requires $\Omega(\log n)$ rounds to converge. Not assuming a large initial bias of the majority color significantly complicates the analysis. Indeed, the major challenges arise from (almost) *balanced* initial configurations where the system needs to break symmetry. A key ingredient of our analysis is a suitable application of the *martingale optional stopping theorem*. While the use of that theorem is standard in the analysis of *sequential* processes of interacting particles that can be modeled as *birth-and-death* chains, our new approach allows

[1] As usual, we say that an event $\mathcal{E}_n$ holds *w.h.p.* if $\mathbf{P}(\mathcal{E}_n) \geq 1 - n^{-\Theta(1)}$

[2] Our analysis also considers initial configurations with undecided nodes.

us to analyze the process yielded by running the U-Dynamics in synchronous parallel rounds, that is a somewhat "wild" process where an *unbounded* number of particles may change state at every round. The symmetry-breaking phase terminates when the U-Process reaches some configuration having a bias $s = \Omega(\sqrt{n \log n})$. Then we prove that, starting from *any* configuration having that bias, the process reaches consensus within $\mathcal{O}(\log n)$ rounds, with high probability. Even though our analysis of this "majority" part of the process is based on standard concentration arguments, it must cope with some *non-monotone* behaviour of the key random variables (such as the bias and the number of undecided nodes at the next round). Our refined analysis shows that, during this majority phase, the winning color never changes and, thus, the U-Dynamics also ensures Plurality Consensus in logarithmic time whenever the initial bias is $s = \Omega(\sqrt{n \log n})$. Interestingly enough, we also show that configurations with $s = \mathcal{O}(\sqrt{n})$ exist so that the system may converge toward the minority color with non-negligible probability.

Other related work. The interest in the U-Dynamics arises in fields beyond the borders of Computer Science and it seems to have a key-role in important biological processes modelled as so-called chemical reaction networks [7, 12]. For such reasons, the convergence time of this dynamics has been analyzed on different communication models [1, 3, 4, 6, 9, 11, 13, 15, 17]. Concerning the sequential model, [15] recently analyzes the U-Dynamics in arbitrary graphs when the intitial configuration is sampled uniformly at random between the two colors. In this (average-case) setting, they prove that the system converges to the initial majority color with higher probability than the initial minority one. They also give results for special classes of graphs where the minority can win with large probability if the initial configuration is chosen in a suitable way. In [4, 6, 13, 17], the same dynamics for the binary case has been analyzed in further sequential communication models.

References

1 M. A. Abdullah et al. Majority consensus on random graphs of a given degree sequence. *arXiv:1209.5025*, 2012.
2 Mohammed Amin Abdullah et al. Global majority consensus by local majority polling on graphs of a given degree sequence. *Discrete Applied Mathematics*, 180, 2015.
3 Dana Angluin et al. A Simple Population Protocol for Fast Robust Approximate Majority. *Distributed Computing*, 21(2), 2008.
4 A. Babaee et al. Distributed multivalued consensus. In *Computer and Information Sciences III*. Springer, 2013.
5 Luca Becchetti et al. Plurality consensus in the gossip model. In *ACM-SIAM SODA'15*, 2015.
6 F. Bénézit et al. Interval consensus: from quantized gossip to voting. In *IEEE ICASSP'09*, 2009.
7 L. Cardelli et al. The cell cycle switch computes approximate majority. *Scientific Reports*, Vol. 2, 2012.
8 K. Censor-Hillel et al. Global computation in a poorly connected world: Fast rumor spreading with no dependence on conductance. In *ACM STOC'12*, 2012.
9 C. Cooper et al. The power of two choices in distributed voting. In *ICALP'14*, 2014.
10 A. Demers et al. Epidemic algorithms for replicated database maintenance. In *ACM PODC'87*, 1987.
11 Benjamin Doerr et al. Stabilizing consensus with the power of two choices. In *ACM SPAA'11*, 2011.
12 David Doty. Timing in chemical reaction networks. In *ACM-SIAM SODA'14*, 2014.

13 Moez Draief et al. Convergence speed of binary interval consensus. *SIAM Journal on Control and Optimisation*, 50(3), 2012.

14 D. Kempe et al. Gossip-based computation of aggregate information. In *IEEE FOCS'03*, 2003.

15 George B. Mertzios et al. Determining majority in networks with local interactions and very small local memory. In *ICALP'14*, 2014.

16 Elchanan Mossel et al. Majority dynamics and aggregation of information in social networks. *Autonomous Agents and Multi-Agent Systems*, 28(3), 2014. doi:10.1007/s10458-013-9230-4.

17 Etienne Perron et al. Using Three States for Binary Consensus on Complete Graphs. In *INFOCOM'09*, 2009.

Brief Announcement: Shape Formation by Programmable Particles*

Giuseppe A. Di Luna[1], Paola Flocchini[2], Nicola Santoro[3], Giovanni Viglietta[4], and Yukiko Yamauchi[5]

1 **University of Ottawa, Ottawa, Canada**
gdiluna@uottawa.ca
2 **University of Ottawa, Ottawa, Canada**
paola.flocchini@uottawa.ca
3 **Carleton University, Ottawa, Canada**
santoro@scs.carleton.ca
4 **University of Ottawa, Ottawa, Canada**
gvigliet@uottawa.ca
5 **Kyushu University, Fukuoka, Japan**
yamauchi@inf.kyushu-u.ac.jp

Abstract

Shape formation is a basic distributed problem for systems of computational mobile entities. Intensively studied for systems of autonomous mobile robots, it has recently been investigated in the realm of programmable matter. Namely, it has been studied in the geometric Amoebot model, where the anonymous entities, called particles, operate on a hexagonal tessellation of the plane, have constant memory, can only communicate with neighboring particles, and can only move from a grid node to an empty neighboring node; their activation is controlled by an adversarial scheduler. Recent investigations have shown how, starting from a well-structured configuration in which the particles form a (not necessarily complete) triangle, the particles can form a large class of shapes. This result has been established under several assumptions: agreement on the clockwise direction (i.e., chirality), a sequential activation schedule, and randomization.

In this paper we provide a characterization of which shapes can be formed deterministically starting from any simply connected initial configuration of n particles. As a byproduct, if randomization is allowed, then any input shape can be formed from any initial (simply connected) shape by our algorithm, provided that n is large enough. Our algorithm works without chirality, proving that chirality is computationally irrelevant for shape formation. Furthermore, it works under a strong adversarial scheduler, not necessarily sequential. We also consider the complexity of shape formation both in terms of the number of rounds and the total number of moves performed by the particles executing a universal shape formation algorithm. We prove that our solution has a complexity of $O(n^2)$ rounds and moves: this number of moves is also asymptotically optimal.

1998 ACM Subject Classification F.1.1 Models of Computation, F.2.2 Nonnumerical Algorithms and Problems, I.2.11 Distributed Artificial Intelligence, I.2.9 Robotics

Keywords and phrases Shape formation, pattern formation, programmable matter, Amoebots, leader election, distributed algorithms, self-assembly

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.48

* Full version available at https://arxiv.org/abs/1705.03538.

© Giuseppe A. Di Luna, Paola Flocchini, Nicola Santoro, Giovanni Viglietta, and Yukiko Yamauchi; licensed under Creative Commons License CC-BY

1 Background

The term *programmable matter*, introduced by Toffoli and Margolus [4], is used to denote matter that has the ability to change its physical properties in a programmable fashion, based upon user input or autonomous sensing. Often programmable matter is envisioned as a very large number of very small locally interacting computational particles, programmed to collectively perform a complex task. Such particles could have applications in a variety of important situations: smart materials, minimally invasive surgery, etc.

Of particular interest, from the distributed computing viewpoint, is the *geometric Amoebot* model. In this model, introduced in [3] and so called because inspired by the behavior of amoeba, programmable matter is viewed as a swarm of decentralized autonomous self-organizing entities, operating on a hexagonal tessellation of the plane. These entities, called *particles*, are constrained by having simple computational capabilities (they are finite-state machines), strictly local interaction and communication capabilities (only with particles located in neighboring nodes of the hexagonal grid), and limited motorial capabilities (from a grid node to an empty neighboring node); furthermore, their activation is controlled by an adversarial (but fair) synchronous scheduler. A feature of the Amoebot model is that particles can be in two modes: *contracted* and *expanded*. When contracted, a particle occupies only one node, while when expanded the particle occupies two neighboring nodes; it is indeed this ability of a particle to expand and contract that allows it to move on the grid.

The pioneering study of [1] on *shape formation* in the geometric Amoebot model showed how particles can build simple shapes, such as a hexagon or a triangle. Subsequent investigations [2] have recently shown how, starting from a well-structured configuration in which the particles form a (not necessarily complete) triangle, they can form a larger class of shapes under several assumptions, including randomization (which is used to elect a leader), chirality, and a sequential activation schedule (i.e., at each time unit the scheduler selects only one particle which will interact with its neighbors and possibly move). Notice that, without the availability of a unique leader (provided by randomization), dropping the chirality assumption becomes a problem with a non-sequential schedule.

2 Our Contributions

We continue the investigation, significantly extending the existing results. Among other things, we provide a constructive characterization of which shapes S_F can be formed *deterministically* starting from an unknown *simply connected* initial configuration S_0 of n particles (i.e., a connected configuration without "holes").

As in [2], we assume that the size of the description of S_F is constant with respect to the size of the system, so that it can be encoded by each particle in a part of its internal memory. Such a description is available to all the particles at the beginning of the execution as their "input". The particles will form a final configuration that is an appropriate scaling, translation, rotation, and perhaps reflection of the input shape S_F. Since all particles of S_0 must be used to construct S_F, they may have to scale up S_F in order to fit: we stress that an appropriate scale factor is unknown to particles, and they must determine it autonomously. (We assume that the input shape S_F that is actually given to the particles is the smallest possible among the scaled copies of itself that can be embedded in the hexagonal grid.)

Given two shapes S_0 and S_F, we say that the pair (S_0, S_F) is *feasible* if there exists a deterministic algorithm that, in every execution (thus, regardless of the activation schedule), allows the particles to form a scaled copy of S_F starting from S_0, and no longer move. Our characrerization of feasibility is based on symmetries that are unbreakable: a shape is said to be *unbreakably k-symmetric*, for some integer $k > 1$, if it has a center of k-fold rotational symmetry that does not coincide with any vertex of the hexagonal grid.

▶ **Theorem 1.** *If (S_0, S_F) is a feasible pair and S_0 is unbreakably k-symmetric, then S_F is also unbreakably k-symmetric.*

Interestingly, all the pairs not excluded by the above theorem turn out to be feasible (provided that the size of S_0 is large enough with respect to the size of S_F), and for them we give a *universal shape formation algorithm*: this algorithm does not need any information on S_0, except that it is simply connected. The algorithm first elects 1, 2, or 3 leaders among the particles. Electing a unique leader may be impossible due to the symmetry of S_0: if $k > 1$ leaders are elected, it means that S_0 is necessarily unbreakably k-symmetric. Each leader takes an equal portion of S_0 and rearranges it into a straight line. Then, all leaders reconfigure their respective lines to form a portion of S_F, scaled up by an appropriate factor. The optimal factor is computed by each leader by simulating a Turing machine on its line of particles: the leader acts as the head, and uses the particles as memory cells on a tape.

▶ **Theorem 2.** *Let P be a system of n particles forming a simply connected shape S_0. Let S_F be a shape of constant size m that is unbreakably k-symmetric if S_0 is unbreakably k-symmetric. If all particles of P execute the universal shape formation algorithm with input a representation of the final shape S_F, and if n is at least $\Theta(m^2)$, then eventually P forms a scaled copy of S_F, and the particles cease to move.*

The total number of movements performed by the system executing our algorithm is $O(n^2)$, which is asymptotically optimal: indeed, if S_0 is a full hexagon and S_F is a line segment, $\Omega(n^2)$ moves are needed. The number of *rounds* (i.e., periods of time in which each particle is activated at least once) that an execution of our algorithm takes is also $O(n^2)$.

Our algorithm works under a stronger adversarial scheduler than [2], as it activates an arbitrary number of particles at each execution step (i.e., not necessarily just one, like the sequential scheduler). We also need a slightly less demanding communication system. Moreover, in our algorithm, no chirality is assumed: indeed, unlike in [2], different particles may have a different notion of clockwise direction. Because of this difficutly, part of the algorithm is dedicated to a "handedness agreement" procedure. We stress that our results prove that *chirality is computationally irrelevant* for shape formation.

These results concern *deterministic* shape formation. If randomization were allowed, we could always elect a unique leader with arbitrarily high probability, and apply our algorithm to *any* pair of shapes (S_0, S_F) where S_0 is simply connected, regardless of their symmetry. This extends the result of [2], which assumes the initial configuration to be a (possibly incomplete) triangle. Additionally, our notion of shape generalizes the one used in [2], where a shape is only a collection of full triangles, while we include also 1-dimensional segments as its constituting elements. Our technique actually allows us to generalize the concept of shape much further, to include essentially anything that is Turing-computable.

References

1 Z. Derakhshandeh, R. Gmyr, A.W. Richa, C. Scheideler, and T. Strothmann. An algorithmic framework for shape formation problems in self-organizing particle systems. In *Proc. of NanoCom*, pages 21:1–21:2, 2015.

2 Z. Derakhshandeh, R. Gmyr, A.W. Richa, C. Scheideler, and T. Strothmann. Universal shape formation for programmable matter. In *Proc. of SPAA*, pages 289–299, 2016.

3 Z. Derakhshandeh, R. Gmyr, T. Strothmann, R.A. Bazzi, A.W. Richa, and C. Scheideler. Leader election and shape formation with self-organizing programmable matter. In *Proc. of DNA*, pages 117–132, 2015.

4 T. Toffoli and N. Margolus. Programmable matter: concepts and realization. *Physica D*, 47(1):263–272, 1991.

Brief Announcement: Fast Aggregation in Population Protocols

Ryota Eguchi[1] and Taisuke Izumi[2]

1 **Graduate School of Engineering, Nagoya Institute of Technology, Nagoya, Japan**
28414018@stn.nitech.ac.jp

2 **Graduate School of Engineering, Nagoya Institute of Technology, Nagoya, Japan**
t-izumi@nitech.ac.jp

Abstract

The coalescence protocol plays an important role in the population protocol model. The conceptual structure of the protocol is for two agents holding two non-zero values a, b respectively to take a transition $(a, b) \to (a + b, 0)$, where $+$ is an arbitrary commutative binary operation. Obviously, it eventually aggregates the sum of all initial values. In this paper, we present a fast coalescence protocol that converges in $O(\sqrt{n} \log^2 n)$ parallel time with high probability in the model with an initial leader (equivalently, the model with a base station), which achieves an substantial speed-up compared with the naive implementation taking $\Omega(n)$ time.

1998 ACM Subject Classification C.2.4 Distributed Systems, I.1.2 protocols

Keywords and phrases population protocol, aggregation

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.49

1 Introduction

A *passively-mobile* system, which is an abstract notion of wireless ad-hoc networks, consists of a collection of moving objects (called *agents*) in a certain region, with computing devices that do not control over how they move. Despite the restrictions on communication range, memory, and computational power caused by the mobility requirement, the devices must execute cooperatively some task through tiny local computation and short-range communication with other devices located nearby. Typical examples of passively-mobile systems are the network of smart devices attached to cars or animals. *Population protocol* is one of the promising models for such a system, which is first introduced by Angluin et al. [2]. A Population protocol consists of anonymous and identical n agents, which are defined as deterministic state machines. The communication among agents is performed by pairwise interactions, where two interacting agents change their states following a transition function (protocol) deployed to all agents. An execution of a population protocol is a sequence of pairwise interactions. In the basic model, the scheduling of interactions is worst-case but guaranteed to be *fair*, which means that if in the infinitely-many interactions every two agents interact infinitely often.

Recent trends of this model are to design fast protocols for popular problems (e.g. leader election, majority) converging in $O(polylog(n))$ time, and to reveal trade-offs between time and space for several problems. To measure time in the runs of the population protocol models, the (uniform) probabilistic scheduler is often assumed. In the model, two agents interacting at each step are selected at random uniformly and independently. In the literature


© Ryota Eguchi and Taisuke Izumi;
licensed under Creative Commons License CC-BY

of this model, time complexity is defined as the number of interactions divided by the number of agents n, and space complexity is the number of states used by each agent.

We consider some abstract problem, called the *aggregation* problem. Precisely, the aggregation problem is defined by any monoid $(X, +)$, where initially each agent i has a value $x_i \in X$, and eventually one specific agent must output the value of $s = \sum_i x_i$. This problem can be solved by the traditional *coalescence* protocol, whose transition rule for two agents with values a and b respectively is specified by $(a, b) \to (a + b, 0)$. One can see that, the standard coalescence protocol needs $\Theta(n)$ time for convergence, since the probability that the last two agents having non-zero values interact is $1/n^2$.

In this paper, we present a new coalescence protocol. It achieves $O(\sqrt{n} \log^2 n)$-convergence time in the special model with existence of one unique leader (equivalently, the model with a base station). On the space complexity side, agents (including the leader) uses $O(|X|^3)$ states.

Problem Statement

Let $(X, +)$ be an arbitrary commutative monoid whose identity element is zero (where $+$ is not necessarily the standard arithmetic sum), and $\hat{X} = X \setminus \{0\}$. In the aggregation problem for $(X, +)$, each agent i initially has a value $x_i \in X$, and the goal of the task is that the leader computes the value $s = \sum_i x_i$. More precisely, we assume that the leader equips an output register storing a value in X. The value of the output register must be converged and stabilized into s. Note that the leader does not have to detect the termination of an execution, and is allowed to update answers multiple times. The computation time of the aggregation problem is defined as the time taken until the convergence of the output register.

Outline of Our Algorithm

Our algorithm utilizes several algorithmic tools proposed in past literature as building blocks. Before the presentation of our protocol, we illustrate three tools. The first algorithm called *epidemics* (or propagation) is a straightforward subroutine used in many algorithms. The abstract structure of the epidemics is as follows: At first there are at least one agent with value v, which wishes to propagate v to all other agents, and the other agents initially with value $\perp$. The transition rule is $(v, \perp)$ or $(\perp, v)$ to (v, v). The analysis by Angluin et al. [3] shows that under the random scheduler the epidemics algorithm finishes within $O(\log n)$ parallel time with high probability.

The second tool is a synchronization mechanism called *phase clock* which counts approximately $O(\log n)$ time or $O(\log^2 n)$ time. The phase clock is first presented in the paper by Angluin et al. [3]. The phase clock is mainly introduced for a unique leader to detect the end of the epidemics (i.e. $O(\log n)$ time), and by a simple extension, it is also possible to count $O(\log^2 n)$ time [4]. A non-trivial advantage of the phase clock mechanism is that it uses only $O(1)$ states per agent.

The third tool is synthetic coin flips due to Alistarh et al. [1], which provides the accessibility of private random bits to each agent. It gives a coin flipping mechanism with reasonably small bias to the agents. The randomness of the synthetic coin flips is extracted from the random interaction-pattern of the scheduler, and thus it works only on the random scheduler.

The idea of our algorithm is very simple: The bottleneck of the standard coalescence algorithm is the situation where the number of agents with non-zero values becomes small. If only m agents have non-zero values, an interaction selected by the scheduler gets no progress

of the algorithm with probability $1-\Theta((m/n)^2)$. In the naive coalescence algorithm, spending $O(\sqrt{n}\mathrm{polylog}(n))$ parallel time, $\Theta(\sqrt{n})$ agents still have non-zero values. To accelerate the following coalescence process, when only $O(\sqrt{n})$ agents have non-zero values, we utilizes another mechanism, called *sequential absorption*. The sequential absorption first chooses an agent (which is not leader) with a non-zero value as an absorption agent only spending $O(\log n)$ parallel time. This process is achieved by utilizing the phase clock and the synthetic coin flips: At each phase, the agents with non-zero values flip the synthetic coin, the agents which get value 1 start epidemics, and the epidemics kill the agents with value 0. The number of phases to elect one unique absorption agent is $O(\log n)$, thus the total time to elect the agent is $O(\log^2 n)$. The absorption agent runs the epidemics its value, and immediately become an agent with value zero. The value reaches to the leader within $O(\log n)$ time. Repeating this procedure $\Theta(\sqrt{n})$ times, we can complete the aggregation. Since both the election and epidemics take $O(\log^2 n)$ time, the total running time of the sequential absorption is $O(\sqrt{n}\log^2 n)$. The remaining issue is to combine those two algorithms. While the sequential composition is obviously correct, it requires the timer for (exactly or approximately) counting $\Theta(\sqrt{n})$ parallel time. To avoid consuming extra memory space, instead we choose fair composition, that is, simply running them concurrently. This composition does not affect the correctness of our protocol, since the absorption agent behave following way so that the value of the sum does not change: When the absorption agent with value x_i detect its uniqueness by the phase clock counting $O(\log^2 n)$ time, it immediately change its state to the value zero, and thus x_i is never aggregated in the standard coalescence side. Here we present our main theorem. Note that our algorithm have a low probability of error, that is conversely the algorithm convergences only with high probability.

▶ **Theorem 1.** *Our algorithm solves an aggregation problem for $(X,+)$ in expected $O(\sqrt{n}\log^2 n)$ time using $O(|X^3|)$ states per agent, with high probability.*

Discussion and Research Direction

In [2], authors show the simple coalescence protocol can compute semilinear predicates, which are exact characterization of the basic population protocol model, and thus our protocol computes the predicates in $O(\sqrt{n}\log^2 n)$ time. However for computation of semilinear predicates with leader, there is much faster protocol presented by Angluin et al. [3] which converges in $O(\log^4 n)$ time with high probability. We believe that due to its simplicity and generality, there are some applications of our algorithm in population protocol models.

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*, pages 2560–2579. SIAM, 2017. doi:10.1137/1.9781611974782.169.

2 Dana Angluin, James Aspnes, Zoë Diamadi, MichaelJ. Fischer, and Rene Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, 18(4):235–253, 2006. doi:10.1007/s00446-005-0138-3.

3 Dana Angluin, James Aspnes, and David Eisenstat. Fast computation by population protocols with a leader. *Distributed Computing*, 21(3):183–199, September 2008. doi:10.1007/s00446-008-0067-z.

4 Leszek Gasieniec and Grzegorz Stachowiak. Fast space optimal leader election in population protocols. *arXiv preprint arXiv:1704.07649*, 2017.

Brief Announcement: A Persistent Lock-Free Queue for Non-Volatile Memory*

Michal Friedman[1], Maurice Herlihy[2], Virendra Marathe[3], and Erez Petrank[4]

1 **Technion, Haifa, Israel**
michal.f@cs.technion.ac.il
2 **Brown University, Providence, RI, USA**
mph@cs.brown.edu
3 **Oracle Labs, Redwood Shores, CA, USA**
virendra.marathe@oracle.com
4 **Technion, Haifa, Israel**
erez@cs.technion.ac.il

Abstract

Non-volatile memory is expected to coexist with (or even displace) volatile DRAM for main memory in upcoming architectures. As a result, there is increasing interest in the problem of designing and specifying *durable* data structures that can recover from system crashes. Data-structures may be designed to satisfy stricter or weaker durability guarantees to provide a balance between the strength of the provided guarantees and performance overhead. This paper proposes three novel implementations of a concurrent lock-free queue. These implementations illustrate the algorithmic challenges in building persistent lock-free data structures with different levels of durability guarantees. We believe that by presenting these challenges, along with the proposed algorithmic designs, and the possible levels of durability guarantees, we can shed light on avenues for building a wide variety of durable data structures. We implemented the various designs and evaluate their performance overhead compared to a simple queue design for standard (volatile) memory.

1998 ACM Subject Classification E.1 Data Structures, D.4.2 Storage Management, D.1.3 Concurrent Programming

Keywords and phrases Non-volatile Memory, Concurrent Data Structures, Non-blocking, Lock-free

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.50

1 Introduction

Memory is said to be *non-volatile* if it does not lose its contents after a system crash. Non-volatile memory is soon expected to co-exist with or even displace volatile DRAM for main memory (but not caches or registers) in many architectures. As a result, there is increasing interest in the problem of designing and specifying *durable* data structures, that is, data structures whose state can be recovered after a system crash.

A major challenge in designing durable data structures is that caches and registers are expected to remain volatile. Thus, the state of main memory following a crash may be

* This work was supported by the United States - Israel Binational Science Foundation (BSF) grant No. 2012171. Maurice Herlihy was supported by NSF grant 1331141.

© Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank;
licensed under Creative Commons License CC-BY

inconsistent, missing all previous writes to the data structure that were present in the cache, but not yet written into the main memory. Dealing with arbitrary missing words after a crash requires non-trivial data structure algorithms that make sure key data does get written to main memory (without incurring too much overhead), so that restoration of the data structure to a consistent state becomes possible.

It would be interesting to know if it is possible to build libraries of high performance persistent data structures that are heavily optimized using ad-hoc techniques informed by the data structure architecture and semantics. Previous work focuses on B-tree implementations. The interest in B-trees is natural given their prevalence in file system and database implementations. However, other foundational data structures are also used in application domains that care about persistence; e.g. hash tables in key-value stores, persistent message queues, etc. Since traditional storage media have been block-based, all these applications persist these data structures by marshaling them to a block-based format. Doing so involves non-trivial overhead that was dwarfed by the high cost of disk access. As a result, the in-memory representation and on-disk (-SSD) representation of these data structures are quite different. Byte-addressable persistent memory can be used to create a unified persistent representation. As far as we know, there is no previous work that attempts to optimize these data structures for persistent memory. Furthermore, none of the above works attempt to build highly concurrent, *nonblocking* persistent data structures.

In order to strive for high-performance crash-resilient software on non-volatile memories, we propose to look at modern highly-concurrent data structures, such as the ones used in java.util.concurrent, and enhance them to work with non-volatile memories. Designing such concurrent data structures for upcoming non-volatile memories requires dealing with the challenge of high concurrency and non-volatile durability combined.

We study these challenges by designing a durable version of the lock-free concurrent queue data structure of Michael and Scott [2], which also serves as the base algorithm for the queue in java.util.concurrent. This concurrent data structure is complicated enough to demonstrate the challenges that concurrent durable data structures raise, and simple enough to demonstrate solutions for these challenges.

Recently various definitions were proposed to formalize durability. In this paper we adopt and work with the definition of linearizable durability by Izraelevitz *et al.* [1]. Informally, durable linearizability guarantees that the state of a data structure following a crash reflects a consistent subhistory of the operations that actually occurred. This subhistory includes all operations that completed before the crash, and may or may not include operations in progress when the crash occurred. The main tool for achieving durable linearizability for a concurrent data structure is the use of explicit instructions that force volatile cashed data to be written to non-volatile memory. While such *persistence barrier* instructions enforce correctness, they also carry a performance cost and their use should be minimized.

An alternative, weaker condition, is *buffered durable linearizability*. Informally, this condition guarantees that the state of the object following a crash reflects a consistent subhistory of the operations that actually occurred, but this subhistory *need not* include all operations that completed before the crash.

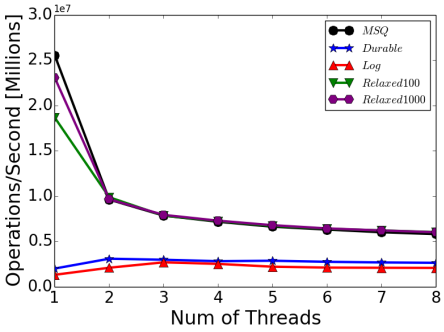
The first main contribution of this paper is the proposal of three novel designs of durable concurrent queues, extending the original Michael-Scott queue for use with non-volatile memory. It is easy to obtain a durable linearizable queue by adding many persistence barrier operations automatically. But, in general, the obtained performance can be very low. In this paper, we attempt to minimize the overhead and still achieve robustness to crashes. The first implementation, denoted *durable* queue, provides durable linearization. The second

implementation, denoted *log* queue, provides durable linearization, as well as an additional property that we discuss next. The third implementation, denoted *relaxed* queue, provides buffered durable linearizability.

When crashes occur during an execution, it is often difficult to tell which operations were executed and which operations failed to execute when a crash occurs. Durable linearizability does not provide a mechanism to determine whether an operation that executed concurrently with a crash was eventually executed. Without the ability to distinguish completed operations from lost operations, it would be difficult to recover the entire program, because in practice it is often important to execute each operation exactly once. In this paper we enable a more robust use of the queue, by defining a new (natural) notion of *detectable execution*. A data structure provides detectable execution if it is possible to tell at the end of a recovery phase whether a specific operation was executed. The *log* queue provides durable linearization and detectable execution. If the program that uses the queue follows a similar procedure for detecting execution, then it is possible to tell how much of the execution has completed on recovery from a crash, and program recovery at higher level becomes possible.

2 Measurements

We have implemented the three queue designs and evaluated their performance by comparing them one against the other and also against the original MS queue. We ran measurements on an 8-cores **Intel** Xeon D-1540 2.6GHz.



Above, we depict the throughput of five queues: *MSQ* is the Michel and Scott's queue, *Durable* is the durable queue, *Log* is the queue that can detect which operations completed before the crash, and *Relaxed* is the queue that only guarantees a view of a prefix of the operations executed before the crash. We ran *Relaxed* with an additional operation *sync* that makes all history durable once every 100 or 1000 operations, denoted *Relaxed100* and *Relaxed1000*. As expected, implementations that provide durable linearization have a noticeable cost, and interestingly, providing detectable execution does not add a significant overhead and may be worthwhile in this case. In addition, implementations that provides only buffered durable linearizability obtain good performance when the `sync()` method is invoked infrequently.

References

1 Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *Distributed Computing - 30th International Symposium, DISC*, pages 313–327, 2016.

2 Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 267–275, 1996.

Brief Announcement: Lower Bounds for Asymptotic Consensus in Dynamic Networks*†

Matthias Függer[1], Thomas Nowak[2], and Manfred Schwarz[3]

1 **CNRS, LSV, ENS Paris-Saclay, Inria, Paris, France**
mfuegger@lsv.fr
2 **Université Paris-Sud, Paris, France**
thomas.nowak@lri.fr
3 **ECS, TU Wien, Vienna, Austria**
mschwarz@ecs.tuwien.ac.at

Abstract

In this work we study the performance of asymptotic and approximate consensus algorithms in dynamic networks. The asymptotic consensus problem requires a set of agents to repeatedly set their outputs such that the outputs converge to a common value within the convex hull of initial values. This problem, and the related approximate consensus problem, are fundamental building blocks in distributed systems where exact consensus among agents is not required, e.g., man-made distributed control systems, and have applications in the analysis of natural distributed systems, such as flocking and opinion dynamics. We prove new nontrivial lower bounds on the contraction rates of asymptotic consensus algorithms, from which we deduce lower bounds on the time complexity of approximate consensus algorithms. In particular, the obtained bounds show optimality of asymptotic and approximate consensus algorithms presented in [Charron-Bost et al., ICALP'16] for certain classes of networks that include classical failure assumptions, and confine the search for optimal bounds in the general case.

Central to our lower bound proofs is an extended notion of valency, the set of reachable limits of an asymptotic consensus algorithm starting from a given configuration. We further relate topological properties of valencies to the solvability of exact consensus, shedding some light on the relation of these three fundamental problems in dynamic networks.

1998 ACM Subject Classification F.1.2 Modes of Computation

Keywords and phrases Asymptotic Consensus, Dynamic Networks, Contraction Rate, Time Commplexity, Lower Bounds

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.51

1 Introduction

In the *asymptotic consensus* problem a set of agents, each starting from an initial value in $\mathbb{R}^d$, update their values such that all agents' values converge to a common value within the convex hull of initial values. The problem is closely related to the *approximate consensus* problem, in which agents have to irrevocably decide on values that lie within a predefined distance $\varepsilon > 0$ of each other. The latter is weaker than the *exact consensus* problem in which agents need to decide on the same value. Both the asymptotic and the approximate

* The research was partially funded by the Austrian Science Fund (FWF) project SIC (P26436) and ADynNet (P28182).
† The full version is available at https://arxiv.org/abs/1705.02898


© Matthias Függer, Thomas Nowak, and Manfred Schwarz;
licensed under Creative Commons License CC-BY
31st International Symposium on Distributed Computing (DISC 2017).
Editor: Andréa W. Richa; Article No. 51; pp. 51:1–51:3

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

consensus problems have not only a variety of applications in the design of man-made control systems like sensor fusion [1], clock synchronization [8], formation control [6], rendezvous in space [9], or load balancing [5], but also for analyzing natural systems like flocking [11], firefly synchronization [10], or opinion dynamics [7]. These problems often have to be solved under harsh-environmental restrictions: with limited computational power and local storage, under restricted communication abilities, and in presence of communication uncertainty.

In this work we study asymptotic consensus in round-based computational models with a dynamic communication topology whose directed communication graphs are chosen each round from a predefined set of communication graphs, the so-called *network model*. In previous work [2], Charron-Bost et al. showed that asymptotic consensus is solvable precisely within *rooted network models* in which all communication graphs contain rooted spanning trees. These rooted spanning trees need not have any edges in common and can change from one round to the next.

An interesting special case of rooted network models are network models whose graphs are *non-split*, that is, any two agents have a common incoming neighbor. Their prominent role is motivated by two properties: (i) They occur as communication graphs in benign classical distributed failure models. For example, in synchronous systems with crashes, in asynchronous systems with a minority of crashes, and synchronous systems with send omissions. (ii) In [2], Charron-Bost et al. showed that non-split graphs also play a central role in arbitrary rooted network models: they showed that any product of $n-1$ rooted graphs with n nodes is non-split, allowing to transform asymptotic consensus algorithms for non-split network models into their *amortized* variants for rooted models.

Contribution

In this work, we prove lower bounds on the contraction rate of any asymptotic consensus algorithm. All lower bounds hold regardless of the structure of the algorithm. In particular, algorithms can be full-information and agents can set their outputs outside the convex hull of received values. This, e.g., includes using higher-order filters in contrast to the 0-order filters of averaging algorithms.

The proof strategy is as follows: The central idea is the concept of the *valency of a configuration* of an asymptotic consensus algorithm, defined as the set of limits reachable from this configuration. By studying the changes in valency along executions, we infer bounds on the contraction rate. Notably, the lower bounds are valid for arbitrary dimensions.

Note that if exact consensus is solvable in network model $\mathcal{N}$, an optimal contraction rate of 0 can be achieved. Otherwise, we show the following non trivial bounds:

- We show a tight lower bound of $1/3$ in non-split network models with $n = 2$ agents.
- We prove that the contraction rate is lower bounded by $1/2$ in a system with $n \geq 3$ agents and $\mathrm{deaf}(G) \subseteq \mathcal{N}$ where, for an arbitrary communication graph G, $\mathrm{deaf}(G) = \{F_1, \dots, F_n\}$ and F_i is derived from G by making agent i *deaf* in F_i, i.e., removing the incoming links of i in G. Additionally we show tightness for $d \in \{1, 2\}$ dimensional values.
- The study of the valencies' topological structure with respect to the network model where the asymptotic consensus algorithm is executed in, reveals that any asymptotic consensus algorithm must have a contraction rate of at least $1/(D+1)$, where D is the so-called α-diameter of $\mathcal{N}$. This generalizes the previous two lower bounds.

Tightness for $1/3$ and $1/2$ results from the combination with algorithms presented in [3] and [4]. Together with the algorithm for arbitrary dimensions d with contraction rate $\frac{d}{d+1}$ in non-split models [4] the bounds in Table 1 follow. Furthermore we extend our results on

Table 1 Summary of lower and upper bounds on contraction rates if consensus is not solvable. New lower bounds proved in this work are marked with a *. The three right columns distinguish between the case the network model is (i) non-split and contains deaf(G) for some communication graph G, (ii) is non-split, and (iii) is rooted.

agents	dimension	network model in which exact consensus is unsolvable		
		non-split with deaf graphs	$\subseteq$ non-split $\subseteq$	rooted
$n = 2$	$d \geq 1$	$\frac{1}{3}^*$	$\frac{1}{3}^*$	$\frac{1}{3}^*$
$n \geq 3$	$d \in \{1, 2\}$	$\frac{1}{2}^*$	$\left[\frac{1}{D+1}^*, \frac{1}{2}\right]$	$\left[\frac{1}{D+1}^*, \sqrt[n-1]{\frac{1}{2}}\right]$
	$d \geq 3$	$\left[\frac{1}{2}^*, \frac{d}{d+1}\right]$	$\left[\frac{1}{D+1}^*, \frac{d}{d+1}\right]$	$\left[\frac{1}{D+1}^*, \sqrt[n-1]{\frac{d}{d+1}}\right]$

contraction rates to derive new lower bounds on the decision time of approximate consensus algorithms.

Acknowledgments

We would like to thank Bernadette Charron-Bost for the many fruitful discussions and her valuable input which greatly helped improve the paper.

References

1 J. A. Benediktsson and P. H. Swain. Consensus theoretic classification methods. *IEEE Transactions on Systems, Man, and Cybernetics*, 22(4):688–704, 1992.

2 Bernadette Charron-Bost, Matthias Függer, and Thomas Nowak. Approximate consensus in highly dynamic networks: The role of averaging algorithms. In *Proceedings of ICALP*, pages 528–539, 2015.

3 Bernadette Charron-Bost, Matthias Függer, and Thomas Nowak. Fast, robust, quantizable approximate consensus. In *Proceedings of the 43rd International Colloquium on Automata, Languages, and Programming, ICALP16*, pages 137:1–137:14, 2016.

4 Bernadette Charron-Bost, Matthias Függer, and Thomas Nowak. Multidimensional asymptotic consensus in dynamic networks. *CoRR*, abs/1611.02496, 2016.

5 George Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7(2):279–301, 1989.

6 Magnus Egerstedt and Xiaoming Hu. Formation constrained multi-agent control. *IEEE Transactions on Robotics and Automation*, 17(6):947–951, 2001.

7 R. Hegselmann and U. Krause. Opinion dynamics and bounded confidence models, analysis, and simulation. *Journal of artificial societies and social simulation*, 5(3):1–33, 2002.

8 Qun Li and Daniela Rus. Global clock synchronization in sensor networks. *IEEE Transactions on Computers*, 55(2):214–226, 2006.

9 J. Lin, A.S. Morse, and B.D.O. Anderson. The multi-agent rendezvous problem. In Vijay Kumar, Naomi Leonard, and A. Stephen Morse, editors, *Cooperative Control: A Post-Workshop Volume 2003 Block Island Workshop on Cooperative Control*, pages 257–289. Springer, Heidelberg, 2005.

10 Renato E. Mirollo and Steven H. Strogatz. Synchronization of pulse-coupled biological oscillators. *SIAM Journal on Applied Mathematics*, 50(6):1645–1662, December 1990.

11 Tamás Vicsek, András Czirók, Eshel Ben-Jacob, Inon Cohen, and Ofer Shochet. Novel type of phase transition in a system of self-driven particles. *Physical Review Letters*, 75(6):1226–1229, 1995.

Brief Announcement: Applying Predicate Detection to the Stable Marriage Problem*†

Vijay K. Garg

The University of Texas at Austin, TX, USA
garg@ece.utexas.edu

Abstract

We show that many techniques developed in the context of predicate detection are applicable to the stable marriage problem. The standard Gale-Shapley algorithm can be derived as a special case of detecting linear predicates. We also show that techniques in computation slicing can be used to represent the set of all constrained stable matchings.

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases Stable Matching, Linear Predicates, Distributive Lattices

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.52

1 Introduction

The Stable Matching Problem (SMP) [2] has wide applications in economics, distributed computing, and resource allocation.

In this paper, we show that techniques for detecting global predicates can be used to derive solutions to a more general problem than SMP, called *constrained SMP*. In our formulation, in addition to men and women preferences, there may be a set of constraints on the set of marriages consistent with men's preferences. For example, we may state that Peter's regret [4] should be less than that of Paul where the *regret* of a man in a matching is the choice number he is assigned.

To solve a constrained SMP, we define a distributed computation such that every assignment of women to men corresponds to a global state of the distributed computation. The set of global states form a finite distributive lattice under the natural order on the global states. The problem of finding a stable matching reduces to that of finding a consistent global state that satisfies the boolean predicate B for the constrained stable matching. We show that B satisfies the linearity property introduced in predicate detection [1]. Consequently, we can use the algorithm that can detect a linear predicate in [1] to find a constrained stable matching. The stable matching found by this algorithm is man-optimal [2]. The Gale-Shapley algorithm is a special case of the linear predicate detection algorithm when the set of external constraints is null. While there always exists a stable matching for the SMP problem, there may not exist a stable matching in the constrained SMP. Our algorithm is guaranteed to find one whenever it exists. In addition, the stable matching it finds is man-optimal.

We then consider the constrained SMP in a distributed setting where men and women know only those constraints that either they choose (such as their preference lists) or the external constraints in which they participate. We present a distributed algorithm based on

* This work was partially supported by NSF CSR-1563544.

† A full version of the paper is available at http://users.ece.utexas.edu/~garg/TechReports/2017/TR-PDS-2017-001.pdf.

© Vijay K. Garg;
licensed under Creative Commons License CC-BY

diffusing computation that solves the constrained SMP. When external constraints are null, the algorithm reduces to a distributed version of the Gale-Shapley Algorithm [2]. To our knowledge, this paper is the first one to propose sequential and distributed algorithms for constrained SMP.

In the full version of the paper [3], we also consider the problem of computing all constrained stable matchings. Since the number of stable matchings may be exponential in the number of men, instead of keeping all matchings in an explicit form, we would like a concise representation of polynomial size that can be used to enumerate all constrained stable matchings. In SMP literature, rotation posets are used to capture all stable matchings. We use the notion of computation slicing introduced in [5] for this purpose. In particular, we give an efficient algorithm to compute the slice for the constrained SMP computation. A rotation poset [4] is a special case of the slice when the set of external constraints is empty.

2 Modeling Stable Matching Problem (SMP) as a Distributed Computation

Let E be the set of proposals made by men to women. We also call these proposals *events* which are executed by n processes corresponding to n men denoted by $\{P_1 \dots P_n\}$. Each of the events can be characterized by a tuple (i, j) that corresponds to the proposal made by man i to woman j. We impose a partial order $\rightarrow_p$ on this set of events to model the order in which these proposals can be made. In the standard SMP, every man P_i has its preference list mpref$[i]$ such that mpref$[i][k]$ gives the k^{th} most preferred woman for P_i. We model mpref using $\rightarrow_p$; if P_i prefers woman j to woman k, then there is an edge from the event (i, j) to the event (i, k). As in SMP, we assume that every man gives a total order on all women.

In the standard stable matching problem, there are no constraints on the order of proposals made by different men, and $\rightarrow_p$ can be visualized as a partial order $(E, \rightarrow_p)$ with n disjoint chains. We generalize the SMP problem to include external constraints on the set of proposals. In the constrained SMP problem, $\rightarrow_p$ can relate proposals made by different men and therefore $\rightarrow_p$ forms a general poset $(E, \rightarrow_p)$.

A global state $G \subseteq E$ is simply the subset of events executed in the computation such that it preserves the order of events within each P_i. A global state G is *consistent* if it preserves the $\rightarrow_p$ order. We will deal only with consistent global states from now on. We let $G[i]$ denote the last proposal made by P_i. Initially, $G[i]$ is null for all men. If P_i has made $k > 0$ proposals, then mpref$[i][k]$ gives the identity of the woman last proposed by P_i. We model women preferences using edges on the computation graph as follows. If an event e corresponds to a proposal by P_i to woman q and she prefers P_j, then we add a $\rightarrow_w$ edge from e to the event f that corresponds to P_j proposing to woman q. The set E along with $\rightarrow_w$ edges also forms a partial order $(E, \rightarrow_w)$ where $e \rightarrow_w f$ iff both proposals are to the same woman and that woman prefers the proposal f to e.

The above discussion motivates the following definition.

▶ **Definition 1** (Constrained SMP Graph)**.** Let $E = \{(i, j) | i \in [1..n] \text{ and } j \in [1..n]\}$. A Constrained SMP Graph $((E, \rightarrow_p), \rightarrow_w)$ is a directed graph on E with two sets of edges $\rightarrow_p$ and $\rightarrow_w$ with the following properties: (1) $(E, \rightarrow_p)$ is a poset such that the set $P_i = \{(i, j) | j \in [1..n]\}$ is a chain for all i, and (2) $(E, \rightarrow_w)$ is a poset such that the set $Q_j = \{(i, j) | i \in [1..n]\}$ is a chain for all j and there is no $\rightarrow_w$ edge between proposals to different women, i.e., for all $i, j, k, l : (i, j) \rightarrow_w (k, l) \Rightarrow (j = l)$.

We define the *frontier* of a global state G as the set of maximal events executed by any process in G. It includes only the last event executed by P_i (if any). We call the events in G that are not in $frontier(G)$ as pre-frontier events. A consistent global state G is *admissible* if $\forall e, f \in frontier(G) : \forall g \in G : (e \rightarrow_w g) \Rightarrow \neg((g \rightarrow_p f) \vee (g = f))$. We now have the following lemma.

▶ **Lemma 2.** *Let $((E, \rightarrow_p), \rightarrow_w)$ be a constrained SMP graph. A consistent global state G such that $G \cap P_i \neq \emptyset$ for all i is admissible iff the assignment by G corresponds to a constrained stable matching.*

Therefore, the problem of finding a stable matching is the same as finding a consistent global state that satisfies the predicate *admissible* which is defined purely in graph-theoretic terms on the constrained SMP graph.

We now show that admissibility satisfies linearity introduced in [1]. Any linear predicate can be detected efficiently. A key concept in deriving an efficient predicate detection algorithm is that of a *forbidden* state. Let L be the lattice of all global states of a poset $(E, \rightarrow_p)$. Given a predicate B, and a global state $G \in L$, a state $G[i]$ is called forbidden if its inclusion in any global state $H \in L$, where $G \subseteq H$, implies that B is false for H. Formally, $\text{forbidden}(G, i, B) \equiv \forall H \in L : G \subseteq H : (G[i] \neq H[i]) \vee \neg B(H)$.

A predicate B is linear with respect to the poset $(E, \rightarrow_p)$ if for any global state G in the poset, B is false in G implies that G contains a *forbidden state*. Formally, a boolean predicate B is *linear* with respect to a poset $(E, \rightarrow_p)$ iff $\forall G \in L : \neg B(G) \Rightarrow \exists i : \text{forbidden}(G, i, B)$. We now have

▶ **Lemma 3.** *For any global state G that is not a constrained stable matching, there exists an i such that $forbidden(G, i, admissible)$.*

We now discuss detection of linear global predicates. On account of linearity of B, if B is evaluated to be false in some global state G, then we can determine i such that $forbidden(G, i, B)$. We can then simply advance on any i such that $forbidden(G, i, B)$ holds.

We now consider the constrained SMP in a distributed system setting. We assume that each man and woman knows only his or her preference lists. In addition, each man is given a list of prerequisite proposals for each of the women that he can propose to. In terms of the constrained-SMP graph, this corresponds to every man knowing the incoming $\rightarrow_p$ edges for the chain that corresponds to that man in the graph. The full paper [3] presents a diffusing computation for solving the constrained SMP problem.

Acknowledgements. I thank Rohan Garg for many discussions on this topic and anonymous reviewers for their comments.

References

1 Craig M Chase and Vijay K Garg. Detection of global predicates: Techniques and their limitations. *Distributed Computing*, 11(4):191–201, 1998.

2 David Gale and Lloyd S Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962.

3 Vijay K. Garg. Applying predicate detection to the stable marriage problem. Technical Report TR-PDS-2017-001, The University of Texas at Austin, May 2017. URL: `http://users.ece.utexas.edu/~garg/TechReports/2017/TR-PDS-2017-001.pdf`.

4 Dan Gusfield and Robert W Irving. *The stable marriage problem: structure and algorithms*. MIT press, 1989.

5 Neeraj Mittal and Vijay K Garg. Computation slicing: Techniques and theory. In *International Symposium on Distributed Computing*, pages 78–92. Springer, 2001.

Brief Announcement: Towards Reduced Instruction Sets for Synchronization*

Rati Gelashvili[1], Idit Keidar[2], Alexander Spiegelman[2], and Roger Wattenhofer[4]

1 **MIT, Cambridge, MA, USA**
2 **Technion, Haifa, Israel**
3 **Technion, Haifa, Israel**
4 **ETH, Zürich, Switzerland**

Abstract

Contrary to common belief, a recent work by Ellen, Gelashvili, Shavit, and Zhu has shown that computability does not require multicore architectures to support "strong" synchronization instructions like *compare-and-swap*, as opposed to combinations of "weaker" instructions like *decrement* and *multiply*. However, this is the status quo, and in turn, most efficient concurrent data-structures heavily rely on *compare-and-swap* (e.g. for swinging pointers).

We show that this need not be the case, by designing and implementing a concurrent linearizable Log data-structure (also known as a History object), supporting two operations: *append*(*item*), which appends the item to the log, and *get-log*(), which returns the appended items so far, in order. Readers are wait-free and writers are lock-free, hence this data-structure can be used in a lock-free universal construction to implement any concurrent object with a given sequential specification. Our implementation uses atomic *read*, *xor*, *decrement*, and *fetch-and-increment* instructions supported on X86 architectures, and provides similar performance to a *compare-and-swap*-based solution on today's hardware. This raises a fundamental question about minimal set of synchronization instructions that the architectures have to support.

1998 ACM Subject Classification C.1.2 Multiple Data Stream Architectures (Multiprocessors), D.1.3 Concurrent Programming

Keywords and phrases Consensus hierarchy, universal construction, synchronization instruction

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.53

1 Introduction

In order to develop efficient concurrent algorithms and data-structures in multiprocessor systems, processes that take steps asynchronously need to coordinate their actions. In shared memory systems, this is accomplished by applying hardware-supported low-level atomic instructions to memory locations. An atomic instruction takes effect as a single indivisible step. The most natural and universally supported instructions are *read* and *write*, as these are useful even in uniprocessors to store and load data from memory.

A concurrent implementation is *wait-free*, if any process that takes infinitely many steps completes infinitely many operation invocations. An implementation is *lock-free* if in any infinite execution infinitely many operations are completed. Binary consensus is a synchronization task where processes start with input bits, and must agree on an output bit

* A full version of the paper is available at `http://arxiv.org/abs/1705.02808`.


© Rati Gelashvili, Idit Keidar, Alexander Spiegelman, and Roger Wattenhofer;
licensed under Creative Commons License CC-BY

that was an input to one of the processes. For one-shot tasks like consensus, wait-freedom and lock-freedom are equivalent. Herlihy's Consensus Hierarchy [2] assigns a *consensus number* to each object, namely, the number of processes for which there is a wait-free binary consensus algorithm using only instances of this object and *read-write* registers. An object with a higher consensus number is hence a more powerful tool for synchronization. Moreover, Herlihy showed that consensus is a fundamental synchronization task, by developing a universal construction which allows n processes to wait-free implement any object with a sequential specification, provided that they can solve consensus among themselves.

Herlihy's hierarchy provides an explanation as to why, for instance, the *compare-and-swap* instuction can be viewed "stronger" than *fetch-and-increment*, as the consensus number of a Compare-and-Swap object is n, while the consensus number of Fetch-and-Increment is 2.

However, key to this hierarchy is treating synchronization instructions as distinct objects, an approach that is far from the real-world, where multiprocessors do let processes apply supported atomic instructions to arbitrary memory locations. In fact, a recent work by Ellen et al. [1] has shown that a combination of instructions like *decrement* and *multiply-by-n*, whose corresponding objects have consensus number 1 in Herlihy's hierarchy, when applied to the same memory location, allows solving wait-free consensus for n processes. Thus, in terms of computability, a combination of instructions traditionally viewed as "weak" can be as powerful as a *compare-and-swap* instruction, for instance.

The practical question is whether we can really replace a *compare-and-swap* instruction in concurrent algorithms and data-structures with a combination of weaker instructions. *compare-and-swap* is ubiquitous in practice and used heavily for various tasks like swinging a pointer. Also, the protocol given by Ellen et al. solves only binary n-process consensus. It is not clear how to use it for implementing complex concurrent objects, as utilizing Herlihy's universal construction is not a practical solution. On the optimistic side, there exists a concurrent queue implementation based on *fetch-and-add* that outperforms *compare-and-swap*-based alternatives [3]. Both a Queue and a Fetch-and-Add object have consensus number 2, and this construction does not "circumvent" Herlihy's hierarchy by applying different non-trivial synchronization instructions to the same location. Indeed, we are not aware of any practical construction that relies on applying different instructions to the same location.

We develop a lock-free universal construction using only *read*, *xor*, *decrement*, and *fetch-and-increment* instructions. The construction could be made wait-free by standard helping techniques. In particular, we implement a Log object (also known as a History object), which supports high-level operations *get-log*() and *append*(*item*), where *get-log*() returns all previously appended items in order. This interface can be used to agree on a simulated object state, and thus, provides the universal construction [2]. In practice, we require a *get-log*() for each thread to return a suffix of items after the last *get-log*() by this thread. We design a lock-free Log with wait-free readers, which performs as well as a *compare-and-swap*-based solution on modern hardware. We could replace *fetch-and-increment* and *decrement* with the atomic *fetch-and-add* instruction, reducing the instruction set size even further.

2 Algorithm

We work in the bounded concurrency model where at most n processes will ever access the Log implementation. The object is implemented by a single *fetch-and-increment*-based counter C, and an array A of b-bit integers on which the hardware supports atomic *xor* and *decrement* instructions. We assume that A is unbounded. Otherwise, processes can use A to agree on the next array A' to continue the construction. C and the elements of

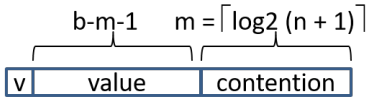


Figure 1 Element of A.

A are initialized by 0. We call an array location *invalid* if it contains a negative value, i.e., if its most significant bit is 1, *empty* if it contains value 0, and *valid* otherwise. The least significant $m = \lceil \log_2 (n+1) \rceil$ bits are *contention bits* and have a special importance to the algorithm. The remaining $b - m - 1$ bits are used to store items. See Figure 1 for illustration.

For every array location, at most one process will ever attempt to record a $(b-m-1)$-bit item, and at most $n-1$ processes will attempt to invalidate this location. No process will try to record to or invalidate the same location twice. In order to record item x, a process invokes $xor(x')$, where x' is x shifted by m bits to the left, plus $2^m - 1 \geq n$, i.e., the contention bits set to 1. To invalidate a location, a process calls a *decrement*. The following properties hold:

1. After a *xor* or *decrement* is performed on a location, no *read* on it ever returns 0.
2. If a *xor* is performed first, no later read returns an invalid value. Ignoring the most significant bit, the next most significant $b - m - 1$ bits contain the item recorded by *xor*.
3. If a *decrement* is performed first, then all values returned by later *read*s are invalid.

A *xor* instruction *fails to record an item* if it is performed after a decrement. To implement a *get-log* operation, process p starts at index $i = 0$, and keeps reading the values of $A[i]$ and incrementing i until it encounters an empty location $A[i] = 0$. By the above properties, from every valid location $A[j]$, it can extract the item x_j recorded by a *xor*, and it returns an ordered list of all such items $(x_{i_1}, x_{i_2}, \ldots, x_{i_k})$. In practice, we require p to return only a suffix of items appended after the last *get-log*() invocation by p. This can be accomplished by keeping i in static memory instead of initializing it to 0 in every invocation. To make *get-log* wait-free, p first performs $l = C.read()$. Then, if i becomes equal to l during the traversal, it stops and returns the items extracted so far. To implement $append(x)$, process p starts by $\ell = C.fetch\text{-}and\text{-}increment()$. Then it attempts to record item x in $A[\ell]$ using an atomic *xor* instruction. If it fails to record an item, the process does another *fetch-and-increment* and attempts *xor* at that location, and so on, until it is able to successfully record x. Suppose this location is $A[\ell']$. Then p iterates from $j = \ell' - 1$ down to $j = 0$, reading each $A[j]$, and if $A[j]$ is empty, performing a *decrement* on it. Afterwards, process p can safely return. The proofs of lock-freedom and linearizability can be found in the full version at `http://arxiv.org/abs/1705.02808`.

We implemented the algorithm on X86 processor and with 32 threads. It gave the same performance as an implementation that used *compare-and-swap* for recording items and invalidating locations. It turns out that in today's architectures, the cost of supporting *compare-and-swap* is not significantly higher than that of supporting *xor* or *decrement*. This may or may not be the case in future Processing-in-Memory architectures [4]. Finding a compact set of synchronization instructions that, when supported, is equally powerful as the set of instructions used today is an important question to establish in future research.

References

1 Faith Ellen, Rati Gelashvili, Nir Shavit, and Leqi Zhu. A complexity-based hierarchy for multiprocessor synchronization:[extended abstract]. In *Proceedings of the 35th ACM Symposium on Principles of Distributed Computing*, 2016.

2 Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 1991.

3 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*, volume 48 of *PPoPP'13*, pages 103–112, 2013.

4 David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A case for intelligent ram. *IEEE Micro*, 17(2):34–44, 1997.

Brief Announcement: On Connectivity in the Broadcast Congested Clique*

Tomasz Jurdziński[1] and Krzysztof Nowicki[2]

1 **Institute of Computer Science, University of Wrocław, Poland**
2 **Institute of Computer Science, University of Wrocław, Poland**

Abstract

Recently, very fast deterministic and randomized algorithms have been obtained for connectivity and minimum spanning tree in the unicast congested clique. In contrast, no solution faster than a simple parallel implementation of the Boruvka's algorithm has been known for both problems in the broadcast congested clique. In this announcement, we present the first sub-logarithmic deterministic algorithm for connected components in the broadcast congested clique.

1998 ACM Subject Classification F.1.2 Modes of Computation, F.2.3 Tradeoffs between Complexity Measures, F.2 Analysis of Algorithms and Problem Complexity

Keywords and phrases congested clique, broadcast, connected components, bandwidth

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.54

1 Introduction

In the congested clique model, each pair of n nodes of a network is connected by a separate communication link. Communication is synchronous, each node in each round can send message of $O(\log n)$ bits to each other node of the network. The main purpose of such a model is to understand the role of congestion in distributed computation, separately from limitations of locality. In the *unicast* congested clique, a node can send (possibly) different message to each other node of the network. In contrast, in the *broadcast* congested clique, each node can only send a single (the same) message to all other nodes in a round.

Graph problems in the congested clique model are considered under the assumption that the input is an undirected n-node weighted graph $G(V, E, w)$, where each node corresponds to a node of the communication network which initially knows the network size n, its unique ID in $[n]$, the IDs of its neighbors in the input graph and the weights of its incident edges. In the connected components problem, the set of edges inducing connected components of the input graph has to be determined.

The main complexity measure is *round complexity*, equal to the number of rounds in an execution of an algorithm. A natural generalization parametrizes the size (in bits) of messages transmitted in a round, called *bandwidth* and denoted by b. Yet another generalization is that the size of messages in various rounds might be different, but not larger than the bandwidth. Then, *bit complexity* is defined as the sum of sizes of messages in all rounds.

Formal study of the congested clique model was initiated in [4], where a $O(\log \log n)$ round deterministic algorithm for minimum spanning tree (MST), and therefore also for the connected components problem, in the unicast congested clique is presented. The best known randomized solution for MST in the unicast model works in $O(\log^* n)$ rounds [2],

* This work was supported by the Polish National Science Centre grant DEC-2012/07/B/ST6/01534.

© Tomasz Jurdziński and Krzysztof Nowicki;
licensed under Creative Commons License CC-BY

improving the $O(\log\log\log n)$ bound [3]. If the bandwidth is $b = \sqrt{n}\log n$, one can compute deterministically Connectivity in $O(1)$ of rounds, even in the broadcast congested clique [5]. In an extreme scenario of one-round protocols in the broadcast congested clique, connected components can be computed with public random bits using $\Theta(\log^3 n)$-size messages [1].

2 Connected Components Algorithm

In this section we describe our algorithm for the connected components problem. In the following $[p]$ denotes the set $\{1, 2, \ldots, p\}$. Given a partition $\mathcal{C}$ of a graph $G(V, E)$ into connected components and $v \in V$, C^v denotes the component containing v. We define $\deg_{\mathcal{C}}(v)$ for a vertex v wrt a partition $\mathcal{C}$ as the number of components connected with v, i.e., $\deg_{\mathcal{C}}(v) = |N_{\mathcal{C}}(v)|$, where $N_{\mathcal{C}}(v) = \{C \in \mathcal{C} \mid \exists u \in C$ such that $(v, u) \in E$ and $C \neq C^v\}$. For a component $C \in \mathcal{C}$, we define $\deg_{\mathcal{C}}(C) = \max_{v \in C}\{\deg_{\mathcal{C}}(v)\}$. Given a partition $\mathcal{C}$ of the graph into components, we define the linear ordering $\succ$ of components, where $C \succ C'$ iff $\deg_{\mathcal{C}}(C) > \deg_{\mathcal{C}}(C')$ or $\deg_{\mathcal{C}}(C) = \deg_{\mathcal{C}}(C')$ and $\mathrm{ID}(C) > \mathrm{ID}(C')$. A component C is a *local maximum* if all its neighbors are smaller with respect to the $\succ$ ordering.

The algorithm consists of the main part and the *playoff* (see Alg. 1). The main part is split into *phases*. In a phase, edges connecting currently build connected components are reported. The edges which connect nodes to the components of large degree are preferred. The intended result of a phase is that each component either has a small degree (smaller than s) or it is connected to some "host" of large degree (directly or by a path). As the number of such "hosts" will be relatively small, we obtain significant reduction of the number of components of large degree in each phase. Moreover, we separately deal with components of small degree by allowing them to broadcast all their neighbours in the playoff.

At the beginning of phase 1 of the main part, each node is *active* and it forms a separate component. During an execution of the algorithm, nodes from components of small degree (smaller than s) are *deactivated*. At the beginning of a phase, a partition $\mathcal{C}$ of the graph of active nodes is known. In Round 1 of a phase, each node v determines $N_{\mathcal{C}}(v)$ and announces its degree $\deg_{\mathcal{C}}(v)$. With this information, each node v knows the ordering of components of $\mathcal{C}$ according to $\succ$. Then, each *active* node v (except of members of local maxima) broadcasts its incident edge to the largest active component from $N_{\mathcal{C}}(v)$ according to $\succ$ (Round 2). Next, each node v of each local maximum C checks whether edges connecting C to all components from $N_{\mathcal{C}}(v)$ have been already broadcasted. If it is not the case, an edge connecting v to a new component C' is broadcasted by v (Round 3). Based on broadcasted edges, new components are determined and their degrees are computed (Round 4). Each new component with degree smaller than s is *deactivated* at the end of a phase.

The playoff lasts s rounds in which each node v of each deactivated component broadcasts an edge going to each component connected to v at the moment of deactivation (there are at most s such components for each deactivated node).

The key property of the algorithm is that each active component C of degree $\geq s$ is either connected during a phase to all its neighbors or to a component which is larger than C according to $\succ$. Thus, the number of active components decreases s times in each phase.

▶ **Theorem 1.** *Alg. 1 solves the spanning forest problem in $O(s + \log_s n)$ rounds.*

For $s = \frac{\log n}{\log\log n}$ Algorithm 1 gives the claimed $o(\log n)$ result.

▶ **Corollary 2.** *It is possible to solve the connected components problem in the broadcast congested clique in $O\left(\frac{\log n}{\log\log n}\right)$ rounds.*

Algorithm 1 BroadcastCC(v, s) ▷ s is the threshold between small/large degree

$\mathcal{C} \leftarrow$ the partition into one-node components $\{v_1\}, \ldots, \{v_n\}$
while there are active components **do** ▷ execution at a node v
  Round 1: v broadcasts $\deg_{\mathcal{C}}(v)$
  if $\deg_{\mathcal{C}}(v) > 0$ **then** ▷ $\mathcal{C}$ – the current partition into connected components
    $C_{\max}(v) \leftarrow$ the largest element of $N_{\mathcal{C}}(v)$ wrt the ordering $\succ$
    Round 2:
    if C^v is not a local maximum **then** v broadcast an edge (u, v) such that $u \in C_{\max}$
    Round 3:
    if C^v is a local maximum **then**
      $N_{\text{lost}}(v) \leftarrow \{C \mid C \in N_{\mathcal{C}}(v)$ and no edge connecting C and C^v was broadcasted$\}$
      if $N_{\text{lost}}(v) \neq \emptyset$ **then**
        $u \leftarrow$ a neighbor of v such that $u \in C$ for some $C \in N_{\text{lost}}(v)$
        v broadcasts an edge (u, v)
  v computes the new partition $\mathcal{C}$ into components, using all broadcasted edges
  Round 4: v broadcasts $\deg_{\mathcal{C}}(v)$ ▷ degrees wrt the new components!
  if $\deg_{\mathcal{C}}(C^v) < s$ **then** deactivate v
  Remove deactivated components from the partition $\mathcal{C}$
Playoff (s rounds): deactivated nodes broadcast edges to neighboring components.

Now, assume that the bandwidth is $b = d \log n$. If $s = d$ in Algorithm 1, we get $\log_d n$ phases, each requiring $O(\log n)$ bits per node. Edges from deactivated nodes can be broadcasted during playoff in one round, using $O(d \log n)$ bits bandwidth. This gives $O(\log_d n)$ round algorithm using $O(\log n(d + \frac{\log n}{\log d}))$ bit complexity.

▶ **Corollary 3.** *It is possible to solve the connectivity problem in the broadcast congested clique with bandwidth $d \log n$ in $\log_d n$ rounds and $O(\log n(d + \frac{\log n}{\log d}))$ bit complexity.*

The above corollary gives an improvement over a result from [5], where bit complexity is $O\left(d\frac{\log^2 n}{\log d}\right)$ in $O(\log_d n)$ rounds. Moreover, our algorithm does not use number theoretic techniques as d-pruning or deterministic sparse linear sketches needed in [5].

Conclusions. We have shown the first sub-logarithmic algorithm for connected components in the broadcast congested clique. On the other hand, it is still not known whether MST can be computed in $o(\log n)$ rounds.

References

1 Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Analyzing graph structure via linear measurements. In *Proceedings of SODA 2012*, pages 459–467, 2012. URL: `http://dl.acm.org/citation.cfm?id=2095116.2095156`.

2 Mohsen Ghaffari and Merav Parter. MST in log-star rounds of congested clique. In *Proceedings of PODC 2016*, pages 19–28, 2016. `doi:10.1145/2933057.2933103`.

3 James W. Hegeman, Gopal Pandurangan, Sriram V. Pemmaraju, Vivek B. Sardeshmukh, and Michele Scquizzato. Toward optimal bounds in the congested clique: Graph connectivity and MST. In *Proceedings of PODC 2015*, pages 91–100, 2015. `doi:10.1145/2767386.2767434`.

4 Zvi Lotker, Boaz Patt-Shamir, Elan Pavlov, and David Peleg. Minimum-weight spanning tree construction in $O(\log \log n)$ communication rounds. *SIAM J. Comput.*, 35(1):120–131, 2005. `doi:10.1137/S0097539704441848`.

5 Pedro Montealegre and Ioan Todinca. Brief announcement: Deterministic graph connectivity in the broadcast congested clique. In *Proceedings of PODC 2016*, pages 245–247, 2016. doi:10.1145/2933057.2933066.

Brief Announcement: Towards a Complexity Theory for the Congested Clique*

Janne H. Korhonen[1] and Jukka Suomela[2]

1 **Aalto University, Department of Computer Science, Aalto, Finland**
`janne.h.korhonen@aalto.fi`
2 **Aalto University, Department of Computer Science, Aalto, Finland**
`jukka.suomela@aalto.fi`

Abstract

The *congested clique* model of distributed computing has been receiving attention as a model for densely connected distributed systems. While there has been significant progress on the side of upper bounds, we have very little in terms of lower bounds for the congested clique; indeed, it is now know that proving explicit congested clique lower bounds is as difficult as proving circuit lower bounds. In this work, we use traditional complexity-theoretic tools to build a clearer picture of the complexity landscape of the congested clique, proving *non-constructive* lower bounds and studying the relationships between natural problems.

1998 ACM Subject Classification F.1.1 Models of Computation; F.1.3 Complexity Measures and Classes

Keywords and phrases distributed computing, congested clique, complexity theory

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.55

1 Introduction

In this work, we study computational complexity questions in the *congested clique* model of distributed computing. The congested clique is essentially a fully-connected specialisation of the classic CONGEST model of distributed computing: There are n nodes that communicate with each other in a fully-connected synchronous network by exchanging messages of size $O(\log n)$. Each node in the network corresponds to a node in an input graph G, each node starts with knowledge about their incident edges in G, and the task is to solve a graph problem related to G.

The congested clique has recently been receiving increasing attention especially on the side of the upper bounds, and the fully-connected network topology allows for significantly faster algorithms than what is possible in the CONGEST model. However, on the side of complexity theory, there has been significantly less development. Compared to the LOCAL and CONGEST models, where complexity-theoretic results have generally taken the form of explicit unconditional lower bounds for concrete problems, such developments have not been forthcoming in the congested clique. Indeed, it was show by Drucker et al. [5] that congested clique lower bounds imply *circuit* lower bounds, and the latter are notoriously difficult to prove – overall, it seems that there are many parallels between computational complexity in the congested clique and *centralised* computational complexity.

* This work was supported in part by the Academy of Finland, Grant 285721. A full version of this paper [8] is available at `https://arxiv.org/abs/1705.03284`.


© Janne H. Korhonen and Jukka Suomela;
licensed under Creative Commons License CC-BY

Following these parallels, we apply concepts and techniques from centralised complexity theory to map out the complexity landscape of the congested clique model. In this brief announcement, we present an overview of our main results. For more details, refer to the full version of this paper [8].

2 Results in brief

Time hierarchy. We prove a time hierarchy theorem for the congested clique, analogous to the centralised time hierarchy theorem [7]. Writing $\mathsf{CLIQUE}\big(T(n)\big)$ for the set of decision problems that can be solved in $O(T(n))$ rounds, we prove the following: for any computable increasing functions S and T with $S(n) = o(T(n))$, we have that

$$\mathsf{CLIQUE}\big(S(n)\big) \subsetneq \mathsf{CLIQUE}\big(T(n)\big)\,.$$

In particular, this stands in contrast to the widely studied distributed computing setting of LCL problems in the LOCAL model, where *complexity gaps* are known to exist [3, 4, 9].

The proof of the time hierarchy theorem is based on the earlier circuit counting arguments for a non-uniform version of the congested clique [1, 5]. We show how to lift this result into the uniform setting, allowing us to show the existence of decision problems of essentially arbitrary complexity.

Nondeterministic congested clique. The class NP and NP-complete problems are central in our understanding of centralised complexity theory. We build towards a similar theory for the congested clique by introducing a *nondeterministic congested clique model*. We define the class $\mathsf{NCLIQUE}\big(T(n)\big)$ as the class of decision problems that have nondeterministic algorithms with running time $O(T(n))$, or equivalently, as the set of decision problems L for which there exists a deterministic algorithm A that runs in $O(T(n))$ rounds and satisfies

$$G \in L \quad \text{if and only if} \quad \exists z\colon A(G, z) = 1\,,$$

where z is a *labelling* assigning each node v a nondeterministic guess z_v.

We show that nondeterminism is only useful up to the number of bits communicated by the algorithm: any nondeterministic algorithm with running time $O(T(n))$ can be converted to a *normal form* in which we only need to use $O(T(n)n \log n)$ bits of nondeterminism. As an application of this result, we show that $\mathsf{NCLIQUE}\big(S(n)\big)$ does not contain $\mathsf{CLIQUE}\big(T(n)\big)$ for any $S(n) = o(T(n))$.

Constant-round nondeterministic decision. We argue that the class $\mathsf{NCLIQUE}(1)$ is a natural complexity class of interest; it can be seen as the analogue of the class NP in centralised computing, and the class LCL in the LOCAL model of distributed computing. In particular, the question of proving that $\mathsf{CLIQUE}(1) \neq \mathsf{NCLIQUE}(1)$ can be seen as playing a role similar to the P vs. NP question. While we cannot prove a separation between deterministic and nondeterministic constant time, we identify a family of canonical edge labelling problems for $\mathsf{NCLIQUE}(1)$, which give a limited notion of *completeness* for $\mathsf{NCLIQUE}(1)$.

Constant-round decision hierarchy. We extend the notion of nondeterministic clique by studying a *constant-round decision hierarchy*, where each node runs an *alternating* Turing machine, similarly to the recent work in the LOCAL model [2, 6] – the centralised analogue is the polynomial hierarchy. Unlike for nondeterministic algorithms, it turns out that the label

size for algorithms on the higher levels of this hierarchy is not bounded by the amount of communication. Thus, we get two very different versions of this hierarchy:

- *Unlimited hierarchy* $(\Sigma_k, \Pi_k)_{k=1}^{\infty}$ with unlimited label size: we show that this version of the hierarchy collapses, as all decision problems are contained on the second level.
- *Logarithmic hierarchy* $(\Sigma_k^{\log}, \Pi_k^{\log})_{k=1}^{\infty}$ with label size $O(n \log n)$ per node: we show that there are problems that are not contained in this hierarchy.

Fine-grained complexity. There are decision problems of all complexities, but it is beyond our current techniques to prove lower bounds for any specific problem, assuming we exclude lower bounds resulting from input or output sizes. However, what we can do is study the relative complexity of natural problems, much in the vein of centralised *fine-grained* complexity: for a problem P, we define the *exponent* of P as

$$\delta(P) = \inf\{\delta \in [0,1] \colon P \text{ can be solved in } O(n^\delta) \text{ rounds}\}.$$

The basic idea is that the problem exponent captures the polynomial complexity of the problem, and we can compare the relative complexity of problems by comparing their exponents. In the full version of this paper [8], we map out some known relationships between prominent problems in the congested clique using this framework.

Acknowledgements. We thank Alkida Balliu, Parinya Chalermsook, Juho Hirvonen, Petteri Kaski, Dennis Olivetti and Christopher Purcell for comments and discussions.

References

1 Benny Applebaum, Dariusz R. Kowalski, Boaz Patt-Shamir, and Adi Rosén. Clique here: On the distributed complexity in fully-connected networks. *Parallel Processing Letters*, 26(01):1650004, 2016. doi:10.1142/S0129626416500043.

2 Alkida Balliu, Gianlorenzo D'Angelo, Pierre Fraigniaud, and Dennis Olivetti. What can be verified locally? In *Proc. 34th Symposium on Theoretical Aspects of Computer Science (STACS 2017)*, volume 66, pages 8:1–8:13, 2017. doi:10.4230/LIPIcs.STACS.2017.8.

3 Yi-Jun Chang, Tsvi Kopelowitz, and Seth Pettie. An exponential separation between randomized and deterministic complexity in the LOCAL model. In *Proc. 57th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2016)*, pages 615–624, 2016.

4 Yi-Jun Chang and Seth Pettie. A time hierarchy theorem for the LOCAL model, 2017. arXiv:1704.06297 [cs.DC].

5 Andrew Drucker, Fabian Kuhn, and Rotem Oshman. On the power of the congested clique model. In *Proc. 33rd ACM Symposium on Principles of Distributed Computing (PODC 2014)*, pages 367–376, 2014. doi:10.1145/2611462.2611493.

6 Laurent Feuilloley, Pierre Fraigniaud, and Juho Hirvonen. A hierarchy of local decision. In *Proc. 43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016)*, volume 55, pages 118:1–118:15, 2016. doi:10.4230/LIPIcs.ICALP.2016.118.

7 Juris Hartmanis and Richard E. Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965.

8 Janne H. Korhonen and Jukka Suomela. Towards a complexity theory for the congested clique, 2017. arXiv:1705.03284 [cs.DC].

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

Brief Announcement: Compact Topology of Shared-Memory Adversaries*

Petr Kuznetsov[1], Thibault Rieutord[2], and Yuan He[3]

1 **LTCI, Télécom ParisTech, Université Paris-Saclay, Paris, France**
petr.kuznetsov@telecom-paristech.fr
2 **LTCI, Télécom ParisTech, Université Paris-Saclay, Paris, France**
thibault.rieutord@telecom-paristech.fr
2 **UCLA, Los Angeles, USA**
yuan.he@cs.ucla.edu

Abstract

The paper proposes a simple topological characterization of a large class of *adversarial* distributed-computing models via *affine tasks*: sub-complexes of the second iteration of the standard chromatic subdivision. We show that the task computability of a model in the class is precisely captured by iterations of the corresponding affine task. While an adversary is in general defined as a *non-compact* set of infinite runs, its affine task is just a finite subset of runs of the 2-round iterated immediate snapshot (IIS) model. Our results generalize and improve all previously derived topological characterizations of distributed-computing models.

1998 ACM Subject Classification C.2.4 Distributed Systems, F.1.1 Models of Computation.

Keywords and phrases Adversarial models, Affine tasks, Topological characterization.

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.56

1 Introduction

Distributed computing is a jungle of models, parameterized by types of failures, synchrony assumptions, and employed communication primitives. Determining relative computability power of these models ("is model A more powerful than model B") is an intriguing and important problem.

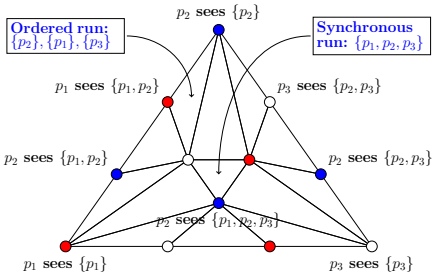
The task computability of the *wait-free* model of computation, which makes no assumptions about the number of failures that can occur, was characterized by Herlihy and Shavit [7] through the existence of a specific continuous map from a subdivision of the input complex of a task $\mathcal{I}$ to its output complex $\mathcal{O}$. (The reader is referred to [6] for a thorough discussion of the use of combinatorial topology in distributed computability.) In particular, the characterization can consider the *iterated* standard chromatic subdivision ($\mathrm{Chr}\,\mathbf{s}$ depicted in Figure 1a) and, thus, derive that a task is wait-free solvable if and only if it can be solved in the IIS model.

The aim of this paper is to generalize this topological characterization to models beyond the wait-free model using the formalism of *affine tasks* [5]. An affine task is defined through a pure subcomplex of an iterated *standard chromatic subdivision* or, equivalently, a subset of finite runs of the IIS model. Iterations of such affine tasks provide a failure-less *compact* model (according to the "longest-prefix" metric [1]).

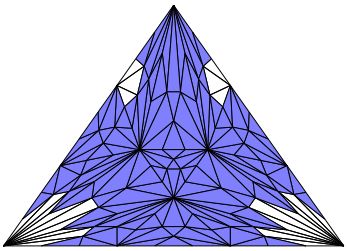
Given that many fundamental tasks are not wait-free solvable, the prominent *adversarial* failure model [2] has been introduced to strengthen the wait-free model. An adversary

* A full version of the paper is available at https://hal.archives-ouvertes.fr/hal-01572257.


© Petr Kuznetsov, Thibault Rieutord, and Yuan He;
licensed under Creative Commons License CC-BY



(a) $\mathrm{Chr}^1(\mathbf{s})$, equivalent to the output complex of the immediate snapshot (IS) task.



(b) The affine task of the adversary consisting of $\{p_1\}$, $\{p_2, p_3\}$ with all super-sets (in blue).

Figure 1 Standard chromatic subdivision and an affine task example for $n = 3$.

$\mathcal{A}$ is defined through a collection of process subsets, called *live sets*. In every run of the corresponding *adversarial* $\mathcal{A}$*-model*, the set of processes taking infinitely many steps must be a live set. The sub-class of *fair* adversaries [8] does not, intuitively, allow a *subset* of processes participating in a computation to achieve a better set consensus than the whole set of participants (processes taking at least one step). The class of *fair* adversaries is pretty large, as it includes the existing sub-classes of *superset-closed* and *symmetric* adversaries.

We show that a specific affine task $\mathcal{R}_\mathcal{A}$, defined as a subcomplex of the second iteration of the standard chromatic subdivision, captures the *task computability* of any fair adversary $\mathcal{A}$. A task is solved in the $\mathcal{A}$-model if and only it is solvable in the set of IIS runs resulting from iterations of $\mathcal{R}_\mathcal{A}$ (denoted $\mathcal{R}^*_\mathcal{A}$).

The notion of *agreement functions* [8] was instrumental for this result. (An agreement function α associates each set of processes P with the best level of set consensus solvable when only processes in P might participate.) Fair adversaries are characterized by their agreement function in the sense that they belong to the weakest equivalence class (in terms of task computability) of models with the same agreement function. Our characterization can then be put as a generalization of the celebrated Asynchronous Computability Theorem (ACT) [7]:

> A task $T = (\mathcal{I}, \mathcal{O}, \Delta)$, where $\mathcal{I}$ is the input complex, $\mathcal{O}$ is an output complex, and Δ is a map from $\mathcal{I}$ to sub-complexes of $\mathcal{O}$, is solvable in a fair adversarial $\mathcal{A}$-model if and only if there exists a natural number ℓ and a simplicial map $\phi : \mathcal{R}^\ell_\mathcal{A}(\mathcal{I}) \to \mathcal{O}$ carried by Δ (informally, respecting the task specification Δ).

2 Affine tasks for fair adversaries.

Two classes of affine tasks were recently defined. The class $\mathcal{R}_{t-res}$ was introduced in [10], with $\mathcal{R}^*_{t-res}$ equivalent to the t-resilient model. Similarly, the class $\mathcal{R}_k$ was introduced in [4], with $\mathcal{R}^*_k$ equivalent to the k-concurrent model. Interestingly, these models correspond to two "well-behaved" sub-classes of fair adversaries on opposite sides of the spectrum. In a sense, a fair adversary can be seen as a combination of *concurrency* and *resilience*, grasped using, resp., *contention* and *critical* simplices:

- **Contention simplices:** If processes are executed sequentially, they not only obtain distinct views out of IS, but also obtain the same view (inclusion) ordering out of multiple iterations. But to be combined with resiliency features, concurrency restrictions must be weaken to focus on "fully" conflicting processes. This is why we say that a simplex, or a group of processes, forms a 2-*contention simplex* if any two of its processes have distinct views in both IS iterations, ordered alternatively in each (see [9] for a formal definition).

- **Critical simplices:** For fair adversaries, concurrency may rise with participation irregularly. Critical simplices act as representatives of each increase of participation resulting in a concurrency increase. They are selected among processes with the smallest first IS output providing an observed participation corresponding to some non-nul level of set-consensus power. Moreover, in order to be identifiable as critical, they are selected as critical simplices only if grouped with sufficiently many other processes with the same IS output, so that if they are withdrawn from their own first IS observed participation the remaining participation is associated to a strictly smaller set-consensus power. Hence observing in the second IS all members of a critical simplex is enough to check that they together form a critical simplex.

Now we are ready to define the subcomplex $\mathcal{R}_A \subseteq \text{Chr}^2 \mathbf{s}$. The idea is that a large 2-constention simplex may be allowed only if it terminates after a critical simplex associated with a large enough view, i.e., concurrency may rises only after sufficient ensured resilience. A $(n-1)$-dimensional simplex $\sigma \in \text{Chr}^2 \mathbf{s}$ (composed of n vertices) belongs to $\mathcal{R}_A$ if and only if every sub-simplex of σ of size k which (1) is a 2-contention simplex; (2) does not include critical simplices members; (3) does not include processes observed by identifiable critical simplices (with a smaller second IS view); must observe a critical simplex with a first IS view associated to an agreeent power greater than or equal to k (see Figure 1b for an example).

The proof of equivalence between $\mathcal{R}^*_{\mathcal{A}}$ and the fair $\mathcal{A}$-adversary model is done using the equivalent α-model [8] which (1) allows for a simple resolution of $\mathcal{R}_{\mathcal{A}}$ in the α-model by simply executing two rounds of an IS algorithm, with a waiting phase in between the rounds (similarly to [10]); and (2) can be simulated easily as soon as α-adaptive set-consensus (see [8]) is solvable in the presence of read-write memory (similarly to [4]).

To summarize, this paper generalizes all previous topological characterizations of distributed computing models [7, 5, 4, 10]. We believe that the results can further be extended to all "practical" restrictions of the wait-free model of computations, beyond fair adversaries, which may potentially result in a complete computability theory for distributed computing [3].

References

1. Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.
2. Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, and Andreas Tielmann. The disagreement power of an adversary. *Distributed Computing*, 24(3-4):137–147, 2011.
3. Eli Gafni. Private communication. 2002.
4. Eli Gafni, Yuan He, Petr Kuznetsov, and Thibault Rieutord. Read-Write Memory and k-Set Consensus as an Affine Task. In *OPODIS 2016*, volume 70, pages 6:1–6:17, 2017. doi:10.4230/LIPIcs.OPODIS.2016.6.
5. Eli Gafni, Petr Kuznetsov, and Ciprian Manolescu. A generalized asynchronous computability theorem. In *PODC*, pages 222–231, 2014. doi:10.1145/2611462.2611477.
6. Maurice Herlihy, Dmitry N. Kozlov, and Sergio Rajsbaum. *Distributed Computing Through Combinatorial Topology*. Morgan Kaufmann, 2014.
7. Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(2):858–923, 1999.
8. Petr Kuznetsov and Thibault Rieutord. Agreement functions for distributed computing models. In *NETYS*, pages 175–190, 2017. doi:10.1007/978-3-319-59647-1_14.
9. Petr Kuznetsov, Thibault Rieutord, and Yuan He. Compact topology of shared-memory adversaries. *HAL*, https://hal.archives-ouvertes.fr/hal-01572257, 2017.

10 Vikram Saraph, Maurice Herlihy, and Eli Gafni. Asynchronous computability theorems for t-resilient systems. In *DISC*, pages 428–441, 2016.

Brief Announcement: A Centralized Local Algorithm for the Sparse Spanning Graph Problem*†

Christoph Lenzen[1] and Reut Levi[2]

1 **Max Planck Institute for Informatics, Saarland Informatics Campus, Saarbrücken, Germany**
clenzen@mpi-inf.mpg.de

2 **Max Planck Institute for Informatics, Saarland Informatics Campus, Saarbrücken, Germany**
rlevi@mpi-inf.mpg.de

Abstract

Constructing a sparse *spanning subgraph* is a fundamental primitive in graph theory. In this paper, we study this problem in the Centralized Local model, where the goal is to decide whether an edge is part of the spanning subgraph by examining only a small part of the input; yet, answers must be globally consistent and independent of prior queries.

Unfortunately, maximally sparse spanning subgraphs, i.e., spanning trees, cannot be constructed efficiently in this model. Therefore, we settle for a spanning subgraph containing at most $(1+\epsilon)n$ edges (where n is the number of vertices and ϵ is a given approximation/sparsity parameter). We achieve a query complexity of $\tilde{O}(\text{poly}(\Delta/\varepsilon)n^{2/3})$,[1] where Δ is the maximum degree of the input graph. Our algorithm is the first to do so on arbitrary bounded degree graphs. Moreover, we achieve the additional property that our algorithm outputs a *spanner,* i.e., distances are approximately preserved. With high probability, for each deleted edge there is a path of $O(\log n \cdot (\Delta + \log n)/\epsilon)$ hops in the output that connects its endpoints.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases local, spanning graph, sparse

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.57

1 Introduction

When operating on very large graphs, it is often impractical or infeasible to (i) hold the entire graph in the local memory of a processing unit, (ii) run linear-time (or even slower) algorithms, or even (iii) have only a single processing unit perform computations sequentially. These constraints inspired the Centralized Local model [9], which essentially views the input as being stored in a (likely distributed) database that provides query access to external processing units. To minimize the coordination overhead of such a system, it is furthermore required that there is no shared memory or communication between the querying processes, except for shared randomness provided alongside the access to the input. The idea is now to run sublinear-time algorithms that extract useful global properties of the graph and/or to examine the input graph locally upon demand by applications.

* A full version of the paper is available at http://arxiv.org/abs/1703.05418.
† See [2] for the full version of this paper.
[1] $\tilde{O}$-notation hides polylogarithmic factors in n.


© Christoph Lenzen and Reut Levi;
licensed under Creative Commons License CC-BY

Studying graphs in this model leads to the need for query access to a variety of graph-theoretical structures like, e.g., independent or dominating sets. In such a case, it is crucial that locally evaluating whether a node participates in such a set is *consistent* with the same evaluation for other nodes. This is a non-trivial task, as local decisions can only be coordinated implicitly via the structure of the input (which is to be examined as little as possible) and the shared randomness. Nonetheless, this budding field brought forth a number of elegant algorithms solving, e.g., maximal independent set, hypergraph coloring, k-CNF, and approximate maximum matching (see survey [6] and references therein).

In this work, we consider another very basic graph structure: sparse spanning subgraphs. Here, the task is to select a subset of the edges of the (connected) input graph so that the output is still connected, but has only few edges. By "few" we mean that, for some input parameter $\varepsilon > 0$, the number of selected edges is at most $(1 + \varepsilon)n$, where n denotes the number of nodes. One may see this as a relaxed version of the problem of outputting a spanning tree of the graph, which can not be obtained in sub-linear number of queries.

▶ **Definition 1** ([5])**.** An algorithm $\mathcal{A}$ is a *Local Sparse Spanning Graph (LSSG) algorithm* if, given $n, \Delta \geq 1$, a parameter $\epsilon \geq 0$, and query access to the incidence list representation of a connected graph $G = (V, E)$ over n vertices and of degree at most Δ, it provides oracle access to a subgraph $G' = (V, E')$ of G such that: (1) G' is connected. (2) $|E'| \leq (1 + \epsilon) \cdot n$ with high probability (w.h.p.),[2] where E' is determined by G and the internal randomness of $\mathcal{A}$. By "providing oracle access to G'" we mean that on input $\{u, v\} \in E$, $\mathcal{A}$ returns whether $\{u, v\} \in E'$, and for any sequence of edges, $\mathcal{A}$ answers consistently w.r.t. the same G'.

We are interested in LLSG algorithms that, for each given edge, perform as few queries as possible to G. Observe that Item (2) implies that the answers of an LLSG algorithm to queries cannot depend on previously asked queries. We note that relaxing from requiring a tree as output makes it possible to ask for additional guarantees. Instead of merely preserving connectivity, it becomes possible to maintain *distances* up to small factors. Such subgraphs are known as (sparse, multiplicative) *spanners* [7, 8].

Our Contribution. We give the first non-trivial LSSG algorithm in the Centralized Local model that runs on arbitrary graphs. We achieve a query complexity of $\tilde{O}(\text{poly}(\Delta/\varepsilon)n^{2/3})$ per edge, w.h.p. Moreover, we guarantee that for each edge that is not selected into the spanner, w.h.p. there is a path of $O(\log n \cdot (\Delta + \log n)/\epsilon)$ hops consisting of edges that are selected into the spanner; this is referred to as a *stretch* of $O(\log n \cdot (\Delta + \log n)/\epsilon)$.

For simplicity, assume for the moment that Δ and ε are constants. Our algorithm combines the following key ideas. We classify edges as expanding if there are sufficiently many (roughly $n^{1/3}$) nodes within $O(\log n)$ hops of their endpoints. For non-expanding edges, we can efficiently simulate a standard distributed spanner algorithm at small query complexity, as solutions of running time $O(\log n)$ are known (e.g. [1]).

On the node set induced by expanding edges, we can construct a partition into Voronoi cells with respect to roughly $n^{2/3}$ randomly sampled centers. The Voronoi cells are spanned by trees of depth $O(\log n)$, as expanding nodes have their closest center within $O(\log n)$ hops w.h.p. Finding the closest center has query complexity $\tilde{O}(n^{1/3})$.

We refine the partition into Voronoi cells further into *clusters* of $\tilde{O}(n^{1/3})$ nodes. We simply let a node be a singleton cluster if its subtree in the spanning tree of its cell contains more than $\tilde{O}(n^{1/3})$ nodes. This construction has query complexity $\tilde{O}(n^{2/3})$ for constructing a complete cluster, yet ensures that there are $\tilde{O}(n^{2/3})$ clusters in total due to the low depth of the trees; moreover, each cluster is completely contained in some Voronoi cell.

[2] That is, with probability at least $1 - 1/n^c$ for an arbitrary constant $c > 0$ that is chosen upfront.

It remains to select few edges to interconnect the Voronoi cells. This is the main challenge, for which the above properties of the partition are crucial. To keep the number of selected edges small in expectation, we mark a subset of expected size $\tilde{\Theta}(n^{1/3})$ of the clusters by marking each Voronoi cell (and thereby its constituent clusters) with probability $n^{-1/3}$. We then try to ensure that (i) clusters select an edge to each adjacent marked Voronoi cell and (ii) for each marked Voronoi cell adjacent to an adjacent cluster, they select one edge connecting to *some* cluster adjacent to this cell.

The main issue with the previous step is that we cannot afford to construct each adjacent cluster, preventing us from guaranteeing (ii). We circumvent this obstacle by identifying for adjacent clusters in which cell they are and keeping an edge for the purpose of (ii) if it satisfies a certain minimality requirement with respect to the *rank* of the cell used for symmetry breaking purposes. This way, we avoid construction of adjacent clusters, instead needing to determine the rank of their Voronoi cells only. This way, we maintain query complexity $\tilde{O}(n^{2/3})$. However, this now entails an inductive argument for ensuring connectivity, which also affects stretch. By choosing Voronoi cell ranks uniformly at random, we obtain a total bound of $O(\log^2 n)$ on the stretch of our scheme.

Related work. The problem of finding a sparse spanning subgraph in the Centralized Local model was first studied in [5], where the authors show a lower bound of $\Omega(\sqrt{n})$ queries for constant ϵ and Δ.They also present an upper bound with nearly tight query complexity for graphs that have very good expansion properties. However, for general (bounded degree) graphs their algorithm might query the entire graph for completing a single call to the oracle. They also provide an efficient algorithm for minor-free graphs that was later improved in [4]. A characterization of the query complexity of the problem in terms of expansion properties of the input graph was presented in [3].

References

1 Michael Elkin and Ofer Neiman. Efficient algorithms for constructing very sparse spanners and emulators. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 652–669, 2017.

2 Christoph Lenzen and Reut Levi. A local algorithm for the sparse spanning graph problem. *CoRR*, abs/1703.05418, 2017. URL: `http://arxiv.org/abs/1703.05418`.

3 R. Levi, G. Moshkovitz, D. Ron, R. Rubinfeld, and A. Shapira. Constructing near spanning trees with few local inspections. *Random Structures & Algorithms*, pages n/a–n/a, 2016. `doi:10.1002/rsa.20652`.

4 R. Levi and D. Ron. A quasi-polynomial time partition oracle for graphs with an excluded minor. *ACM Trans. Algorithms*, 11(3):24:1–24:13, 2015.

5 R. Levi, D. Ron, and R. Rubinfeld. Local algorithms for sparse spanning graphs. In *Proceedings of the Eighteenth International Workshop on Randomization and Computation (RANDOM)*, pages 826–842, 2014.

6 Reut Levi and Moti Medina. A (centralized) local guide. *Bulletin of EATCS*, 2(122), 2017.

7 D. Peleg and A. A. Schäffer. Graph spanners. *Journal of Graph Theory*, 13:99–116, 1989.

8 D. Peleg and J. D. Ullman. An optimal synchronizer for the hypercube. *SIAM Journal on Computing*, 18:229–243, 1989.

9 R. Rubinfeld, G. Tamir, S. Vardi, and N. Xie. Fast local computation algorithms. In *Proceedings of The Second Symposium on Innovations in Computer Science (ICS)*, pages 223–238, 2011.

Brief Announcement: Distributed SplayNets*

Bruna S. Peres[1], Olga Goussevskaia[2], Stefan Schmid[3], and Chen Avin[4]

1 **Computer Science Department, Universidade Federal de Minas Gerais, Brazil**
bperes@dcc.ufmg.br
2 **Computer Science Department, Universidade Federal de Minas Gerais, Brazil**
olga@dcc.ufmg.br
3 **Department of Computer Science, Aalborg University, Denmark**
schmiste@cs.aau.dk
4 **Comm. Sys. Eng. Department, Ben Gurion University of the Negev, Israel**
avin@cse.bgu.ac.il

Abstract

SplayNets are reconfigurable networks which adjust to the communication pattern over time. We present *DiSplayNets*, a distributed (concurrent and decentralized) implementation of SplayNets.

1998 ACM Subject Classification C.2.1 Network Architecture and Design, Distributed networks

Keywords and phrases Decentralization, Concurrency, Reconfigurable Networks

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.58

1 Introduction

SplayNets [3] are locally-routable tree networks whose topology self-adjusts to the workload: nodes communicating more frequently become topologically closer to each other over time. SplayNets are hence reminiscent of classic splay tree *data structures*: however, in contrast to splay trees where requests always originate from the root, in a SplayNet, requests occur between arbitrary *node pairs*. SplayNets are motivated, among other, by the advent of reconfigurable datacenter interconnects like ProjecToR [2], and are very different from many traditional network designs which are either entirely oblivious to the communication demand or are optimized towards the demand but cannot be reconfigured over time [1].

In this work, we present DiSplayNets, the first distributed, i.e., decentralized and concurrent implementation of SplayNets. Moving from centralized-sequential to decentralized-concurrent algorithms is challenging, as simultaneous local network reconfigurations can interfere, potentially leading to starvation or even deadlocks, and hence ruining the potential benefits of concurrent operations. Moreover, it needs to be ensured that traffic forwarding and (locally/greedy) routing is unaffected by the topological changes.

We present a distributed algorithm that overcome these challenges, and demonstrate that decentralized SplayNets are feasible.

▶ **Theorem 1.** *DiSplayNets self-adjust to the communication pattern in a fully-decentralized manner, eventually serving all communication requests (in a starvation- and deadlock-free manner).*

* This work is part of the PhD thesis of B.S. Peres and was supported by CNPq, CAPES, Fapemig, and partially by the German-Israeli Foundation for Scientific Research (GIF) Grant I-1245-407.6/2014


© Bruna S. Peres, Olga Goussevskaia, Stefan Schmid, and Chen Avin;
licensed under Creative Commons License CC-BY

2 DiSplayNets

Background and Model. We want to design a tree $\mathcal{T}$ (i.e., a SplayNet) which adjusts according to a sequence $\sigma = (\sigma_0, \sigma_1, \ldots, \sigma_{m-1})$ of communication requests occurring over time, where $\sigma_i = (s, d)$ denotes that source $src(\sigma_i) = s$ communicates to destination $dst(\sigma_i) = d$. Each request σ_i is generated in some time-slot $t_b(\sigma_i)$, and we will denote its completion time (which depends on the algorithm) by $t_e(\sigma_i)$.

SplayNets are Binary Search Trees (BST) and hence naturally support greedy routing. SplayNets aggressively move communicating nodes together, using the classic splay operations zig, zig-zig and zig-zag [4] of splay trees. However, rather than splaying nodes to the root of the BST, in contrast to splay tree data structures, locality is preserved in that the source and the destination nodes are only rotated to their *least common ancestor* ($LCA(s, d)$).

This motivates us to use the following notion of *splay request* $\mathcal{S}_i(s, d)$: A splay request between the source and destination node of a communication request $\sigma_i(s, d) \in \sigma$ is comprised of two sequences of local network transformations, requested by s and d, with the objective to bring these two nodes topologically closer, without violating the BST properties. We say that the splay request $\mathcal{S}_i(s, d)$ has been completed in time-slot t' if the distance $d_{t'}(s, d) = 1$, i.e., s becomes the parent of d or vice versa, whichever happens first.

We define a *distributed* SplayNet as follows: **DiSplayNet** $\mathcal{T}_t = (V, E_t)$ is comprised of n nodes, with distinct identifiers, interacting *concurrently* according to the communication pattern of σ. In each time-slot t, the set of edges E_t connects the nodes in V, s.t. $\mathcal{T}_t$ is a BST. We assume that the execution starts with an arbitrary BST topology $\mathcal{T}_0$. Each node u stores the identifiers of its direct neighbors in the tree, i.e., its parent ($u.p$), its left child ($u.l$) and its right child ($u.r$), and the smallest (u') and the largest (u'') identifiers currently present in the sub-tree rooted at u. This information is used for local routing and for splaying.

DiSplayNet Design and Distributed Algorithm. In DiSplayNet, a changing communication pattern leads to local adjustments (possibly concurrently to prior requests) of the communication links in $\mathcal{T}_t$ over time. Consider a DiSplayNet instance $\mathcal{T}_t$, and a communication request $\sigma_i(s, d) \in \sigma, t_b(\sigma_i) \leq t$. Differently from sequential SplayNets, s and d rotate *in parallel* towards the $LCA_t(s, d)$. Due to concurrency, the LCA might change over time. Therefore, instead of approaching a specific LCA node, s and d keep splaying towards the root of $\mathcal{T}_t$, until becoming each other's ancestor. Upon generating a request $\sigma_i(s, d)$, node s must advertise node d so that both start splaying.

DiSplayNet can be described in terms of a state machine, executed by each node in parallel: (1) *Passive:* a node is in passive state in time-slot t if it is not the source or destination of any request in $\sigma_i \in \sigma, t_b(\sigma_i) \leq t$; (2) *Climbing:* a node s (or d) is climbing in time-slot t if it has an *active request*: $\exists \sigma_i(s, d) \in \sigma, t_b(\sigma_i) \leq t$ and $d_t(s, d) > 1$, and additionally s (or d) $\neq LCA_t(s, d)$; (3) *Waiting:* a node s (or d) is waiting in time-slot t if has an active request and $s = LCA_t(s, d)$. (4) *Communicating:* a node s or d is communicating in time-slot t if $\exists \sigma_i(s, d) \in \sigma, t_b(\sigma_i) \leq t$ and $d_t(s, d) = 1$. Figure 1 shows the states transitions.

In order to ensure deadlock and starvation freedom, concurrent rotations are performed according to a *priority*. An older request in the network has a higher priority than a more recent request (Figure 2). Note that, a node s in the *waiting* state might be removed from the LCA position by a rotation with higher priority. If that happens, s returns to the *climbing* state and resumes requesting rotations. Finally, when s and d meet, they communicate.

To synchronize the process between the nodes, we execute the algorithm in rounds. Each round is composed of five phases (1. Rotation Requests; 2. Top-down Acks; 3. Bottom-up

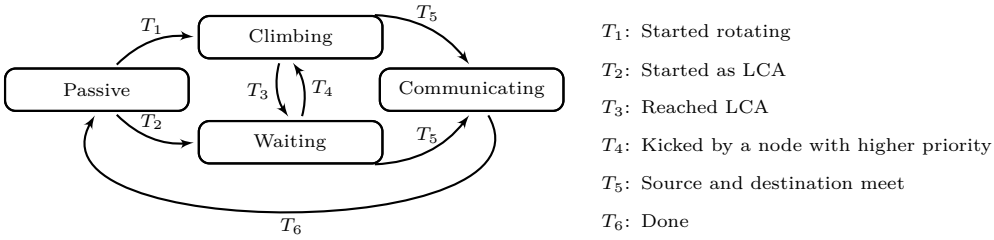


Figure 1 State Transition Diagram.

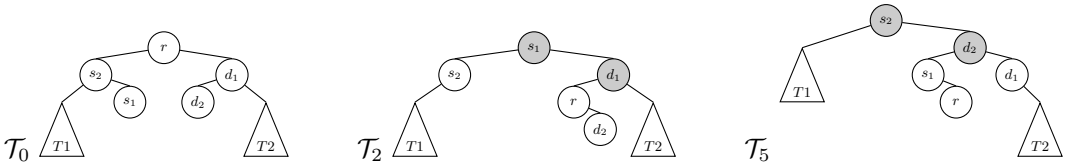


Figure 2 DiSplayNets: $\sigma_1(s_1, d_1)$ and $\sigma_2(s_2, d_2) \in \sigma$, $t_e(\sigma_1) < t_e(\sigma_2)$.

Algorithm 1 The distributed DiSplayNets algorithm (one round).

1: Rotation Requests (3 time-slots)
if Climbing for some $\sigma_i(s, d)$ **then**
send own rotation request $\beta(u)$ upward;
insert $\beta(u)$ into buffer;
upon receiving rotation request $\beta(v)$:
insert $\beta(v)$ into buffer;
forward $\beta(v)$ upward;

2: Top-down Acks (3 time-slots)
get highest priority request $\beta(x)$ in buffer
if Master($\beta(x)$) **then** ▷ farthest node from x
send Ack($\beta(x)$) downward

3: Bottom-up Acks (3 time-slots)
upon receiving top-down ack($\beta(v)$)
if $\beta(v) = \beta(x)$ **then** ▷ highest priority ack
send ack($\beta(v)$) up toward master

4: Link Updates (1 time-slot)
if received bottom-up ack($\beta(x)$) **then**
update links according to $\beta(x)$;

5: State Updates (1 time-slot)
update state; ▷ Figure 1
clear buffer;

Acks; 4. Link Updates; 5. State Updates), summarized in Algorithm 1. Each node u maintains a local buffer, containing a queue of rotation requests, generated by itself, its right or left child, one of its four grandchildren or eight great-grandchildren. In each round, each rotation request $\beta(u)$ is sent upwards until reaching its *master* (2 hops ancestor in case of a zig and 3 hops ancestor in case of a a zig-zig or zig-zag). Once all requests have been received, the highest priority request is acknowledged top down, from master to requesting node. Upon receiving a top-down ack, the requesting node sends an acknowledgment upwards to the master. We say that neighboring nodes agree to perform rotation $\beta(u)$ if all of them received one top-down and one bottom-up acknowledgment for $\beta(u)$.

Future Work. It remains to rigorously analyze the efficiency, i.e., amortized work and time. Our simulations show first promising results.

References

1 Avin et al. Demand-aware network designs of bounded degree. In *DISC*, 2017.
2 M. Ghobadi et al. Projector. In *ACM SIGCOMM*, 2016.
3 S. Schmid, C. Avin, C. Scheideler, M. Borokhovich, B. Haeupler, and Z. Lotker. Splaynet: Towards locally self-adjusting networks. *IEEE/ACM Tran. on Networking*, (99):1–13, 2015.
4 D. Sleator and R. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.

Brief Announcement: Rapid Mixing of Local Dynamics on Graphs

Laurent Massoulié[1] and Rémi Varloot[2]

1 **Inria, MSR-Inria Joint Centre, Palaiseau, France**
`laurent.massoulie@inria.fr`
2 **Inria, MSR-Inria Joint Centre, Palaiseau, France**
`remi.varloot@inria.fr`

Abstract

In peer-to-peer networks, it is desirable that the logical topology of connections between the constituting nodes make a well-connected graph, i.e., a graph with low diameter and high expansion. At the same time, this graph should evolve only through local modifications. These requirements prompt the following question: are there local graph dynamics that i) create a well-connected graph in equilibrium, and ii) converge rapidly to this equilibrium?

In this paper we provide an affirmative answer by exhibiting a local graph dynamic that mixes provably fast. Specifically, for a graph on N nodes, mixing has occurred after each node has performed $O(\text{polylog}(N))$ operations. This is in contrast with previous results, which required at least $\Omega(N \,\text{polylog}(N))$ operations per node before the graph had properly mixed.

1998 ACM Subject Classification E.1 Graphs and Networks

Keywords and phrases Markov chains, Mixing time, Dynamic graphs, Local dynamics

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.59

1 Introduction

Peer-to-peer networks perform best if the graph describing their topology is well-connected. Indeed, the diameter of the graph conditions the time required to broadcast information from any one node to the rest of the network. The expansion of the graph conditions the robustness of epidemic algorithms for maintaining content in the system. It also conditions how quickly a random walk over the graph reaches stationarity, and as such determines the performance of various distributed algorithms, e.g., for searching content over the network.

The distributed evolution of such networks, however, can potentially create ill-connected graphs through an unlucky series of node arrivals and departures. This motivates our goal in the present paper: identify local graph dynamics that create a well-connected graph in a short amount of time, i.e., after each node has performed few operations ($O(\text{polylog}(N))$, where N denotes the total number of nodes), and this regardless of how poorly connected the initial graph is.

2 Related Work

Graph models meant to capture properties of real-life networks have been thoroughly studied [7]. Important examples include the Barabási-Albert preferential attachment model, yielding graphs with power law degree distribution [1], and random regular graphs, shown to have the small-world property of social networks (i.e., a small diameter) with high probability [12, 15].


© Laurent Massoulié and Rémi Varloot;
licensed under Creative Commons License CC-BY

Other papers address the design of dynamics meant to alter the overall structure of the graph in a given way [13] or to minimize the convergence rate [9]. Closer to our present work, these issues have been considered specifically for the construction of p2p networks in [6, 5, 8].

The speed of convergence to equilibrium of graph dynamics has been studied in different contexts. [4] considers non-local dynamics. Local dynamics are considered in [14] and [10] for the synthesis of cladograms and bipartite graphs respectively. Closer to our context, [2] considers the local switch dynamic introduced in [8], and proves that it yields an expander graph after $O(N \operatorname{polylog}(N))$ operations per node. The analysis in [2] is very elaborate, and the stated bound improves upon all previous results on local graph dynamics proposed for peer-to-peer topology maintenance. Nevertheless, this bound is not yet satisfactory, as it still increases quickly (at least linearly) with the system size.

3 Our Contribution

Consider the following setting: fix a vertex set $[N] = \{1, \dots, N\}$, where N is a positive integer, and connect the vertices in $[N]$ as follows. First, add a set of fixed edges $(i, i+1)$ constituting a cycle $(N+1 \equiv 1)$. Then, have each node $n \in [N]$ maintains two pointers: a blue pointer to a node $b_n \in [N]$, and a red one to $r_n \in [N]$, such that each node n is the destination of exactly one blue pointer and one red pointer. In essence, b and r constitute permutations over $[N]$. From these, we construct a set of undirected blue edges $\{(n, b_n) : n \in [N]\}$, and a set of undirected red edges $\{(n, r_n) : n \in [N]\}$. The result is a 6-regular graph G over $[N]$ composed of N cycle edges, N blue edges and N red edges.

The dynamic then proceeds as follows. The graph evolves through alternating red and blue phases; during a blue phase, only the blue pointers are modified, while the red pointers are kept fixed. The blue pointers slide along the graph G^r formed by the union of the cycle edges and the undirected edges $\{(n, r_n) : n \in [N]\}$ formed by the red pointers. For the red phases, the roles of the blue and red pointers are swapped.

Formally, the dynamic for the blue edges over G^r is as follows: at each time step, pick an edge (i, j) uniformly at random in G^r, and denote n and m the two nodes in $[N]$ such that $b_n = i$ and $b_m = j$. These two nodes swap their pointers: now $b_n = j$ and $b_m = i$. Notice that b is still a permutation over $[N]$. This dynamic is known in the literature as the interchange process [11, 3]. Our main result is then as follows:

▶ **Theorem 1.** *Let $T = N \ln(N)^a$ where $a > 8$ is a constant. Then with high probability, after $O(\ln(N))$ alternating phases of length T, the blue and red pointers constitute uniformly and independently distributed permutations of $[N]$.*

▶ **Corollary 2.** *With the above process, G is an expander with high probability after each node has performed only $O(\operatorname{polylog}(N))$ operations.*

4 Sketch of Proof

For any d-regular graph H over $[N]$, denote $\phi_k(H) = \min_{S \subset [N] : 0 < |S| \leq k} \frac{|\partial S|}{d|S|}$, where ∂S is the set of edges between a node in S and a node not in S. Exploiting properties of the interchange process stated in [3], we show that, if $\phi_{N/2}(G^r) \geq \gamma$ for a well chosen γ, dependent on N, then at the end of the following blue phase, the blue pointers are uniformly distributed with high probability.

We then construct an increasing sequence $(k_t)_{t \in \mathbb{N}}$ such that i) at the end of the t-th blue/red phase, $\phi_{k_t}(G^{b/r}) \geq \gamma$ and ii) there exists $\tau = O(\log(N))$ such that $k_\tau = N/2$. Joining the dots proves the theorem.

Acknowledgements. The authors would like to thank George Giakkoupis for bringing the problem studied in this paper to their attention, and to acknowledge stimulating discussions on the topic with both George Giakkoupis and Marc Lelarge.

References

1. Réka Albert and Albert lászló Barabási. Statistical mechanics of complex networks. *Rev. Mod. Phys*, page 2002.
2. Zeyuan Allen Zhu, Aditya Bhaskara, Silvio Lattanzi, Vahab S. Mirrokni, and Lorenzo Orecchia. Expanders via local edge flips. *CoRR*, abs/1510.07768, 2015.
3. N. Berestycki. Mixing times of markov chains: Techniques and examples. *Alea-Latin American Journal of Probability and Mathematical Statistics*, 2016. URL: `http://www.statslab.cam.ac.uk/~beresty/teach/Mixing/mixing3.pdf`.
4. Shankar Bhamidi, Guy Bresler, and Allan Sly. Mixing time of exponential random graphs. *The Annals of Applied Probability*, 21(6):2146–2170, 12 2011.
5. Colin Cooper, Martin Dyer, and Andrew J. Handley. The flip markov chain and a randomising p2p protocol. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, PODC'09, pages 141–150, New York, NY, USA, 2009. ACM. `doi:10.1145/1582716.1582742`.
6. Colin Cooper, Ralf Klasing, and Tomasz Radzik. A randomized algorithm for the joining protocol in dynamic distributed networks. Technical Report RR-5376, INRIA, November 2004. URL: `https://hal.inria.fr/inria-00070627`.
7. Rick Durrett. *Random Graph Dynamics*. Cambridge University Press, Cambridge, 2007. URL: `http://www.math.cornell.edu/~durrett/RGD/RGD.html`.
8. Tomas Feder, Adam Guetz, Milena Mihail, and Amin Saberi. A local switch markov chain on given degree graphs with application in connectivity of peer-to-peer networks. In *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science*, FOCS'06, pages 69–76, Washington, DC, USA, 2006. IEEE Computer Society. `doi:10.1109/FOCS.2006.5`.
9. Riko Jacob, Andrea Richa, Christian Scheideler, Stefan Schmid, and Hanjo Täubig. A distributed polylogarithmic time algorithm for self-stabilizing skip graphs. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, PODC'09, pages 131–140, New York, NY, USA, 2009. ACM. `doi:10.1145/1582716.1582741`.
10. M. Jerrum and Alistair Sinclair. Approximating the permanent. *SIAM J. Comput.*, 18(6):1149–1178, dec 1989.
11. Johan Jonasson. Mixing times for the interchange process. *Alea-Latin American Journal of Probability and Mathematical Statistics*, 9(2):667–683, 2012.
12. Minkyu Kim and Muriel Medard. Robustness in large-scale random networks. *INFOCOM*, 2004.
13. Manos Papagelis. Refining social graph connectivity via shortcut edge addition. *ACM Trans. Knowl. Discov. Data*, 10(2):12:1–12:35, oct 2015.
14. Jason Schweinsberg. An o(n2) bound for the relaxation time of a markov chain on cladograms. *Random Struct. Algorithms*, 20(1):59–70, jan 2002.
15. Lingsheng Shi and Nicholas Wormald. Models of random regular graphs. In *IN SURVEYS IN COMBINATORICS*, pages 239–298. University Press, 1999.